# The Research Computing User’s Guide

October 16, 2023

Jason W. Bacon

# Contents

# List of Figures

# List of Tables

October 16, 2023

## 0.1 Acknowledgments

Many thanks to (in alphabetical order) Aishwarya Shrestha, Anisha Tasnim, and Shamar Rhadre Webster, for extensive feedback and corrections.

## 0.2 Practice Problem Instructions

- Practice problems are designed to help you think about and verbalize the topic, starting from basic concepts and progressing through real problem solving.
- Use the latest version of this document.
- Read one section of this document and corresponding materials if applicable.
- Try to answer the questions from that section. If you do not remember the answer, review the section to find it.
- Do the practice problems *on your own*. Do not discuss them with other students. If you want to help each other, discuss *concepts* and illustrate with different examples if necessary. Coming up with the correct answer on your own is the only way to be sure you understand the material. If you do the practice problems on your own, you will succeed in the subject. If you don't, you won't.

  If you're still not clear after doing the practice problems, wait a while and do them again. This is how athletes perfect their game. The same strategy works for any skill.
- Write the answer in your own words. Do not copy and paste. Verbalizing answers in your own words helps your memory and understanding. Copying does not, and it demonstrates a lack of interest in learning.

  Answer questions completely, but *in as few words as possible*. Remove all words that don't add value to the explanation. Brevity and clarity are the most important aspects of good communication. Unnecessarily lengthy answers are often an attempt to obscure a lack of understanding and may lead to reduced grades. "If you can't explain it simply, you don't understand it well enough." -- Albert Einstein
- Check the answer key to make sure your answer is correct and complete.

  DO NOT LOOK AT THE ANSWER KEY BEFORE ANSWERING QUESTIONS TO THE BEST OF YOUR ABILITY. In doing so, you only cheat yourself out of an opportunity to learn and prepare for the quizzes and exams.
- ALWAYS explain your answer. No exceptions. E.g., justify all yes/no or other short answers, show your work or indicate by other means how you derived your answer for any question that involves a process, no matter how trivial it may seem, draw a diagram to illustrate if necessary. This will improve your understanding and ensure full credit for the homework.
- Verify your own results by testing all code written, and double checking short answers and computations. In the working world, no one will check your work for you. It will be entirely up to you to ensure that it is done right the first time.
- Start as early as possible to get your mind chewing on the questions, and do a little at a time. Using this approach, many answers will come to you seemingly without effort, while you're showering, walking the dog, etc.
- For programming questions, adhere to all coding standards as defined in the text, e.g. descriptive variable names, consistent indentation, etc.

## 0.3 Best Practices for Students and Instructors

### 0.3.1 Take Ownership of your Education

Read this text, and everything else, with a critical eye. Don't fall for the *appeal to authority* fallacy, believing that the author of a book is an expert and therefore must be right. It's almost certainly true that someone who wrote a book about a subject knows much more than you do, but they are not infallible. They make mistakes and still have a few misconceptions despite all the experience and research that went into writing the book. The only way to be certain of any assertion is by checking the facts for yourself, or applying sound logic to infer conclusions that available facts do not indicate directly.

On that note, don't blame your teachers, book authors, or anyone else for your misconceptions, even if they did misinform you. Doing so only highlights gullibility. To quote Obi-Wan Kenobi: "Who's the more foolish, the fool or the fool who follows?"

Go well beyond what you learn in your classes. Your teachers know a tiny fraction of what you'll need to know during your career. They only have time to teach you a tiny fraction of what they know. If all you know when you graduate is what you were spoon-fed in lectures, you won't have much to offer your employer. All employers care about grades, but the better ones care more about what you've done *beyond* your classes. This shows a real interest in your field and shows the ability to solve problems independently. Develop this ability while in college so that potential employers will see you as an asset to their team.

### 0.3.2 Note to Lecturers

Instructors should maintain a reasonable pace in lectures. Be thorough, but don't rush. Give students the opportunity to ask questions during lecture. On the other hand, don't try to make every student understand perfectly during lectures. This is not possible. Most learning comes from practice outside of class. If the course includes labs or discussions, allow them to serve a purpose as well. The purpose of lectures is to give students material to think about and practice, so that the time they spend practicing outside of lecture will be productive.

If there is a lab/discussion associated with this course, one simple example should suffice for each topic in lecture. Additional examples can be covered in lab/discussion and in the homework. If there is no lab/discussion, then an additional example may be appropriate in some cases, but students should still be expected to practice outside of class.

### 0.3.3 Making the Most of Class Time

Learning results from TIME and REPETITION. Lectures only provide material for students to practice. Don't expect to leave a lecture, discussion, or lab session of any class with a deep understanding of the material. Take detailed notes in class, read the course materials, and then immediately start practicing by trying to put it to use. We provide an extensive set of practice questions for this very purpose. Without practice to cement in the concepts, you will forget quickly. Use it or lose it.

Study early so your brain has time to digest the material. Study often to reinforce the neural connections that make up long-term memory. The learning process literally involves rewiring your brain, which is a slow biological process that cannot be accelerated.

**Note** OK, that's a white lie: It has been shown that traumatic experiences result in long-term memory comparable to an extensive amount of practice. However, scaring the pants off of students is not a practical way to help them learn.

Everyone should take pencil and paper notes during lectures rather than rely on online materials. Take notes on *everything*, even if you think you know it already. Review these notes first to ensure there are no major gaps in your knowledge. The act of writing or explaining something has a powerful effect on memory and understanding. Writing something once does as much for your memory and understanding as reading it ten times. Don't sit back and be passive about your education. That strategy will backfire. You'll learn much more with less effort by actively engaging.

### 0.3.4 Run the Whole Race

Make sure you get off to a good start so the rest of the semester won't be a struggle to catch up.

If you do have a good start, don't fall victim to the common tendency to think you can coast for a while. Some topics will be harder for you than the ones you just aced. What's hard varies from student to student, so ignore what others are saying about it, and just put in the amount of effort that *you* need to. Be thorough about studying every topic throughout the semester, regardless of how you've done on previous topics. If you just do that, you'll do well overall.

### 0.3.5 Abandon your ego

In general, if you want to know whether you want someone in your life, observe them for a while and see if they can laugh at themselves. If not, smile, walk away, and don't look back.

One of the most important goals in any scientific education is to get over the fear of being wrong. Ego is the enemy of real science and engineering. Abandon it. Learn to feel comfortable making suggestions and having them shot down. Maybe it was a good suggestion and others just aren't seeing it. Maybe it was a dumb idea and you're not seeing it. Don't get upset either way. Laugh it off for now, keep thinking about the problem, and let the situation play out over time.

Transition your thinking from "My code sucks, what am I doing in this field?" to "My code sucks, how can I improve it?". Top-notch scientists and engineers are completely humble, emotionless, and objective about their work. They abandon bad ideas without hesitation, embarrassment, remorse, and focus all their energy on finding better ones. They are *grateful* when others point out their mistakes.

The sooner we let go of bad ideas, the less time we will waste trying to make them work, and the more time we can spend at the beach. A happy, balanced engineer will accomplish more in 8 hours a day than one who struggles for 16 hours a day and has no fun because [s]he can't admit a mistake. Take your pick. It's entirely up to you which one you want to be.

All that really matters is that we keep moving forward. It won't always be quickly enough to get straight A's, and that's fine. Just put in a solid effort and you will grow as a result. Growth is more important than grades.

### 0.3.6 Why do Quality Work?

Every customer wants a quality product, but what's the real motivation for creating them? Why should we write fast, reliable programs? Why design fast, reliable, inexpensive hardware? So the boss will give us a raise? Probably not. Most bosses wouldn't recognize quality work if it licked their face. So we'll be admired by our peers? No, doesn't really work. Most of them will just become jealous and trash you on social media.

Think about how often you've wasted time waiting for something that seems inexplicably slow, or worse, breaks down so you have to start over. As a result, you missed happy hour, your kid's soccer game, or something else you were really looking forward to. Low quality products cause massive amounts of wasted time and aggravation. The best reason to do quality work is to help everybody (including yourself) get their work done quickly and correctly, so we can all spend more time with our families and friends. Quality work makes everybody's lives better. This is how you can have a positive impact and garner real appreciation as an engineer.

So how to we get there? Some would say "take pride in your work". But this often backfires, because it depends on what makes an engineer proud. Many engineers are proud of how clever they are. While a normal person would say "If it ain't broke, don't fix it.", many engineers say "If it ain't broke, it doesn't have enough features yet.". Clever engineers make things needlessly complicated to prove that they're clever. Wise engineers make things as simple as possible so they will be reliable, inexpensive, and easy to use. Remember this simple equation:

cleverness * wisdom = constant

Remember KISS (Keep It Simple, Stupid). If you follow this ideal, you'll be a top-notch engineer.

### 0.3.7 Stick to the course materials

The materials provided for this course are all you should need to succeed. Do not trust alternative information from web forums such as stackoverflow.com, geeksforgeeks.org, etc. These sites do not provide curated information. Anyone with a web browser can post their opinions there. Most of the information on these sites ranges from suboptimal to complete rubbish. If you really must look to web forums for information, be sure to verify any assertions you find there via experimentation or more reliable sources. Never believe the first answer you find on a web forum.

Outside sources should *only* be used to help you understand the course materials, and rarely for this purpose. They should never be trusted as a substitute. If anything in the course materials is unclear, it is better to contact the instructor than to use outside materials for clarification. This will prevent you from getting things wrong, and will help the instructor improve the course materials.

If there is anything in the course materials you don't understand, RISE TO THE CHALLENGE IMMEDIATELY and make sure you master the material. Don't try to work around it by finding a quicker, easier way to get the homework done. Doing the latter will only cause you to fall behind in the class and you will not do well in the end.

### 0.3.8 Homework, quiz, and exam format

Most questions are short answer, coding, or diagram format rather than true/false or multiple choice. The act of explaining a concept goes a long way toward helping you remember and understand it, so writing out the answer in your own words is a far better learning experience than picking the answer out of a list.

In fact, you can help yourself understand the material better by explaining it to your mom, your cat, or anyone else with the patience to listen to nerdy ramblings about computer science.

Also, the real world is not multiple choice. Good luck finding a job where your boss solves all the problems and pays you to pick the correct solution from among several incorrect ones. The real world is open book, but it also has time limits, so you do not want to rely on references entirely. You need important knowledge internalized in order to be productive. The goal here is to practice for that scenario.

## 0.4 Motivation and Goals

### 0.4.1 Why Are We Here?

The difference between what is being accomplished in scientific computing today and what could be accomplished, using existing inexpensive and free tools, is staggering. Many researchers spend months struggling to do simple computational analyses that could be done in a few hours with the right tools and knowledge. Worse yet, they often give up, leaving potentially life-saving research unfinished. The entire reason for all of this unrealized potential is a simple lack of proper education.

A major problem in scientific computing is people who know just enough to be dangerous. Many computational scientists know a little Unix, a little scripting, a little Python, and a little C or C++. This leads to badly designed programs and scripts that waste computing resources and make it difficult or impossible to reproduce results, a cornerstone of all science.

This book and this course are here to address these issues by showing how to use many of the amazing tools available for free, manage the software you need with minimal effort, and write software of your own as efficiently as possible. We will focus on *depth*, not *breadth*, so that you will learn how to do things *well*. The hope is that you will then carry these good habits forward as you develop more breadth of knowledge. Perhaps someday you will teach others as well, so that the benefits of this knowledge will continue to spread far and wide.

### 0.4.2 Damn it Jim, I'm a Scientist, not a Systems Programmer...

Why should researchers learn about computing? Because nobody can do it for you. Some researchers cling to the dream of hiring computer staff to handle these things, so they won't have to learn and do it themselves. This is unrealistic for several reasons:

- There aren't enough computer experts around to fill even a small fraction of the needs in scientific research.
- Even if the talent existed, we couldn't afford it. Most computer professionals with these skills are earning six figure salaries. This is more than most PIs (principal investigators) doing scientific research will ever earn.
- Computer staff would need to be trained in your field of research before they can do anything more than manage computers for you. Writing software to conduct computation research requires a level of expertise in the domain typically only achieved by experienced researchers.

The bottom line is, if you don't have the computing skills to do your own computational analysis, your research will likely be severely delayed or stalled entirely.

There is an enormous gap between what many researchers want and what is feasible. Many researchers wish for easier ways to do their research computing, including graphical user interfaces or web-based interfaces to software. Convenient point-and-click interfaces help people avoid learning the things they fear such as the Unix command line, scripting, and programming. However, the manpower to build and maintain these interfaces does not exist and never will. Building and maintaining convenient user interfaces is astronomically expensive and the research community has only a tiny fraction of the resources necessary to pay for it. The interfaces that are developed tend to be very limiting and unreliable.

Avoidance is counterproductive and futile. It not only delays the inevitable need to learn other approaches, it also wastes resources attempting to create more convenient interfaces that will likely never be finished or maintained. If you do manage to find a convenient user interface that works for you, chances are it will be abandoned soon, which means your research will not be reproducible.

Basic tools, such as the Unix command line and scripting languages, are widely used by a broader audience than the research community. As a result, they are better maintained, more reliable, and better supported. By using them, we leverage the vast resources of other industries much wealthier than we are. Learning to use them now will be much easier than resisting and giving into the inevitable later. It will also mean more research progress in the meantime.

I spent nearly ten years holding together an understaffed research computing support group. During that time, it became clear that struggling to hire and retain the support staff needed to assist thousands of researchers across campus was a hopeless cause. The only solution to the support problem is to make the researchers more self-sufficient. The required skills are not difficult to learn. A small investment of time will produce a huge payoff for your research output.

### 0.4.3 Solutions Looking for Problems

Beware the tendency to confuse tools with fields of study. Many professionals deliberately study specific technologies for their own sake, such as virtualization, containers, machine learning, specific languages and operating systems, etc. It is often necessary to focus on the technologies while learning them, but not such a good idea when *applying* them. E.g. using machine learning to optimize the organization of your sock drawer might be useful as a learning exercise, but would be foolish as a real-world solution.

Problems arise when we try to apply the tools we have invested in learning to solve every new problem they encounter. This is a backward approach to problem-solving that usually leads to a very suboptimal solution at best. It is unfortunately a common mistake, however. When all you have is a hammer, everything looks like a nail. Shoe-horning problems into a solution that you think is "cool" or one that you already know generally leads to wasted effort and computing resources. Unfortunately, the world is full of techno-geeks selling over-complicated solutions to simple problems that could be handled much more cost-effectively.

To find the best solution for a problem, we must look for the best solution for the problem. Sound obvious? It is. Nevertheless, many people don't think this way and instead look for ways to solve it with their favorite tools. Finding the optimal solution means examining the problem with an open mind and exploring solutions that we *don't* know as well as the ones we do. A willingness to learn new things every time you take on a new project is the difference between a great engineer or scientist and a mediocre one.

### 0.4.4 What You will Learn

The overall goal of this user guide is to provide all the knowledge needed for researchers to get started with research computing. If you're a researcher, you will learn to be self-sufficient so that your computational analysis can move forward in the absence of I.T. staff to help you. If you are one of the few I.T. personnel working in scientific computing, you will learn to use your time as efficiently as possible, so that you can effectively serve the researchers who vastly outnumber you.

Different users have different needs and most will not need to read all of the chapters of this guide. The guide is divided into four parts, each of which is focused on the needs of typical types of researchers. You may only need the knowledge presented in one or two parts, or you may need it all!

After reading this document, you should know:

- How computers are commonly used in scientific research
- How to find (or build) and use available computing resources
- How to use Unix-compatible operating system environments, including BSD, Cygwin, Linux, and Mac OS X
- How to write portable shell scripts to automate the execution of your research tools on any operating system
- The types of parallel computing available today
- How to schedule typical jobs on clusters and grids

- Where to find more detailed information on all of the above

This document and other information can be found at:

https://acadix.biz/publications.php

## 0.5 Self-Study Instructions

**Note** This guide is updated frequently. Printing is recommended only for those who own stock in a paper company.

This guide is organized as a tutorial for users with little or no experience using the Unix command line or parallel computing resources.

If your institution does not offer a course following this text, you might consider registering for an independent study that requires turning in the practice problems at the end of each section. This will provide motivation to master the material rather than just read it and move on. You and the course supervisor should review the guide and select the appropriate chapters and sections for your study at the beginning of the semester.

### 0.5.1 Unix Self-Study Instructions

To begin learning the Unix environment, readers should do the following:

1. Get access to a Unix system if you don't have it already. You will need this to practice running Unix commands and writing basic scripts. Apple Macintosh, BSD and Linux systems are all Unix compatible. If you are running Windows, you can quickly and easily add a Unix environment to it by installing Cygwin following the instructions in Section 3.4.1.
2. Thoroughly read Chapter 3 up to and including Section 3.9. The remaining sections can be covered later after gaining some hands-on experience. Do the self-test at the end of each section.
3. Thoroughly read Section 4.1 through Section 4.5. Do the Self-test at the end of each section.

### 0.5.2 Parallel Computing Self-Study Instructions

To begin learning the basics of parallel computing, readers should do the following:

1. Thoroughly read Chapter 6 and Chapter 7.
2. Read Chapter 8, but don't expect to understand it perfectly. Just familiarize yourself with the material so it will be easier to master during your first meeting with a facilitator.
3. If you plan to use the HTCondor pool, skim over Chapter 9.

### 0.5.3 Instructor's Guide

A typical 3-credit semester course with 2.5 hours per week lecture time should be able to easily cover all of Part I, introduce parallel computing concepts and the SLURM scheduler, and possibly touch on Part III, High Performance Programming.

Knowledge of various computational methods is key to helping researchers find the most elegant solution to their research problems, and avoid the "solutions looking for problems" mentality, which often leads to using overly complex approaches. Students must be taught to focus first on the problem, explore all potential solutions, and choose the simplest among them, rather than the sexiest. Many will gravitate toward using machine learning, GPUs, parallel programming, etc. in order to impress people, where much simpler solutions would have worked.

A solid base in the Unix command-line and shell scripting is important, as most researchers waste time or outright fail to succeed due to lack of knowledge in these areas. I've seen many cases where badly written scripts slow down an analysis by an order of magnitude or more. Also problematic are non-portable programs and scripts that work on Ubuntu, but not RHEL, or on most Linux systems, but not Mac or vice versa. The Unix and scripting chapters emphasize portability and provide guidance on how to avoid non-portable "-isms".

A quick introduction to high performance programming is highly valuable, since most incoming students (even computer science students) will not know the difference between compiled and interpreted languages, or that software can be made to run significantly faster with an understanding of memory hierarchy and other hardware specifics. Just raising awareness of what is possible with the right software development choices will help them avoid wasting time going sideways.

### 0.5.4 Digging Deeper

The later sections of high performance programming and parallel computing, along with the parallel programming chapter, are best tackled after becoming comfortable with basic HPC/HTC usage, or in separate courses.

OK, enough talk. Let's get you guys edumacated...

# Part I

# Research Computing

# Chapter 1

# Computational Science

## 1.1 Introduction

### 1.1.1 So, What is Computational Science?

Nope, it's not the study of computation. That would be *computer science*.

*Computational science* is any method of scientific exploration involving the use of computers. This may involve using computer models directly for experimentation or using computers to analyze data from experiments performed by other means.

### 1.1.2 Not Just for Nerds Anymore

Computation has been a core part of mathematics, physics, chemistry, and engineering research for decades. It is rapidly gaining popularity in other areas of research such as biology, psychology, economics, political science, and just about any other discipline you can think of.

This trend is due in part to the introduction of other technologies into these fields, such as rapid gene sequencers and imaging technology such as MRI (Magnetic Resonance Imaging). These new technologies generate vast amounts of data that require significant computing resources to store and process. Researchers in these fields often spend the majority of their time on the computer and only a small fraction in the wet lab. If you don't like computer work, you may want to reconsider becoming a geneticist or MRI researcher.

The trend is also due to computer technology itself facilitating the storage and use of vast amounts of data in all walks of life. The evolution of fast, cheap computer technology and the Internet has made it possible to archive detailed records of things like election results and sales records and make them easily available to almost anyone in the world. There are many researchers these days who don't collect their own data, but simply use archives collected by others in the past.

### 1.1.3 The Computation Time Line

In doing computational science, we ultimately have a few key goals:

- Minimize the *wall time*, the total time from the moment we decide what we want to do, to obtaining good quality results from the software we run.
- Minimize man-hours, the amount of time we spend doing manual work.
- Minimize computer time, the amount of time we wait for the computer to do its part.

These goals almost always go hand-in-hand, but occasionally there may be a trade off, where we sacrifice more man-hours to get results sooner, or accept a delay in results to save precious man-hours.

The figure below represents the time line of computational science project, showing typical time requirements for developing the software, deploying (installing) the software, learning the software, and finally running the software.

Any one of these steps could end up taking the majority of your time, so we need to consider all of them when devising a strategy for achieving our goals. This text will discuss ways to minimize the time required for each step as well as potential trade-offs involved.

| **Development Time** | **Deployment Time** | **Learning Time** | **Run Time** |
|---|---|---|---|
| Hours to years | Hours to months (or never) | Hours to weeks | Hours to months |

Table 1.1: Computation Time Line

### 1.1.4 Development Time

Developing software is inherently time-consuming. Large programs may require many thousands of man-hours to specify, design, implement, and test.

Fortunately, most researchers do not need to write major software of their own. There are many commercial and open source programs available to assist in a wide range of research methods. Many researchers will need to do some level of programming, however. If there is no quality free software for your research, the cost of commercial software or hiring a programmer may be beyond your means and likely to cause major delays even if you can afford them. It is a good idea to learn now and practice regularly so you're ready when you have to write some code of your own.

### 1.1.5 Deployment Time

Deploying software can and should be quick and easy. Unfortunately, many researchers are not aware of efficient deployment methods and often end up wasting time or even failing entirely to get software installed. This is a major obstacle to important research, which we will discuss in detail a bit later.

All software installations should be doable in seconds or minutes. They should not require hours, days, weeks, or months. If they do, then either you or the software developers are doing something wrong. Don't accept difficult software installations as a normal part of doing research.

### 1.1.6 Learning Time

Learning time is largely a matter of focus and quality of documentation. All I can offer here is some advice: Do your homework, locate the best sources of documentation, and invest some time in studying it without distraction.

### 1.1.7 Run Time

Run time depends on many factors, such as the algorithms used by the program, the language used to implement it (compiled languages are many times faster than interpreted as discussed in Section 13.5), and the hardware it runs on.

Many scientific programs are extremely poor quality and could be made to run hundreds or even thousands of times faster with the right programming skills. Optimizing the software should always be done before throwing more hardware resources at it. When it is not feasible to make the program faster, one might consider using parallel computing resources. However, doing so before optimizing the software would be an unwise and possibly unethical waste of costly resources. There is also a steep learning curve involved in using parallel resources that can be avoided by improving the program first.

### 1.1.8 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What kind of research are you currently conducting, and how might computers be used to further your goals?
2. How does computational science differ from computer science?
3. What areas of research benefit from computational science?
4. Describe two reasons that computational science is growing so rapidly.
5. What are the major goals in the computational time line?
6. What are the major steps in the computational time line? Which one takes the longest?
7. Can researchers avoid programming entirely? Why or why not?

## 1.2 Common Methods Used in Computational Science

Some of the most common computational science techniques are described below. However, this is not meant to suggest that they are the only ways to use computers for research. The only true limits are imposed by your own imagination.

### 1.2.1 Numerical Analysis

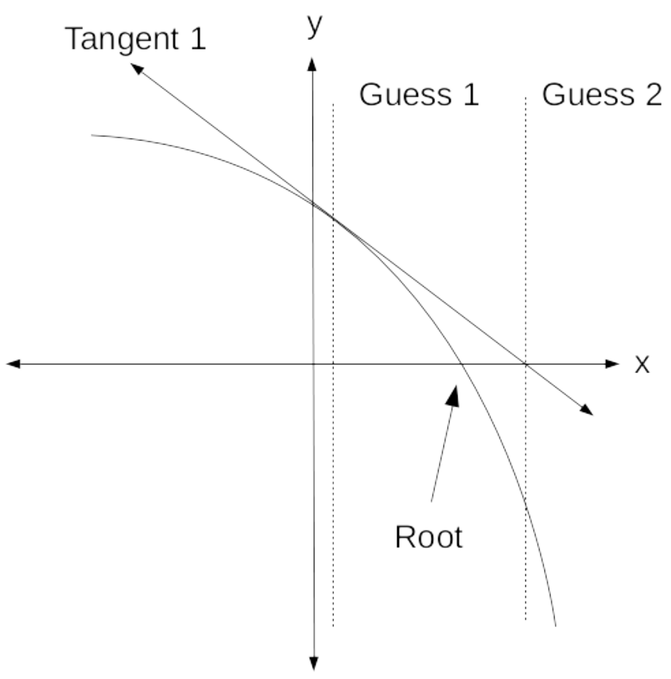
Numerical analysis applies to many processes that occur in the real-world that can be modeled by mathematical equations. Unfortunately, many of these equations, while seemingly simple, cannot be solved directly by known analytical methods. The techniques taught in algebra, trigonometry, and calculus courses, while extremely powerful, apply to only a small fraction of the complex real-world models that researchers encounter.

Numerical analysis takes care of the rest. For many models where it is difficult to find the answer, but easy to verify it, numerical analysis can be used to produce an approximation as precise as we want. Numerical techniques generally involve clever techniques to successively improve guesses at the answer or convert the model into a system of equations which can be solved directly. Both of these methods are tedious to perform by hand, but well-suited to a fast computer.

The classic first example often used in numerical analysis courses is Newton's method for estimating the roots of an equation (where the graph of the equation crosses the x axis). The aim is to answer the question: "For what value(s) of x does f(x) = 0?".

While finding the roots of some equations is not easy to do directly, it is generally easy to compute f(x) for any value of x and see how close it is to 0.

Now for the clever part: In most cases, a line tangent to f(x) will cross the x-axis at a point closer to a root than x. This can be visualized by drawing a graph of an arbitrary function, and a series of tangent lines.



The equation for the tangent line is easily computed using x, f(x), and f'(x), the slope of the curve at f(x). The root of the tangent line (where the tangent line crosses the x axis) is then easily calculated using the equation for the line. This becomes the next "guess" for the root of the equation. With rare exceptions, this next guess for x will be closer to the root than the previous x. We continue this process until f(x) is sufficiently close to 0 or until the difference between subsequent guesses at x is sufficiently small.

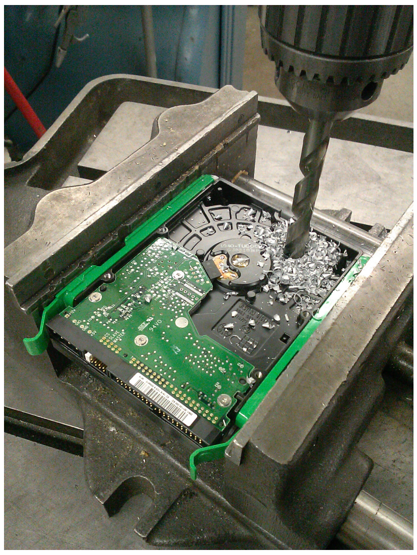
### 1.2.2 Computational Modeling

Computational models simulate real-world processes, following the relevant laws of math and physics.

Models might simulate the motion of individual molecules in a fluid or solid, or larger cells of fluid such as oil in an engine, water in the ocean, or air in the atmosphere. The weather forecasts we rely on (and complain about) are determined mainly through the use of large-scale fluid models utilizing measurements of temperature, pressure, humidity, and wind throughout the planet.

Models are also used to simulate traffic flow on roads and expressways that have not yet been built, usage patterns in buildings still in design phase, and population growth and collapse in remote ecosystems, to mention just a few more cases. Scientists, engineers and architects use models to find out things in advance that could otherwise get them in trouble, like "Will closing a lane during rush hour cause a traffic jam?" or "What will happen if we only put a bathroom on every other floor?".

### 1.2.3 Data Mining



The term "mining" traditionally refers to digging through vast amounts of earth to find small amounts of valuable minerals. The minerals are generally a very small fraction of the earth that's removed, and it requires a lot of work to find and separate them.

There are also vast amounts of data stored on computers that contain small amounts of information of interest to a particular researcher. Data mining is the process of sifting through these data for "interesting" information.

There are many possible approaches to data mining. The ultimate goal is to have a computer search through huge archives or databases of information without human intervention and automatically identify items or patterns of interest. This is not always feasible, so a more practical goal is often to have the computer do as much as possible and simply minimize the human labor involved.

Depending on the type of information being searched, teaching a computer to identify truly interesting patterns can be fairly difficult and may require the use of artificial intelligence techniques. For example, a now-famous data mining experiment used to search hospital records for patterns initially reported that all of the maternity ward patients were women.

This underscores the fact that while computers are powerful tools that can do many things far faster and more accurately than humans, there are still many tasks that require human knowledge and reasoning.

### 1.2.4 Parameter Sweeps

Numerical analysis uses techniques to progressively improve guesses at the solution to a problem. Unfortunately, sometimes we can't come up with a clever method of improving on our current guess and we simply have to test every possible answer until we find one that works.

A parameter sweep tests a range of possible answers to a question, until at least one correct answer is found, or until all possible answers have been checked in order to determine a near-optimal set of parameters. It is a brute-force approach to answering questions where directly computing the answer is not feasible.

It may involve repeating a set of calculations with numerous combinations of multiple parameters in order to determine the optimal set of parameters.

An example involving a single parameter is the brute-force password hack. Passwords are stored in an encrypted form that cannot be directly converted back to the raw password. Since it is the raw password that must be entered in order to log into a computer, this effectively prevents unauthorized access even if the encrypted passwords are known. This is important since many passwords must be transmitted over networks in order to log into remote systems such as email servers. Hence, it is often difficult or impossible to prevent encrypted passwords from becoming known.

While it is practically impossible to decrypt a password, it is relatively straightforward to encrypt a guess and see if it matches the known encrypted form.

The main defense against this type of brute-force attack is forcing the attacker to try more guesses, i.e. maximizing the parameter space that must be swept. An 8-character password randomly consisting of both upper and lower case English letters, digits, and punctuation has $(26 + 26 + 10 + 32)^8 = 6.09 \times 10^{15}$ possible patterns (based on a US-English keyboard).

If a computer can encrypt and compare 100,000 guesses per second, it will take 1,932 years to sweep the entire parameter space. On average, it will take half that time to find one particular password with these qualities.

On the other hand, if the attacker knows that your password is a 10-letter English word with a mix of upper and lower case, then based on the size of the Oxford English dictionary (about 170,000 words), there are only about $(170,000 * 2^{10}) = 174,080,000$ possible passwords. At 100,000 guesses per second, it would take the hacker's tools at most 1741 seconds = 29 minutes to find your raw password.

For this reason, a password should never be any kind of derivative of a real word.

The worst kind of password, of course, is anything containing personal information. Many computer users think they're outsmarting hackers by putting a digit or two after their name to form a "secure" password. Hackers have a standard list of items commonly used by people just begging to get hacked, such as their name, birthday, pet's name, favorite color, etc. Most of this information is readily available online thanks to sites like Facebook. Checking every item in this list followed by every possible number from 1 to 999 (e.g. from azure0 to zebra999) will take only a fraction of a second on a modern computer.

### 1.2.5 Data Sifting

In some cases, there are just gobs and gobs of raw data to be sifted and checked for known or expected patterns. This is different from data mining in the sense that the human programmers know what to look for.

Examples of this type of computational research are well illustrated by the @home projects, such as Einstein@Home, which searches data from laser interferometer gravitational-wave observatory (LIGO) detectors for evidence of gravity waves. The LIGO detectors unfortunately don't beep when they spot a gravity wave. Instead, they generate enormous amounts of data, most of which will not show any evidence of gravity waves, but nevertheless must be examined thoroughly. The Einstein@Home project uses massive numbers of personal computers around the world, each sweeping a small segment of the LIGO raw data.

### 1.2.6 Monte Carlo

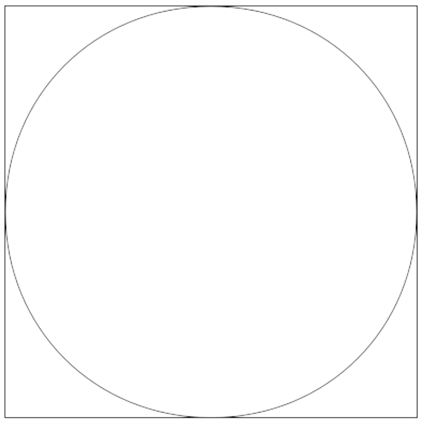
Monte Carlo simulations, named after the gambling city in the French Riviera, utilize random numbers and simulation to piece together answers to scientific questions.

The method actually looks similar to a parameter sweep or data sift in that the same calculations are done on a large number of different inputs. However, the Monte Carlo method uses random inputs rather than a predetermined set of inputs. The random numbers generated are generally designed to be representative of the entire possible range, while being a fraction of the size.

For example, the average of a small, but truly random sample of a population is generally very close to the true average of the entire population. This mathematical fact makes many experiments possible in the humanities and social sciences, where sampling every member of a society is practically impossible.

A classic example of the Monte Carlo method is the estimation of the value of pi using a dart board.

Suppose we have a square dart board with a circle inscribed:



The area of the circle is PI * $(1/2)^2$ = PI/4. The area of the square is 1. The ratio of the area of the circle over the area of the square is therefore PI/4.

If a bad enough darts player throws a large number of darts at the board, darts will end up randomly and uniformly distributed across the square board (and probably the surrounding wall). If and only if the darts are randomly and uniformly distributed, the ratio of the number of darts within the inscribed circle over the number of darts within the entire square board will then be close to pi/4.

Statistically, the more darts are thrown, the closer the ratio will get to pi.

Naturally, this process would take too long with a real dart board, so we might instead choose to simulate it on a computer. Most programming languages offer the ability to generate pseudo-random numbers within some fixed range with a uniform distribution. By randomly generating a sequence of x and y values with a uniform distribution, we can rapidly simulate throwing darts at a board and quickly develop an estimate for pi.

### 1.2.7 Everything Else

The previous sections outline some of the commonly used methods in computational science.

Researchers can explore and understand these methods and also discover or invent new methods of their own for using computers in their research. The computational capacity of today's computers is both vast and vastly underutilized. The possibilities for computational research are almost limitless and bounded only by the skills and imagination of the researcher.

It is our hope that more researchers will simply begin to consider how computational methods might improve their research and then develop the skills and knowledge to tap into the vast and freely available hardware and software resources that are waiting to be utilized.

### 1.2.8 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why are numerical analysis techniques so important to science and engineering?

2. Explain Newton's method for finding the roots of a function. Use a graph to illustrate.
3. What is computational modeling? Describe two examples of processes commonly modeled on computers.
4. What is data mining? What is one of the major challenges in designing useful data mining software?
5. What is a parameter sweep? Describe one task that requires a parameter sweep. Is doing parameter sweeps desirable?
6. What is data sifting? Describe one real-world example that requires data sifting.
7. What is a Monte Carlo simulation?
8. Can you think of any computational science methods that do not fall into one of the categories described here?

## 1.3 Venture Outside the Computer Science Bubble

If you're a computer scientist, you likely have a different perspective on computers than people in other fields. Computer scientists tend to focus on learning about technologies and look for ways to apply them as an afterthought, i.e. the solutions looking for problems perspective.

Researchers in other fields often, but not always, take the opposite perspective, focusing on the problem and looking for technologies with which to solve it. The solution may or may not involve computers. If it does, it may not involve the skills that you are trying to market. That said, people in all fields may be prone to misapply their favorite solutions to inappropriate problems.

Computer scientists should frequently seek out and have conversations with people in other fields who rely on computation, in order to understand their perspectives and needs. This will greatly broaden your perspective and help you better serve potential customers, possibly by telling them that there are more effective solutions to their problems than you can offer with your limited skill set. It will help you understand what skills you really should be building, which may be different than what you're hearing in the computer scientist echo chamber.

Note also that demand for specific skills, such as a particular programming language, machine learning, GPU programming, etc. will wax and wane over time. Problem-solving skills will always be in demand.

## 1.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does the perspective of computer scientists and engineers often differ from that of scientific researchers? Which perspective is better?

# Chapter 2

# Where do I get the Software?

## 2.1 But I Hate Programming...

There are three ways to get the software you need:

1. Buy it.
2. Download it.
3. Write it.

Most researchers today don't need to do much programming beyond some scripting to automate analyses. There's a vast number of both commercial and free software applications available to handle most of the computational needs of researchers, and more being developed all the time.

The need for researchers who can write basic scripts to automate processing pipelines continues to grow. Existing software may do all or most of what you need, but not always conveniently. You will likely need to run multiple programs in sequence to generate the results you need. This sequence is called a *pipeline* and can often be automated with a simple script. Most computational researchers should set out immediately to learn Unix and scripting, as described in Chapter 3 and Chapter 4, but learning hard-core programming will be a lower priority for most.

In many cases, however, solid programming skills could turn out to be a major advantage in the race for research grants.

Unfortunately, most scientific software is very low quality. While there are many well-organized projects around, much of the software is developed as someone's thesis and then abandoned after they graduate. The ability to improve or replace low-quality existing software can remove major barriers to your research.

Also, research by definition involves doing things that have never been done before, and this may require writing new software to perform novel analyses.

Hiring someone else to do the programming is not feasible for most researchers. Experienced scientific programmers are very rare, and likely have higher salaries than you do, so you probably can't afford one even if you can find one. You might find a student to work with you on the cheap or free (for credit), but most likely they'll leave you with badly written, unmaintainable code that the next programmer won't be able to work with.

The only sustainable solution for most researchers who need code written is to do it themselves. The question, then, is which of the dozens of popular programming languages should you learn? This topic is covered in detail in Part III.

For now, suffice it to say that you should become adept at Unix shell scripts, one purely compiled language such as C, which may run hundreds of times faster than scripting languages, and perhaps another interpreted language such as Perl, Python, or R, whichever is most useful in your field. This topic is discussed in more depth below.

### 2.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the three ways to obtain research software?
2. What kind of programming do most researchers need to learn? Why?
3. Why is it a good idea for researchers to learn how to program beyond simple scripting?
4. What is the benefit of learning a compiled language?

## 2.2 Buy It

Commercial software packages are generally a good option for complex engineering computations such as fluid dynamics and finite element analysis. Such software tends to be very costly to develop and therefore exists mainly for needs of wealthy industries such as automotive, aerospace, pharmaceuticals, etc.

Commercial software is generally only available for a limited number of platforms, usually Windows, Mac, and enterprise Linux distributions such as RHEL and SUSE. Furthermore, most commercial software is limited to specific versions of the supported operating systems. For example, some may not run on the latest version of Windows while others will *only* run on the latest version of Windows. This can be a nuisance for those who use multiple commercial applications, which may not be available for the same platforms.

Most commercial software also requires managing licenses that typically limit use to a single computer or require managing a license server, which most IT professionals agree is worse than a root canal. License servers must remain available nearly 24/7, so routine maintenance has to be scheduled at times when they would rather be sleeping or on vacation. License management requires a significant amount of expertise and effort, which should be considered as part of the total cost of ownership (TCO) of the software.

All that said, where there's a market for commercial software, the software often offers powerful capabilities not found elsewhere.

Many computer users fear open source software due to the lack of documentation and direct support from the vendor. These fears are largely unfounded, however. Support for commonly used open source software is usually provided by the user community in the form of online forums and email lists, which are open to everyone, and easily searchable. For all but the most esoteric issues, answers to most of your questions are usually already posted on the Internet and easily found with a simple web search.

It's true that nobody is obligated to help you with free software, but in reality, even in a community where many of the forum participants are rude and arrogant, there are almost always a few people ready and willing to help. Those who do it well will politely point you to existing answers to your question, so you can learn to find your own answers in the future. Even the crabby, rude responses are often helpful, though, and you will learn to be grateful and understanding once you get over the blow to your ego.

Contrary to common expectations, commercial software support does not guarantee answers to your questions either. Access to documentation is often restricted to registered customers who must log into a website to view or search it. Hence, a simple web search may not turn up any answers, because the search engines don't have access to the documentation or discussions. Phone support often involves automated menu systems, spending time on hold, and difficulty finding a support person who can answer the question. The process of finding answers to your questions about commercial software can often take a lot longer than for open source.

When determining whether to purchase a commercial software product, it's best to simply decide whether the features are worth the added purchase cost and effort associated with license management. It may be that the commercial software offers capabilities or performance that are not currently available in any open source equivalent. If this will greatly improve your productivity, then it may justify the costs.

### 2.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. List four disadvantages of commercial software vs free open source software (FOSS).
2. For whom is commercial software generally a good option?

## 2.3 Download It

Fortunately for the vast population of underfunded researchers in most fields, there is a huge and growing collection of open source software available for research.

*Open source software* is software for which the *source code* is freely available. Source code (a collective noun, like "milk" and "honey") is the program in a human-readable language such as C, C++, Fortran, MATLAB, Python, R, etc. It must be *compiled* (translated to *machine language* by a program called a compiler) or *interpreted* by a program called an interpreter, in order to run on a computer. Compiled and interpreted languages are discussed in Section 13.5.

The quality of open source software varies from almost unusable, to better than commercial alternatives. The only way to determine whether open source software will serve your needs is by exploring the available options. Things can change rapidly as well, so what you learned about software options a year ago may no longer apply.

The main advantages of open source over commercial software are:

1. It's usually free.
2. There are no licenses to manage.
3. It will usually run on whatever hardware and operating system you prefer.
4. Installation is trivial if done properly.

On the whole, open source software has come of age. It is now possible for most computer users to do all of their everyday work using exclusively open source operating systems such as BSD, Illumos and Linux and open source applications such as Firefox and LibreOffice.

### 2.3.1 How to Shoot Yourself in the Foot with Open Source Software

Many people fear open source software because they assume it is hard to install and learn. Installation of open source software is actually far easier than commercial software installations when done properly, using a *package manager* such as Debian packages, FreeBSD ports, MacPorts, or Pkgsrc. Package managers became popularized during the 1990s as open source software availability exploded along with computer speed and storage.

Fear of open source software installations usually arises from a lack of awareness of package managers and subsequent unnecessary attempts to perform difficult and poorly-documented "caveman installs", where software is manually downloaded, patched, built, and installed. Unfortunately, many people still perform caveman installs, mostly because they don't know any better. I miss the 1980s too, but nobody should be installing software this way in the 21st century.

Example 2.1 describes a typical caveman install for the R statistics package. Note that this example is relatively simple and well-documented compared to many.

**Example 2.1** A Typical Caveman Install

2.1 Simple compilation
First review the essential and useful tools and libraries in Essential and useful other programs under a Unix-alike, and install those you want or need. Ensure that the environment variable TMPDIR is either unset (and /tmp exists and can be written in and scripts can be executed from) or points to a valid temporary directory (one from which execution of scripts is allowed).
Choose a directory to install the R tree (R is not just a binary, but has additional data sets, help files, font metrics etc). Let us call this place R_HOME. Untar the source code. This should create directories src, doc, and several more under a top-level directory: change to that top-level directory (At this point North American readers should consult Setting paper size.)
Issue the following commands:

```
                ./configure
                make
```

(See Using make if your make is not called 'make'.)
Users of Debian-based 64-bit systems may need

```
                ./configure LIBnn=lib
                make
```

Then check the built system works correctly by

```
                make check
```

Failures are not necessarily problems as they might be caused by missing functionality, but you should look carefully at any reported discrepancies. (Some non-fatal errors are expected in locales that do not support Latin-1, in particular in true C locales and non-UTF-8 non-Western-European locales.) A failure in tests/ok-errors. R may indicate inadequate resource limits (see Running R). More comprehensive testing can be done by

```
                make check-devel
```

or

```
                make check-all
```

See file tests/README. If the configure and make commands execute successfully, a shell-script front-end called R will be created and copied to R_HOME/bin. You can link or copy this script to a place where users can invoke it, for example to /usr/local/bin/R. You could also copy the man page R.1 to a place where your man reader finds it, such as /usr/local/man/man1.
If you want to install the complete R tree to, e.g., /usr/local/lib/R, see installation. Note: you do not need to install R: you can run it from where it was built. You do not necessarily have to build R in the top-level source directory (say, TOP_SRCDIR).
To build in BUILDDIR, run cd BUILDDIR TOP_SRCDIR/configure make and so on, as described further below. This has the advantage of always keeping your source tree clean and is particularly recommended when you work with a version of R from Subversion. (You may need GNU make to allow this, and you will need no spaces in the path to the build directory.)
Now rehash if necessary, type R, and read the R manuals and the R FAQ (files FAQ or doc/manual/R-FAQ.html, or http://CRAN.R-project.org/doc/FAQ/R-FAQ.html which always has the version for the latest release of R).

Before doing the above, however, one must also install dozens of other prerequisite packages, following a similar process for each one. This would include a compiler suite, GNU configure, possibly a make utility, and many math libraries on which R depends.

If you can follow the instructions and all goes well, you may be done with all this in a day or two. More likely, you will struggle for weeks and ultimately give up. If you're not using the exact same version of the exact same operating system as the developers, instructions like these are unlikely to work. Since the developers of the software and all its prerequisites likely use a variety of operating systems, it's very unlikely that you'll get through any installation without running into problems that you're probably not qualified to solve.

### 2.3.2 How Not to Shoot Yourself in the Foot with Open Source Software

Lucky for you, there are thousands of nerds like me creating ports and packages of popular software. As a result, all of the pain described above can be avoided by simply choosing an operating system with a good package manager and learning how to use it. The current state of the most popular package managers can be found at https://repology.org/.

FreeBSD ports, for example, makes it possible to install any one of over 30,000 software packages over the Internet, usually in a matter of seconds. Instead of following the instructions above from the R developers, a FreeBSD user would simply run:

```
            pkg install R
```

This single command will automatically download and install R and all the necessary prerequisite packages required to run it.

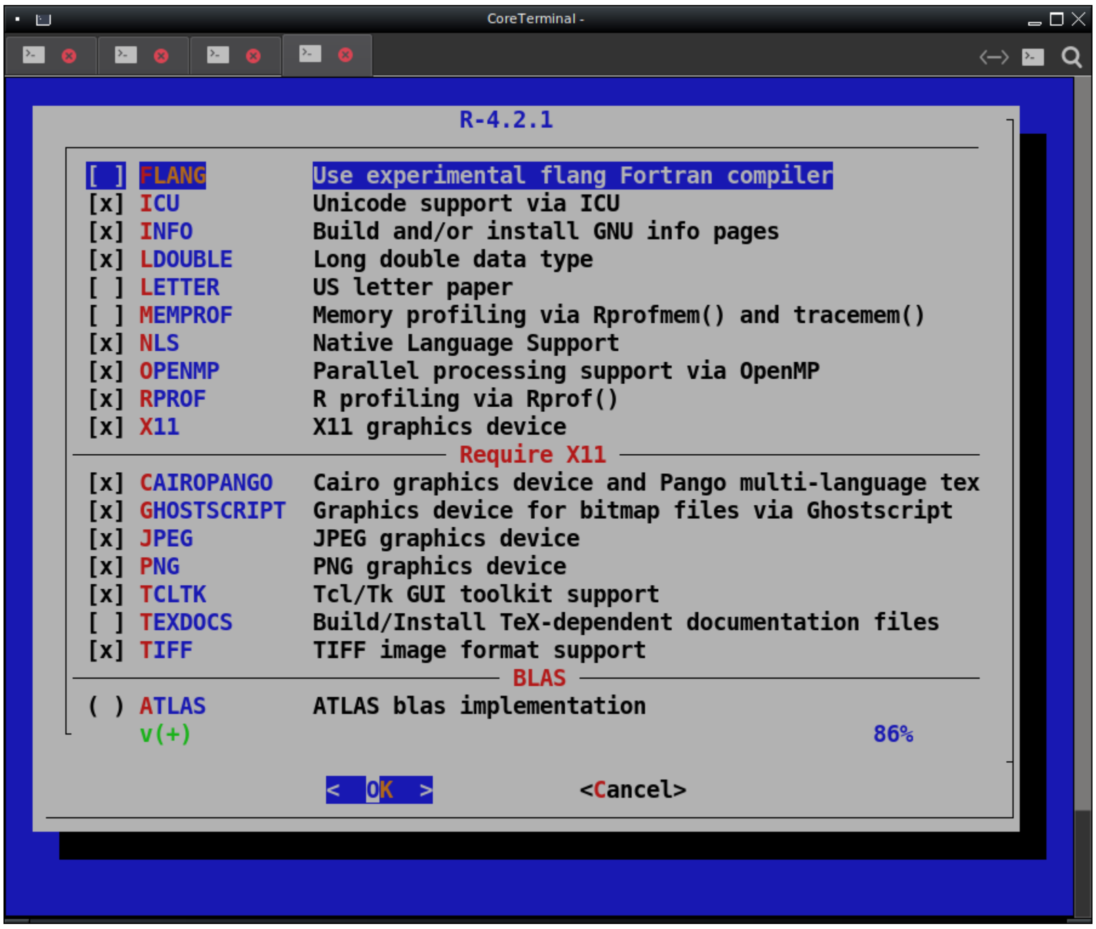
The pkg command installs a "binary" (precompiled) package built to be compatible with most common CPUs. Binary packages are built to utilize only CPU features present on most typical systems. For example, an AMD Epyc processor has features not present in an Intel Core i5 that might make a given program significantly faster. However, binary packages won't use these features, because the software would not run on an i5 processor.

With FreeBSD ports, we can just as easily build and install the R package optimized for the local CPU type. It also allows us to build the package with non-default features and options. The command below will build R, instructing the compiler to use all CPU features available on the computer doing the build.

```
            cd /usr/ports/math/R
            env CFLAGS=-march=native make install
```

Installations of this type take longer to complete, typically minutes to hours, but they require no more effort on your part than installing with the pkg command.

FreeBSD ports also provides a menu for selecting build options, as shown in Figure 2.1. Providing this kind of flexibility via binary packages would mean a separate binary package for every possible combination of options, which is not very practical.



```
                           R-4.2.1
 [ ] FLANG        Use experimental flang Fortran compiler
 [x] ICU          Unicode support via ICU
 [x] INFO         Build and/or install GNU info pages
 [x] LDOUBLE      Long double data type
 [ ] LETTER       US letter paper
 [ ] MEMPROF      Memory profiling via Rprofmem() and tracemem()
 [x] NLS          Native Language Support
 [x] OPENMP       Parallel processing support via OpenMP
 [x] RPROF        R profiling via Rprof()
 [x] X11          X11 graphics device
──────────────────────────── Require X11 ─────────────────────────
 [x] CAIROPANGO   Cairo graphics device and Pango multi-language tex
 [x] GHOSTSCRIPT  Graphics device for bitmap files via Ghostscript
 [x] JPEG         JPEG graphics device
 [x] PNG          PNG graphics device
 [x] TCLTK        Tcl/Tk GUI toolkit support
 [ ] TEXDOCS      Build/Install TeX-dependent documentation files
 [x] TIFF         TIFF image format support
─────────────────────────────── BLAS ─────────────────────────────
 ( ) ATLAS        ATLAS blas implementation
     v(+)                                                   86%

                  <  OK  >            <Cancel>
```

Figure 2.1: FreeBSD Ports Build Options

We can similarly install R via a Debian package on Debian-based Linux distributions (e.g. Debian or Ubuntu) as follows:

```
            apt install R
```

Unfortunately, the Debian package system does not provide an easy way for the average user to build an optimized or customized version from source. Some package managers, such as FreeBSD ports, Gentoo's Portage, MacPorts, and pkgsrc are designed to support conveniently building from source. Others such as Debian packages, RPM, and Conda, are only designed for installing prebuilt binary packages.

Fortunately, Linux users can use the pkgsrc package manager in addition to native package managers such as apt. Pkgsrc is designed to work on any Unix-like platform and can exist alongside other package managers (with some care).

More detail on various package managers and how to use them can be found in Chapter 39. In summary, here's a brief comparison of caveman installs and package managers:

- Caveman installs are very difficult and require extensive knowledge of software development tools. Package managers make installing software trivial.
- Upgrading involves the same nightmarish process as the initial installation. In contrast, upgrading R and all other packages installed via FreeBSD packages is a matter of typing **pkg upgrade**. To upgrade all your Debian packages, simply run **apt update && apt upgrade**.
- Some of the software that the caveman install depends on may come from package managers such as FreeBSD ports or Debian packages. Upgrading or removing these packages may break the caveman install. R will suddenly stop working and it may be difficult to fix. Package managers, in contrast, make sure that all of the packages installed are compatible.
- Uninstalling a caveman install requires knowing where all the files are and removing them manually. Using FreeBSD ports, you would simply run **pkg remove R**. Using Debian packages, **apt remove R**.
- Caveman installs might overwrite files installed by other software. Package managers have safeguards that detect conflicts and prevent this from happening. To get around a conflict with FreeBSD ports, we can install from source to a different installation prefix.

### 2.3.3 What if There is No Package?

If there is no package for your software in the package manager you are using, there are likely better solutions than doing caveman installs for all your software.

- Look into portable package managers such as Conda, pip, or pkgsrc. These can coexist with the native package manager for your system.
- Switch to a different operating system. This might sound radical, but it's actually much easier and safer than a lifetime of caveman installs.
- Learn to create your own packages. This will require an investment of time, but by becoming a package maintainer, you break your dependence on others for managing the software you need. You will forever have the power to cleanly install, remove, and upgrade the software you need.

### 2.3.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are three advantages of FOSS (free open source software)?
2. What can the average computer user do with FOSS nowadays?
3. What is a caveman installation and when should one be performed?

4. What is a package manager?
5. What is an advantage of source-based package managers such as FreeBSD ports, Gentoo Portage, MacPorts, and pkgsrc, over binary-only package managers such as Debian packages and Conda?
6. What are some of the problems that package managers solve when compared with caveman installations?
7. What can you do besides resort to caveman installations if your package manager doesn't have a package for your software?

## 2.4 Containers

Containers are a powerful tool and valuable addition to available software management methods. A container is an isolated environment in which software runs, separated from software running on the host system and software running in other containers. As such, containers are a highly valuable tool for sharing system resources in a secure manner. For example, we can run many web servers on one machine, all completely isolated from each other, so that if any one of them gets hacked, the others and the host system itself remain safe.

Containers have become a trendy solution looking for problems and as such have found their way into research computing. There are valid use cases for containers in research. Unfortunately, though, they have become popular as an alternative to quality software development and build systems. Rather than writing portable software that works with mainstream libraries and is easy to build and install alongside other applications, many developers have recently chosen to containerize their software so it can continue to use outdated libraries (often with known bugs and security holes). In other words, containers are often used to sweep problems under the rug rather than solve them. They essentially become garbage cans full of outdated and low-quality software.

The containerization fad has faded somewhat in recent years as people have begun to see the down side of added overhead and removing the motivation to fix problems and keep software up-to-date. You may find that they remain the only viable option to installing certain software, however. Adding such software to package collections is often difficult, since the developers are not always cooperative about accepting patches to bring it up-to-date with modern libraries, etc. More on this in Chapter 39.

### 2.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Are containers "good"?

## 2.5 Finding Research Software

The number of commercial and free scientific software packages is too vast and too rapidly growing to be listed in any book. The best way to find out about software packages is by searching the WEB for strings such as "finite element software" or "statistical software".

Talking to colleagues can also be helpful, but keep in mind that they are most likely only knowledgeable about one or a few packages that they have been using, and may not even be aware of alternatives, especially newer ones. Choosing software solely on the advice of others is unwise.

Your WEB search will likely lead you to the many Wikipedia articles dedicated to providing an overview of software categories, such as List of finite element software packages, List of software for molecular mechanics modeling, and List of statistical packages.

Some software lists are embedded in other articles, such as Data mining.

A good way to get a quick overview of what's available as established open source projects is looking at the listing of packages available in one of the top-tier package managers, such as Debian packages, FreeBSD ports, Gentoo Portage, MacPorts, or Pkgsrc.

### 2.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Is it a good idea to trust the advice of a colleague about what software to use? Why or why not?
2. How can you be certain that you are using the best available software for your research?

## 2.6 Write It

Writing software is time consuming (i.e. expensive), although not nearly as hard as most people make it for themselves. Those who have the programming skills and esoteric software needs may choose to write their own software.

For these people, choosing the right operating system and the right programming language are critical. All software should be written to be *portable* (to run on any operating system and hardware) and computational software must be *performant* (run as fast and efficiently as possible). Programs written in a compiled language will run on the order of 100 times faster than the same program in an interpreted language. See Chapter 3 for more details about operating systems. Section 13.5 discusses language performance in detail.

Needs may dictate which compiled language you use, but if you have a choice, start with C. It's much simpler and more portable than C++ or Fortran. You can learn the language quickly and then focus on improving the quality of your code rather than getting bogged down in learning more language features. Mastering C++ is a career in and of itself. More on this in Part III.

Interactive software that does minimal computation and is mainly an interface for visualizing data may not need to offer high performance. In this case, the programming language is chosen for convenience rather than speed. Interpreted languages such as Python and Matlab are far slower than compiled languages, but offer convenient plotting libraries such as Python's Matplotlib that make it easy to create beautiful plots and graphs of your data.

Figure 2.2 shows how gene neighborhoods can be visualized using Matplotlib to understand the changes that have occurred over the course of evolution. This is part of a software suite called Microsynteny Tools, which includes several computational programs written in C for optimal performance, and visualization tools written in Python to leverage the convenience and power of Matplotlib.

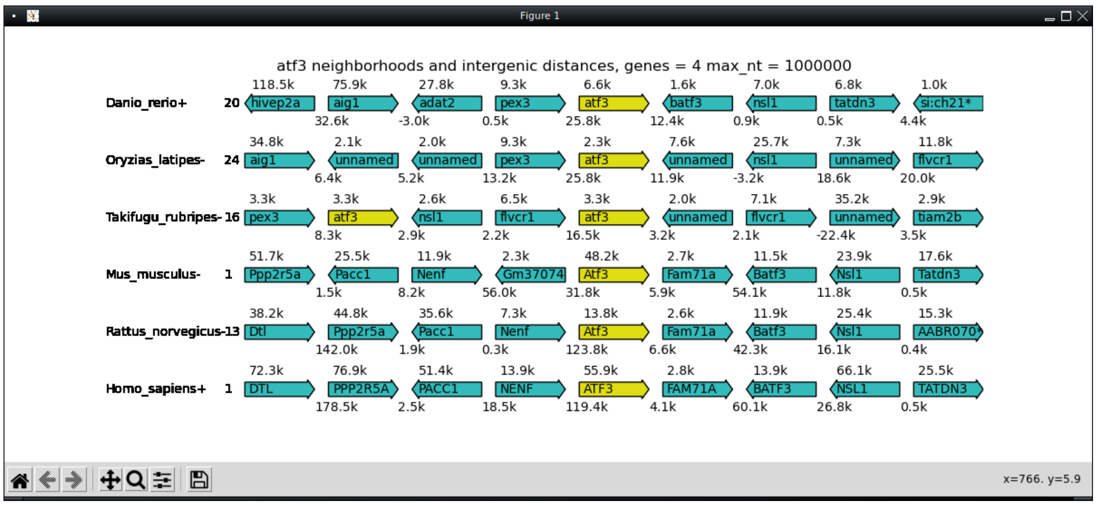


Figure 1
atf3 neighborhoods and intergenic distances, genes = 4 max_nt = 1000000
118.5k 75.9k 27.8k 9.3k 6.6k 1.6k 7.0k 6.8k 1.0k
Danio_rerio+ 20 hivep2a aig1 adat2 pex3 atf3 batf3 nsl1 tatdn3 si:ch21*
32.6k -3.0k 0.5k 25.8k 12.4k 0.9k 0.5k 4.4k
34.8k 2.1k 2.0k 9.3k 2.3k 7.6k 25.7k 7.3k 11.8k
Oryzias_latipes- 24 aig1 unnamed unnamed pex3 atf3 unnamed nsl1 unnamed flvcr1
6.4k 5.2k 13.2k 25.8k 11.9k -3.2k 18.6k 20.0k
3.3k 3.3k 2.6k 6.5k 3.3k 2.0k 7.1k 35.2k 2.9k
Takifugu_rubripes- 16 pex3 atf3 nsl1 flvcr1 atf3 unnamed flvcr1 unnamed tiam2b
8.3k 2.9k 2.2k 16.5k 3.2k 2.1k -22.4k 3.5k
51.7k 25.5k 11.9k 2.3k 48.2k 2.7k 11.5k 23.9k 17.6k
Mus_musculus- 1 Ppp2r5a Pacc1 Nenf Gm37074 Atf3 Fam71a Batf3 Nsl1 Tatdn3
1.5k 8.2k 56.0k 31.8k 5.9k 54.1k 11.8k 0.5k
38.2k 44.8k 35.6k 7.3k 13.8k 2.6k 11.9k 25.4k 15.3k
Rattus_norvegicus-13 Dtl Ppp2r5a Pacc1 Nenf Atf3 Fam71a Batf3 Nsl1 AABR070*
142.0k 1.9k 0.3k 123.8k 6.6k 42.3k 16.1k 0.4k
72.3k 76.9k 51.4k 13.9k 55.9k 2.8k 13.9k 66.1k 25.5k
Homo_sapiens+ 1 DTL PPP2R5A PACC1 NENF ATF3 FAM71A BATF3 NSL1 TATDN3
178.5k 2.5k 18.5k 119.4k 4.1k 60.1k 26.8k 0.5k
x=766. y=5.9

Figure 2.2: Visualizing Gene Neighborhoods with Matplotlib

The main goals when writing a program should always be as follows:

- Write as little of it as possible. Even if you cannot find a program that does exactly what you need, there are probably programs and libraries around that do *most* of what you need.

- Portability: The code should run on any operating system and and hardware.
- Performance: The program should minimize resource use, including CPU, memory, disk, network bandwidth, etc.
- Reliability: The program should produce correct output and never crash.
- Maintainability: The code should be clean, concise, and easy to read.
- User-friendliness: The program should be easy to use and produce meaningful error messages.

Your time will be much better spent finding established and well-tested software to incorporate into your programs, rather than writing everything yourself. For example, if your program involves typical matrix operations, there are many highly-efficient math libraries available that your program can use, such as BLAS, LAPACK, Eigen, Arpack, and METIS, just to name a few. Writing your own matrix multiplication routine would be an enormous waste of your own time and computer time, since the prewritten routines in one of the previously mentioned libraries are probably much faster than anything you would write yourself.

Beware: Bad advice on choosing operating systems and languages abounds in most professions. Many people choose things for irrational reasons, such as finding the appearance pleasing, popularity among friends, etc. A smart selection is based on objective measures such as portability (does it run on any operating system and processor type?), performance, reliability, price, etc.

### 2.6.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the primary goals in writing any software?
2. What is the main advantage of compiled languages over interpreted languages?
3. What is the main advantage of C over other compiled languages for busy researchers?
4. How much should we rely on the advice of others when choosing a language or operating system? Why?

## 2.7 Running Your Software

### 2.7.1 Where do I Run It?

#### Your Own Computers

Many researchers use their own hardware, whether personally owned or company/university owned but for private use. This provides the most flexibility and also the most responsibility and personal time investment in managing the system.

If your organization manages the computer for you, you will spend less time on I.T. and have more time for research. The down side is you cannot always get what you want installed on the machine and if you can, you may have to wait.

The advantages to managing your own hardware and software can be significant, but only if you know how to do it well. Many researchers don't know how to choose and secure an operating system, fail to keep up with security updates, and rely mainly on caveman installs for their scientific software. Most such people would be better off letting I.T. manage their machines.

If you are diligent about keeping your system secure and up-to-date and manage your software via package managers, you may find yourself with a significant edge in the race for research funds.

**Note** Forming a user-group with colleagues in the field can help everyone learn to manage their systems more effectively and efficiently.

#### College Computer Labs

Most colleges and universities maintain computer labs with software to serve the needs of their students. Check with your instructors or department office to find out what's available to you.

#### College Clusters and Grids

Some campuses may also have clusters and grids available for parallel computing. If you need to run large simulations, parameter sweeps, or Monte Carlo simulations, it may be possible to run hundreds at a time instead of one at a time on your PC or a stand-alone lab PC.

#### XSEDE, Open Science Grid

The National Science Foundation funds several very large clusters on campuses around the country for general use by researchers on other campuses.

Use of these resources is free for academic researchers. Small allocations of computing time are easily obtained, while larger allocations require a more extensive proposal.

#### Commercial Services

A number of commercial services are also available for those who have the ability to pay as they go.

Amazon EC2, Google Cloud Platform, Azure, and other cloud computing services allow researchers to create their own custom virtual machines and even virtual clusters. Users pay for CPU time used.

Users of these services can quickly configure virtual machines with a wide variety of configurations such as the number of CPUs, the type of CPU, the amount of RAM, and the amount and type of storage. The cost is typically a fraction of a penny per CPU-hour.

The problem for academic researchers is that funding is usually fixed by research grants, so the unpredictable pay-as-you-go model can be problematic. There may also be significant bureaucracy involved in paying the fees through department channels.

### 2.7.2 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the pros and cons of managing your own computer(s) for research vs using computers managed by your organization?

# Chapter 3

# Using Unix

**Before You Begin**
If you think the word "Unix" refers to Sumerian servants specially "trained" to guard a harem, you've come to the right place. This chapter is designed as a tutorial for users with little or no Unix experience.
If you are following this guide as part of an ungraded workshop, please feel free to work together on the exercises in this text. It would be very helpful if experienced users could assist less experienced users during the "practice breaks" in order to keep the class moving forward and avoid leaving anyone behind.

## 3.1 KISS: Keep It Simple, Stupid

Most people make most things far more complicated than they need to be. Engineers and scientists, especially so.

**Aside**
To the engineer, all matter in the universe can be placed into one of two categories:

1. Things that need to be fixed
2. Things that will need to be fixed after I've had a few minutes to play with them

Engineers like to solve problems. If there are no problems available, they will create their own problems. Normal people don't understand this concept; they believe that if it ain't broke, don't fix it. Engineers believe that if it ain't broke, it doesn't have enough features yet.
No engineer can look at a television remote control without wondering what it would take to turn it into a stun gun. No engineer can take a shower without wondering whether some sort of Teflon coating would make showering unnecessary. To the engineer, the world is a toy box full of sub-optimized and feature-poor toys.
-- The Engineer Identification Test (Anonymous)

Avoid people who tend to look for the most "sophisticated" solution to a problem. Those who look for the simplest solution are more productive, more rational, and more fun to have a beer with. For more amusement on the subject, look up the story of the king's toaster.

Simplicity is the ultimate sophistication. We achieve more when we make things simple for ourselves. We achieve less when we make things complicated. Most people choose the latter. Complexity is the product of carelessness or ego. Simplicity is the product of a wisdom and clarity of thought.

The original Unix designers were an example of wisdom and clarity. Unix is designed to be as simple and elegant as possible. Some things may not seem intuitive at first, but this is probably because the first idea you came up with is not as elegant as the Unix way. The Unix developers had the wisdom to constantly look for simpler ways to implement solutions instead going with what seemed intuitive at first glance. Learning the Unix way will therefore make you a wiser and happier computer user. I speak from experience.

Unix is not hard to learn. You may have gotten the impression that it's a complicated system meant for geniuses while listening to geniuses talk about it. Don't let them fool you, though. The genius ego compels every genius to make things sound really hard, so you'll think they're smarter than you.

Another challenge with learning anything these days is filtering out the noise on the Internet. Most tutorials on any given subject are incomplete and contain misinformation or bad advice. As a result, new users are often led in the wrong direction and hit a dead end before long. One of the goals of this guide is to show a simple, sustainable, portable, and expandable approach to using Unix systems. This will reduce your learning curve by an order of magnitude.

Most researchers don't know enough about Unix. As a result, their productivity suffers dramatically. Unix has grown immensely since it was created, but the reality is, you don't need to know a lot in order to use Unix effectively. You can become more sophisticated over time if you want, but most Unix users don't really need to. It may be better to stick to the KISS principal (Keep It Simple, Stupid) and focus on learning to use the basic tools *well* rather than learning a huge collection of tools and using them poorly. It's quality vs quantity. Knowledge is not wisdom. Wisdom is knowing how to apply it effectively.

**Aside** Einstein was once asked how many feet are in a mile. His reply: "I don't know. Why should I fill my brain with facts I can find in two minutes in any standard reference book?"

Many martial arts students like to collect "forms" (choreographed sequences of moves), for the sake of bragging rights. Knowing more forms does not improve one's Kung Fu, however. The term Kung Fu essentially means "skill". A master is someone who can demonstrate mastery of a few forms, not knowledge of many. This depth of understanding does far more for both self-defense capability and personal development than a shallow knowledge of many forms. Develop your Unix Kung Fu in the same way. Aim to become a master rather than an encyclopedia.

Unix is designed to be as simple as possible and to allow you to work as fast as possible, by staying out of your way. Many other systems will slow you down by requiring you to use cumbersome user interfaces or spend time learning new proprietary methods. As you become a master of Unix, your productivity will be limited only by the speed of the hardware and programs you run.

If something is proving difficult to do under Unix, you're probably going about it the wrong way. There is almost always an easier way, and if there isn't, then you probably shouldn't be trying to do what you're trying to do. If it were a wise thing to do, some Unix developer would have invented an elegant solution by now. Adapt to the wisdom of those who traveled this road before you, and life will become simpler.

### 3.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What's the engineer's motto regarding things that ain't broke?
2. Why do many people believe that Unix is hard to learn?
3. Is it better to accumulate vast amounts of knowledge or to become highly skilled using the fundamentals? Why?
4. What is the core principle of Unix design? Explain.

## 3.2 What is Unix?

### 3.2.1 Aw, man... I Have to Learn Another System?

Well, yeah, but it's the last time, I promise. As you'll see in the sections that follow, once you've learned to use Unix, you'll be able to use your new skills on *virtually any computer*. Over time you'll get better and better at it, and never have to start over from scratch again.

With rare exceptions, if you plan to do computational research, you have two choices:

- Learn to use Unix.
- Rely on the charity of others.

Most scientific software runs only on Unix and very little of it will ever have a graphical or other user interface that allows you to run it without knowing Unix.

The vast majority of high performance computing (HPC) clusters run Unix. You will need basic Unix skills to utilize HPC and HPC clusters generally do not offer a graphical interface. Some HPC administrators attempt to provide for people intent on avoiding Unix, but the results are severely limiting at best.

There have been many attempts to provide access to scientific software via web interfaces, but most of them are abandoned after a short time. People create them with good intentions, but without realizing that they will need to pour effort into maintenance for many years to come. Writing software is like adopting a puppy: It's fun and rewarding, but you need to be committed for the long-term.

In order to be independent in your research computing, you must know how to use Unix in the traditional way. This is the reality of research computing. It's much easier to adapt yourself to reality than to adapt reality to yourself. This chapter will help you become proficient enough to survive and even flourish on your own.

Unix began as the trade name of an operating system developed at AT&T Bell Labs around 1970. It quickly became the model on which most subsequent operating systems have been based. Eventually, "Unix" came into common use to refer to any operating system mimicking the original Unix, much like "Band-Aid" is now used to refer to any adhesive bandage purchased in a drug store.

Over time, formal standards were developed to promote compatibility between the various Unix-like operating systems, and eventually, Unix ceased to be a trade name. Today, the name Unix officially refers to a set of standards to which most operating systems conform.

Look around the room and you will see many standards that make our lives easier. ( Wall outlets, keyboards, USB ports, light bulb sockets, etc. ) All of these standards make it possible to buy interchangeable devices from competing companies. This competition forces the companies to offer better value. They need to offer a lower price and/or better quality than their competition in order to stay in business.

The Unix standards serve the same purpose as all standards; to foster collaboration, give the consumer freedom of choice, reduce unnecessary learning time, and annoy developers who would rather ignore what everyone else is doing and reinvent the wheel at their employer's expense to gratify their own egos. They allow us to become operating system agnostic nomads, readily switching from one Unix system to another as our needs or situations dictate.

In a nutshell, Unix is every operating system you're likely to use except Microsoft Windows. Table 3.1 provides links to many Unix-compatible operating systems. This is not a comprehensive list. Many more Unix-like systems can be found by searching the web.

---

**Note** Apple's macOS has many proprietary extensions, including Apple's own user interface, but is almost fully Unix-compatible and can be used much like any other Unix system by simply choosing not to use the Apple extensions. It is largely based on FreeBSD and other BSD-based components like the Mach kernel.

---

**Note**
When you develop programs for any Unix-compatible operating system, those programs will be *portable*: they can be easily used by people running any other Unix-compatible system, and you can bring them with you when you switch from one Unix-compatible system to another. Most Unix programs can even be run on a Microsoft Windows system with the aid of a *compatibility layer* such as Cygwin (Section 3.4.1).
Unix skills are portable as well. Once you've learned to use one Unix system, you're ready to use any of them. Hence, Unix is the last system you'll ever need to learn!
The time you spend learning or developing programs for non-portable systems make you increasingly dependent on that system and its vendor. By developing code for Unix and developing Unix skills, you maintain your freedom to switch to another operating system whenever you choose.

---

| Name | Type | URL |
|---|---|---|
| AIX (IBM) | Commercial | https://en.wikipedia.org/wiki/IBM_AIX |
| CentOS GNU/Linux | Free | https://en.wikipedia.org/wiki/CentOS |
| Debian GNU/Linux | Free | https://en.wikipedia.org/wiki/Debian |
| DragonFly BSD | Free | https://en.wikipedia.org/wiki/DragonFly_BSD |
| FreeBSD | Free | https://en.wikipedia.org/wiki/FreeBSD |
| GhostBSD | Free | https://en.wikipedia.org/wiki/GhostBSD |
| HP-UX | Commercial | https://en.wikipedia.org/wiki/HP-UX |
| JunOS (Juniper Networks) | Commercial | https://en.wikipedia.org/wiki/Junos |
| Linux Mint | Free | https://en.wikipedia.org/wiki/Linux_Mint |
| MidnightBSD | Free | https://en.wikipedia.org/wiki/MirOS_BSD |
| NetBSD | Free | https://en.wikipedia.org/wiki/NetBSD |
| OpenBSD | Free | https://en.wikipedia.org/wiki/OpenBSD |
| OpenIndiana | Free | https://en.wikipedia.org/wiki/OpenIndiana |
| macOS X and later (Apple Macintosh) | Commercial | https://en.wikipedia.org/wiki/OS_X |
| QNX | Commercial | https://en.wikipedia.org/wiki/QNX |
| Redhat Enterprise Linux | Commercial | https://en.wikipedia.org/wiki/Red_Hat_Enterprise_Linux |
| Slackware Linux | Free | https://en.wikipedia.org/wiki/Slackware |
| SmartOS | Free | https://en.wikipedia.org/wiki/SmartOS |
| Solaris | Commercial | https://en.wikipedia.org/wiki/Solaris_(operating_system) |
| SUSE Enterprise Linux | Commercial | https://en.wikipedia.org/wiki/SUSE_Linux_Enterprise_Desktop |
| Ubuntu Linux (See also Kubuntu, Lubuntu, Xubuntu) | Free | https://en.wikipedia.org/wiki/Ubuntu_(operating_system) |

Table 3.1: Partial List of Unix Operating Systems

Unix systems run on everything from your cell phone to the world's largest supercomputers. Unix is the basis for Apple's iOS, the Android mobile OS, embedded systems such as networking equipment and robotics controllers, most PC operating systems, and many large mainframe systems. Many Unix systems are completely free (as in free beer) and can run tens of thousands of high-quality free software packages. As an extreme example, NetBSD runs on dozens of different CPU architectures, including some hobbyist systems such as Commodore Amigas, 68k-based Macs, etc.

It's a good idea to regularly use more than one Unix system. This will make you aware of how much they all have in common and what the subtle differences are.

### 3.2.2 Operating System or Religion?

**Aside**
Keep the company of those who seek the truth, and run from those who have found it.
-- Vaclav Havel

The more confident someone is in their views, the less they probably know about the subject. As we gain life experience and wisdom, we become less certain about everything and more comfortable with that uncertainty. What looks like confidence is usually a symptom of ignorance of our own ignorance, generally fueled by ego.

If you discuss operating systems at length with most people, you will discover, as the ancient philosopher Socrates did while discussing many topics with "experts", that their views are not based on broad knowledge and objective comparison. Before taking advice from anyone, it's a good idea to find out how much they really know and what role emotion and ego play in their preferences. Ask them to clarify their statements and support them with evidence. This process of questioning has become known as a "Socratic examination", named after the famous Greek philosopher Socrates. Note, however, that if you embarrass the wrong people, it may get you executed, as it did Socrates himself according to legend.

The whole point of the Unix standard, like any other standard, is freedom of choice. However, you won't have any trouble finding evangelists for a particular brand of Unix-compatible operating system on whom this point is lost. "Discussions" about the merits of various Unix implementations often involve arrogant pontification and emotional outbursts, possibly involving some cussing.

If you step back and ask yourself what kind of person gets emotionally attached to a piece of software, you'll realize whose advice you should value and whose you should not. Rational people will keep an open mind and calmly discuss the objective measures of an OS, such as performance, reliability, security, easy of maintenance, specific capabilities, etc. They will also back up their opinions with facts rather than try to bully you into validating their views.

If someone tells you that a particular operating system "isn't worth using", "is way behind the times", or "sucks wads", rather than asking you what you need and objectively discussing alternatives, this is someone whose advice you can safely ignore. They are not interested in helping you. They need you to validate their opinions, because those opinions are not supported by facts.

---

**Aside**
We're all capable of rational thought, but sometimes we only use it to rationalize what we want to believe, despite obvious evidence to the contrary.
"I don't understand why some people wash their bath towels. When I get out of the shower, I'm the cleanest object in my house. In theory, those towels should be getting cleaner every time they touch me. By the way, are towels supposed to bend?"
-- Wally (Dilbert)

---

Evangelists are easy to spot. They will instantly assess your needs without asking you a single question and proceed to explain (often aggressively) why you should be using their favorite operating system or programming language. They invariably have limited or no experience with other alternatives. This is easy to expose with a few simple questions. "How many years of experience to you have with it?" The answer is usually close to 0. "What are the specific advantages and disadvantages?" The response to this will usually be stuttering, silence, or double-talk. Ask them to clarify further and it won't take long to expose their ignorance.

Ultimately, the system that most easily runs your programs to your satisfaction is the best one for you. That could turn out to be BSD, Cygwin, Linux, macOS, OpenIndiana, or any other. Someone who knows what they're doing and truly wants to help you will always begin by asking questions in order to better understand your needs. "What program(s) do you need to run?", "Do they require any special hardware?", "Do you need to run any commercial software, or just open source?", etc. They will then consider multiple alternatives and inform you about the capabilities of each one that might match your needs.

There is another reason besides ego that people often choose inappropriate solutions to a problem; the desire to use what they know instead of being open to learning a better approach.

---

**Aside** When all you have is a hammer, everything looks like a nail.

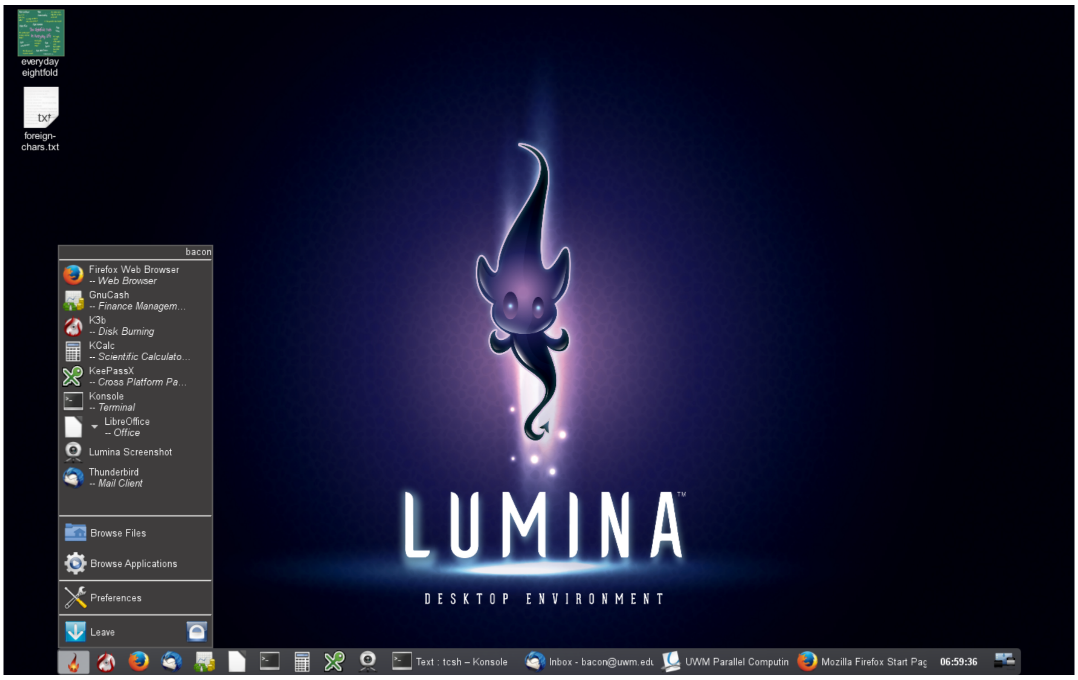
---

I regularly experiment with various Unix variants to evaluate their ease of use, reliability, and resource requirements. This is easy to do using virtual machines (See Chapter 40.) My personal preference for running Unix software (for now, these could change in the distant future) are listed below. All of these systems are somewhat interchangeable with each other and the many other Unix based systems available, so deviating from these recommendations will generally not lead to catastrophe.

More details on choosing a Unix platform are provided in Chapter 37.

- Servers running mostly open source software: FreeBSD.

  FreeBSD is extremely fast, reliable, and secure. It is known as a "set-and-forget" operating system, since it requires very little attention after initial installation and configuration. Software management is very easy with FreeBSD ports, which offers over 30,000 software packages (not counting different builds of the same software). The ports system supports installation via either generic binary packages, or you can just as easily build from source with custom options or optimizations for your specific CPU. With the Linux compatibility module, FreeBSD can directly run most Linux closed-source programs with no performance penalty and a little added effort and resources.



*FreeBSD with the Lumina desktop environment*

- Servers running mainly or commercial applications or CUDA GPU software: Enterprise Linux (AlmaLinux, CentOS, RHEL, Rocky Linux, Scientific Linux, SUSE).

  These systems are designed for better reliability, security, and long-term binary compatibility than bleeding-edge Linux systems. They are the only platforms besides MS Windows and macOS supported by many commercial software vendors. While you may be able to get some commercial engineering software running on Ubuntu or Mint, it is often difficult and the company will not provide support. Packages in the native Yum repository of enterprise Linux are generally outdated, but more recent open source software can be installed using a separate add-on package manager such as pkgsrc.

- An average Joe who wants to browse the web, use a word processor, etc.: Debian, GhostBSD, Ubuntu, or similar open source Unix system with graphical installer and management tools, or Macintosh.

  These systems make it easy to install software packages and system updates, with minimal risk of breakage that Joe would not know how to fix.

*Debian Linux*

- Someone who uses mostly Windows-based software, but needs a basic Unix environment for software development or connecting to other Unix systems: A Windows PC with Cygwin.

  Cygwin is free, entirely open source, and very easy to install in about 10 minutes on most Windows systems. It has some performance bottlenecks, fewer packages than a real Unix system running on the same machine, and a few other limitations, but it's more than adequate for the needs of many typical users. See Section 3.4.1 for details.

### 3.2.3 The Unix Standard API

Programmer time is expensive. Writing a program twice costs twice as much. Unix standards solve this problem.

Unix systems adhere to an *application program interface* (API) standard, which means that programs written for one Unix-based system can be run on any other with little or no modification, even on completely different hardware. For example, programs written for an Intel/AMD-based Linux system will also run an ARM-based Mac, or FreeBSD on an ARM, Power, or RISC-V processor.

An API defines a set of functions (subprograms) used to request services from the operating system, such as opening a file, allocating memory, running another program, etc. These functions are the same on all Unix systems, but some of them are different on Windows and other non-standard systems. For example, to open a file in a C program on any Unix system, one would typically use the fopen() function:

```
FILE *fopen(const char *filename, const char *mode);
```

Microsoft compilers support fopen() as well, but also provide another function for the same purpose that only works on Windows:

```
errno_t fopen_s(FILE** pFile, const char *filename, const char *mode);
```

**Note** Microsoft claims that fopen_s() is more secure, which is debatable. Note however, that even if this is true, the existing fopen() function itself could have been made more secure rather than creating a separate, non-portable function that does the same thing. Non-standard functions like fopen_s() mainly benefit the vendor by making it harder to port software to a competing platform.

Here are a few other standard Unix functions that can be used in programs written in C and most other compiled languages. These functions can be used on any Unix system, regardless of the type of hardware running it. Some of these may also work in Windows, but for others, Windows uses a completely different function to achieve the same goal.

```
            chdir()         // Change current working directory
            execl()         // Load and run another program
            mkdir()         // Create a directory
            unlink()        // Remove a file
            sleep()         // Pause execution of the process
            DisplayWidth()  // Get the width of the graphical screen
```

Because the Unix API is platform-independent, it is also possible to compile and run most Unix programs on Windows with the aid of a *compatibility layer*, software that bridges the difference between two platforms. ( See Section 3.4.1 for details. ) It is not generally possible to compile and run Windows software on Unix, however, because Windows has many features specific to PC hardware.

Since programs written for Unix can be run on almost any computer, including Windows computers, they will never have to be rewritten in order to run somewhere else. Programs written for non-Unix platforms will only run on that platform, and will have to be rewritten (at least partially) in order to run on any other system. This leads to an enormous waste of man-hours that could have gone into creating something new. They may also become obsolete as the proprietary systems for which they were written evolve. For example, most programs written for MS DOS and Windows 3.x are no longer in use today, while programs written for Unix around that same time will still work on modern Unix systems.

### 3.2.4 Shake Out the Bugs

Another advantage of programming on standardized platforms is the ability to easily do more thorough testing. Compiling and running a program on multiple operating systems and with multiple compilers will almost always expose bugs that you were unaware of while running it on the original development system. The same bug will have different effects on different operating systems, with different compilers or interpreters, or with different compile options (e.g. with and without optimization).

For example, an errant array subscript or pointer might cause corruption in a non-critical memory location in some environments, while causing the program to crash in others.

A program may seem to be fine when you compile it with Clang and run it on your Mac, but may not compile, or may crash when compiled with GCC on a Linux machine.

Finding bugs *now* may save you from the stressful situation of tracking them down under time pressure later, with an approaching grant deadline. A bug that was invisible on your Mac for the test cases you've used could also show up on your Mac later, when you run the program with different inputs.

Developing for the Unix API makes it easy to test on various operating systems and with different compilers. There are many free BSD and GNU/Linux systems, as well as free compilers such as Clang and GCC. Most of them can be run in a virtual machine (Chapter 40), so you don't even need another computer for the sake of program testing. Take advantage of this easy opportunity to stay ahead of program bugs, so they don't lead to missed deadlines down the road.

### 3.2.5 The Unix Standard UI

The Unix standards not only make programs portable, they make our knowledge as users portable as well. All Unix systems support the same basic set of commands, which conform to standards so that they behave the same way everywhere. So, if you learn to use FreeBSD, most of that knowledge will directly apply to Linux, macOS, Solaris, etc.

Another part of the original Unix design philosophy was to do everything in the simplest way possible. As you learn Unix, you will likely find some of its features befuddling at first. However, upon closer examination, you will often come to appreciate the elegance of the Unix solution to a difficult problem. If you're observant enough, you'll learn to apply this Zen-like simplicity to your own work, and maybe even your everyday life.

You will also gradually recognize a great deal of consistency between various Unix commands and functions. For example, many Unix commands support a `-v` (verbose) flag to indicate more verbose output, as well as a `-q` (quiet) flag to indicate no unnecessary output. Over time, you will develop an intuitive feel for Unix commands, become adept at correctly guessing how things work, and feel almost God-like at times.

Unix documentation also follows a few standard formats, which users quickly get used to, making it easier to learn new things about commands on any Unix system.

In a nutshell, the time and effort you spend learning any Unix system will make it easy to use any other in the future. You need only learn Unix once, and you'll be proficient with many different implementations such as FreeBSD, Linux, and macOS.

### 3.2.6 Fast, Stable and Secure

Since Unix systems compete directly with each other to win and retain users running the same programs, developers are highly motivated to optimize objective measures of the system such as performance, stability, and security.

Most Unix systems operate near the maximum speed of the hardware on which they run. Unix systems typically respond faster than other systems on the same hardware and run intensive programs in less time. Many Unix systems require far fewer resources than non-Unix systems, leaving more disk and memory for use by your programs.

Unix systems tend to be very reliable and may run for months or even years without being rebooted. I managed a particular FreeBSD HPC cluster for eight years. Except for some problems in the first few months that were traced to a Dell firmware bug, none of the servers in this cluster ever crashed.

Unlike Windows, software installations almost never require a reboot, and even most security updates can be applied without rebooting. Reboots are typically only needed following a kernel update.

Stability is critical for research computing, where computational models may run for weeks or months. Users of non-Unix operating systems often have to choose between killing a process that has been running for weeks and neglecting critical security updates that require a reboot.

Very few viruses or other malware programs exist for Unix systems. This is in part due to the inherently better security of Unix systems and in part due to a strong tradition in the Unix community of discouraging users from engaging in risky practices such as running programs under an administrator account and installing software from pop-ups on the web.

### 3.2.7 Sharing Resources

Your mom probably told you that it's nice to share, but did you know it's also more efficient?

One of the major problems for researchers in computational science is managing their own computers. Most researchers aren't very good at installing operating systems, managing software, apply security updates, etc., nor do they want to be. Unfortunately, they often have to do these things in order to conduct computational research. Computers managed by a tag-team of researchers usually end up full of junk software, out-of-date, full of security issues, and infected with malware.

Since Unix is designed from the ground up to be accessed remotely, Unix creates an opportunity to serve researchers' needs far more cost-effectively than individual computers for each researcher. A single Unix machine on a modern PC can support dozens or even hundreds of users at the same time, depending how demanding their software is.

### 3.2.8 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. After learning Unix, on what operating systems will you be able to use your new skills?
2. What is the major design goal of the Unix standards?

3. What is the alternative to learning Unix for computational scientists? Why?
4. Why does most scientific software lack a convenient graphical or web interface?
5. Is Unix an operating system? Why or why not?
6. What is the advantage of open standards?
7. How many different Unix-compatible operating systems exist? What does this mean for Unix users?
8. Which mainstream operating systems are Unix-compatible and which are not?
9. What types of computer hardware run Unix?
10. How much does Unix cost?
11. Which Unix operating system is the best one?
12. How should we go about choosing a Unix system? What if we make the wrong choice?
13. How do we spot evangelists who are likely to give us irrational advice?
14. What is an API?
15. What is the advantage of the Unix API over the APIs of non-Unix operating systems? What problem does it solve?
16. Can software written for Unix be run on Windows? How?
17. How does the Unix API help us proactively eliminate software bugs?
18. What is a UI? What are three advantages of the Unix UI over the UIs of non-Unix operating systems?
19. Why are Unix-compatible operating systems faster, more stable, and more secure than many non-Unix platforms?
20. How does the inherent remote access capabilities of Unix help researchers?

## 3.3 Unix User Interfaces

A *user interface*, or *UI*, refers to the software that allows a person to interact with the computer. The UI provides the look and feel of the system, and determines how easily and efficiently it can be used. ( Note that ease of use and efficiency are not the same! )

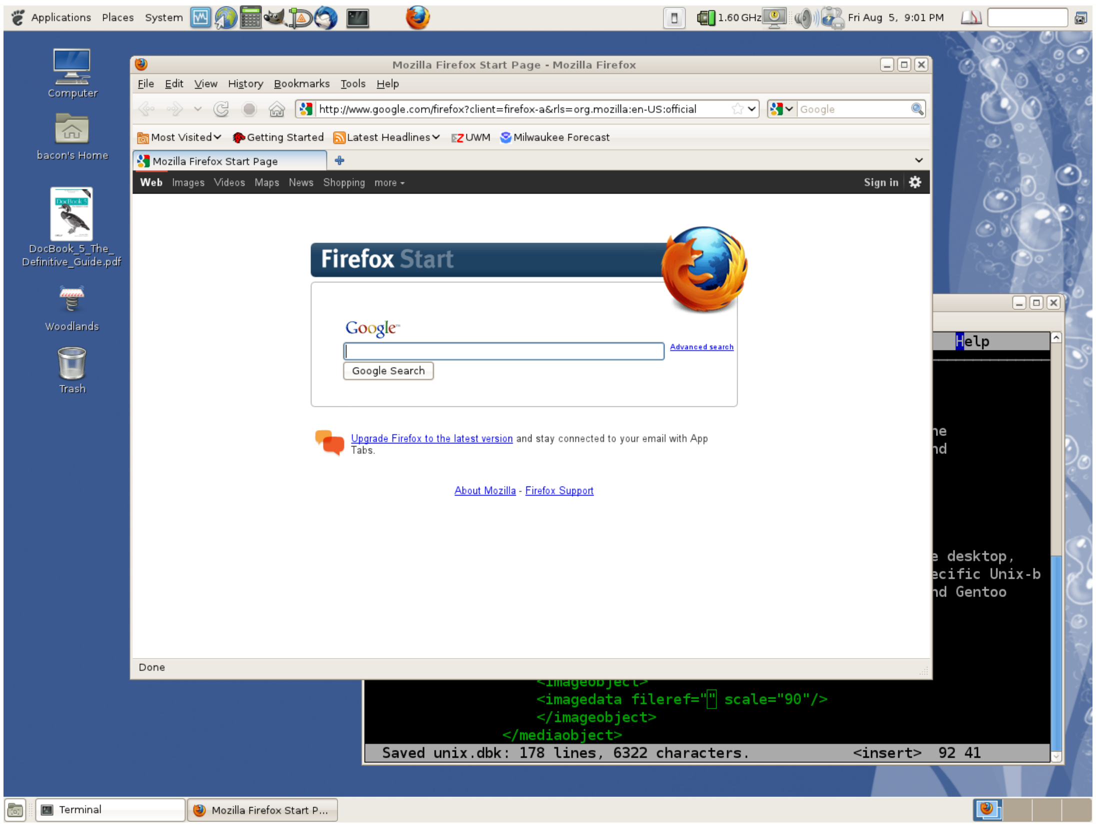
The term "MS Windows" refers to a specific proprietary operating system, and implies all of the features of that system including the API and the UI. When people think of Windows, they think of the Start menu, the Control Panel, etc. Likewise, "Macintosh" refers to a specific product and invokes images of the "Dock" and a menu bar at the top of the screen rather than attached to a window.

The term "Unix", on the other hand, implies an API, but does not imply a specific UI. There are many UIs available for Unix systems. In fact, a computer running Unix can have multiple UIs installed, and each user can choose the one they want when the log in.

### 3.3.1 Graphical User Interfaces (GUIs)

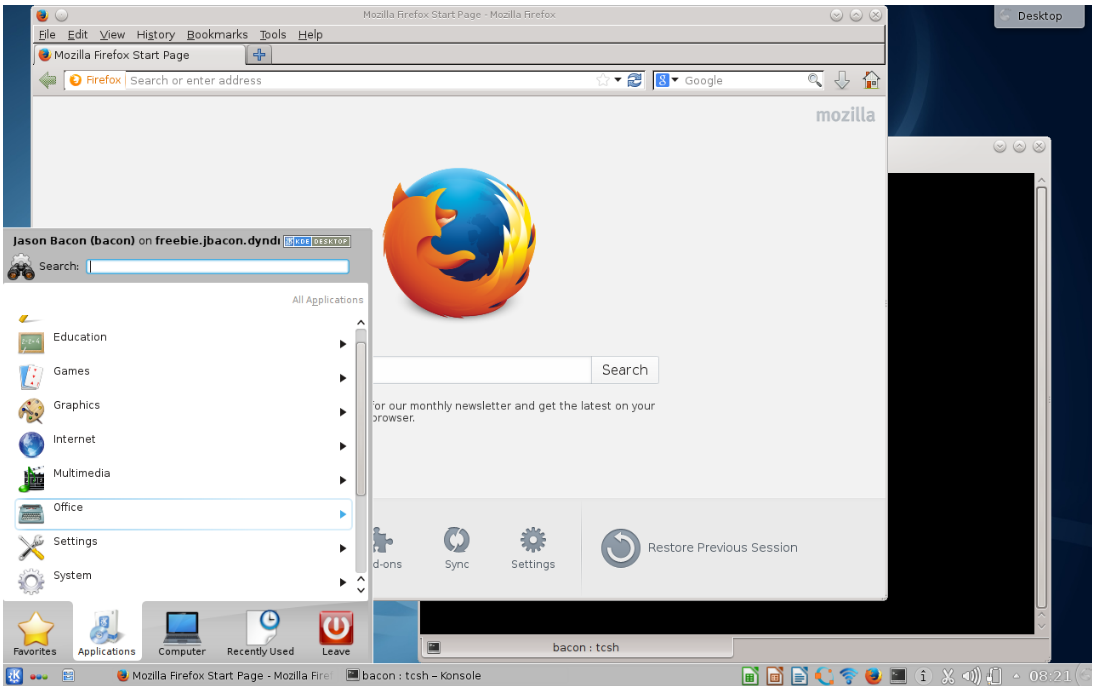
A *Graphical User Interface*, or *GUI* (pronounced goo-ee), is a user interface with a graphical screen, and icons and menus we can select using a mouse or a touch screen.

There are many different GUIs available for Unix. Some of the more popular ones include KDE, Gnome, XFCE, LXDE, OpenBox, CDE, and Java Desktop.

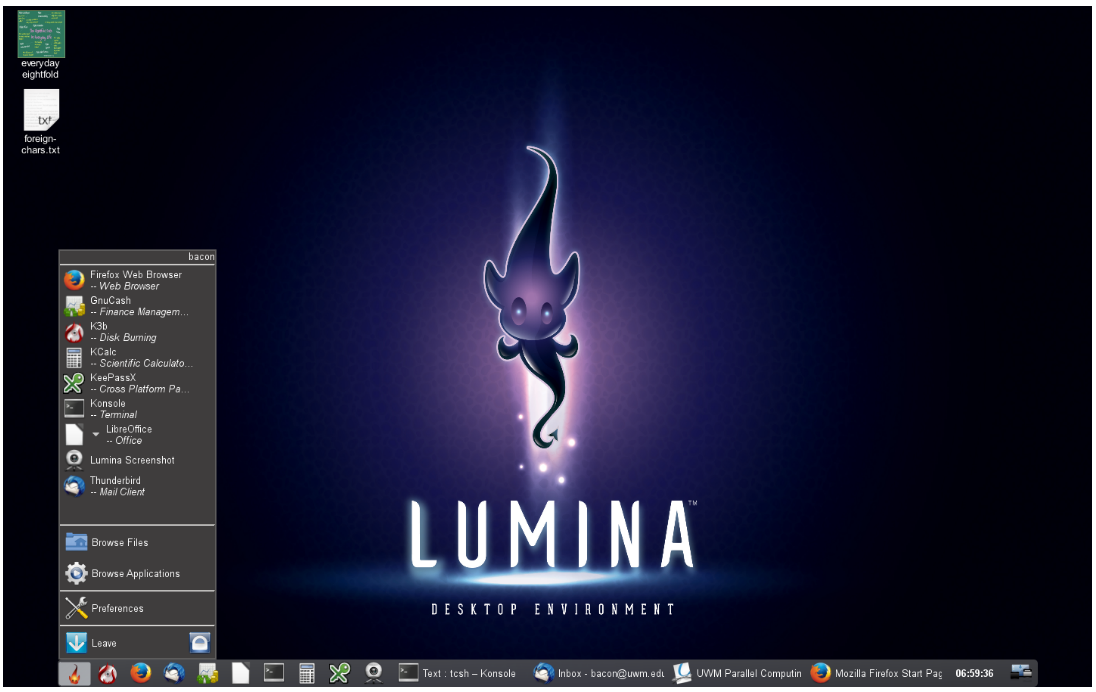

Applications Places System
Fri Aug 5, 9:01 PM
Computer
bacon's Home
DocBook_5_The_Definitive_Guide.pdf
Woodlands
Trash
Mozilla Firefox Start Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.google.com/firefox?client=firefox-a&rls=org.mozilla:en-US:official
Most Visited Getting Started Latest Headlines UWM Milwaukee Forecast
Firefox Start
Google Search
Advanced search
Upgrade Firefox to the latest version and stay connected to your email with App Tabs.
About Mozilla - Firefox Support
Done
Terminal
Mozilla Firefox Start P...

```
        <imageobject>
          <imagedata fileref="" scale="90"/>
        </imageobject>
      </mediaobject>
 Saved unix.dbk: 178 lines, 6322 characters.          <insert>  92 41
```

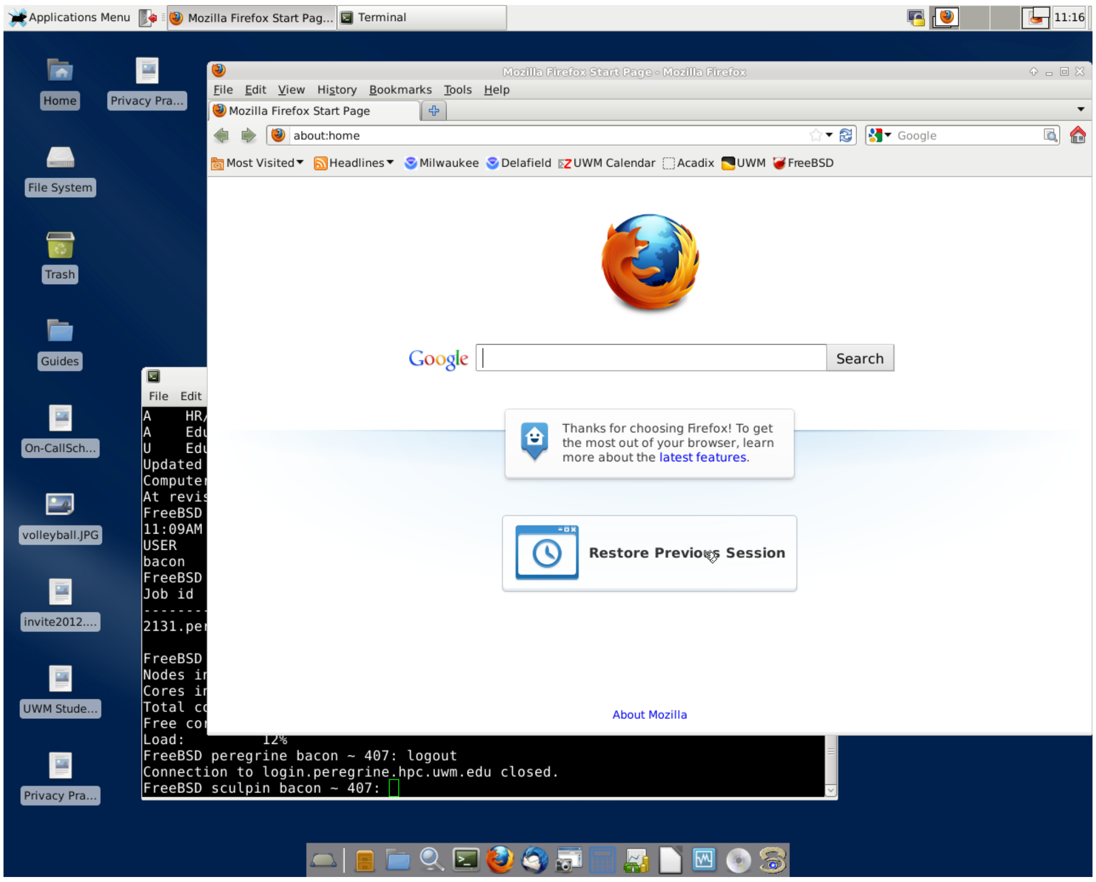
A FreeBSD system running Gnome desktop.


Mozilla Firefox Start Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Search or enter address
mozilla
Desktop
Jason Bacon (bacon) on freebie.jbacon.dyndns
Search:
All Applications
Education
Games
Graphics
Internet
Multimedia
Office
Settings
System
Favorites Applications Computer Recently Used Leave
Search
Sync
Settings
Restore Previous Session
bacon : tcsh
Mozilla Firefox Start Page - Mozilla Firefox
bacon : tcsh – Konsole
08:21

A FreeBSD system running KDE desktop.



*A FreeBSD system running Lumina desktop.*



```
A     HR/
A     Edu
U     Edu
Updated
Compute
At revis
FreeBSD
11:09AM
USER
bacon
FreeBSD
Job id
--------
2131.per

FreeBSD
Nodes in
Cores in
Total co
Free cor
Load:          12%
FreeBSD peregrine bacon ~ 407: logout
Connection to login.peregrine.hpc.uwm.edu closed.
FreeBSD sculpin bacon ~ 407: 
```

*A FreeBSD system running XFCE desktop.*

**Example 3.1** Practice Break

If you have access to a Unix GUI, log into your Unix system via the GUI interface now.

All Unix GUIs are built on top of the X11 networked graphics API. As a result, all Unix systems have the inherent ability to display graphics on other Unix systems over a network. I.e., you can remotely log into another Unix computer over a network and run graphical programs that display output wherever you're sitting.

This is not the same as a *remote desktop* system, which mirrors the console display on a remote system. Unix systems allow multiple users in different locations to run graphical programs independent of each other. In other words, Unix supports multiple independent graphical displays on remote computers.

It is also not the same as a *terminal server*, which opens an entire desktop environment on a remote display.

With Unix and X11, we can have individual applications running on multiple remote computers displayed on the same desktop. Doing so is easy and requires no additional software to be installed.

Most Unix GUIs support multiple *virtual desktops*, also known as *workspaces*. Virtual desktops allow a single monitor to support multiple separate desktop images. It's like having multiple monitors without the expense and clutter. The user can switch between virtual desktops by clicking on a panel of thumbnail images, or in some cases by simply moving the mouse over the edge of the screen.

### 3.3.2 X11 on macOS

macOS is Unix compatible, derived largely from FreeBSD and the Mach kernel project, with components from GNU and other Unix-based projects.

It differs from more traditional Unix systems like BSD and Linux, though, in that it runs Apple's proprietary graphical API and GUI. Native OS X programs don't use the X11 API, but OS X can also run X11-based programs with the XQuartz add-on. See Section 3.19 for instructions on enabling X11 for Mac.

### 3.3.3 Command Line Interfaces (CLIs): Unix Shells

There are two basic types of user interfaces:

- Menu-driven, where choices are displayed on the screen and the user selects one.
- Command-driven, where the user types commands that they have memorized.

A GUI is a type of menu-driven interface, where the menu items may be text or graphical icons. Some menu systems are simply text selected by entering a number on the keyboard.

```
               ___         __           ___       __          _
              /   | __  __/ /_____     /   | ____/ /___ ___  (_)___
             / /| |/ / / / __/ __ \   / /| |/ __  / __ `__ \/ / __ \
            / ___ / /_/ / /_/ /_/ /  / ___ / /_/ / / / / / / / / / /
           /_/  |_\__,_/\__/\____/  /_/  |_\__,_/_/ /_/ /_/_/_/ /_/
           _________________________________________________________

                  Portable Command-line Systems Management
                      https://acadix.biz/auto-admin.php
           _________________________________________________________

This menu system encompasses only a small fraction of the total auto-admin
functionality.  To see what else is available via the command-line, choose
"List available auto-admin scripts" below.

Full documentation is in the works and will be included in a future release.

1.. Update system
2.. User management
3.. Software management
4.. Network management
5.. Power management
```

```
6.. File system actions and settings
7.. Security settings
8.. System settings
9.. Services manager
10.. List available auto-admin scripts
Q.. Quit

Selection?
```

While all modern Unix systems have GUIs, much Unix work is still done via the *command line interface* (*CLI*). A CLI requires the user to type in commands, rather than select them from a menu, much like short-answer questions vs multiple choice.

Menu-driven systems are easier to use if the system has limited functionality and you're new to the system or use it infrequently. However, menus are cumbersome where there is too much functionality to offer in a simple menu. Even simple menu systems can become cumbersome for everyday use.

If a user needs access to dozens or hundreds of features, they cannot all be displayed on the screen at the same time. Hence, it will be necessary to navigate through multiple levels of menus or screens to find the functionality you need. Doing this repeatedly becomes annoying rather quickly. A command line interface, on the other hand, provides instant access to an unlimited number of commands.

An ATM (automatic teller machine) is a good candidate for a menu interface. It has only a few functions and people don't use it every day. An ATM with a command-driven interface would likely be unpopular among banking customers.

You might be surprised to learn that CAD (Computer Aided Design) systems have CLIs. While CAD is inherently graphical in nature, CAD users cannot efficiently access their vast functionality through menus. Most CAD users quickly learn to use the CLI to draw, move, and edit objects via keyboard commands.

Because menu systems slow us down, most support *hot keys*, special key combinations that can be used to access certain features without navigating the menus. Hot keys are often shown in menus alongside the features they activate. For example, Command+q can be used on macOS and Ctrl+q on Windows and most Unix GUIs to terminate many graphical applications, as shown in Figure 3.1.

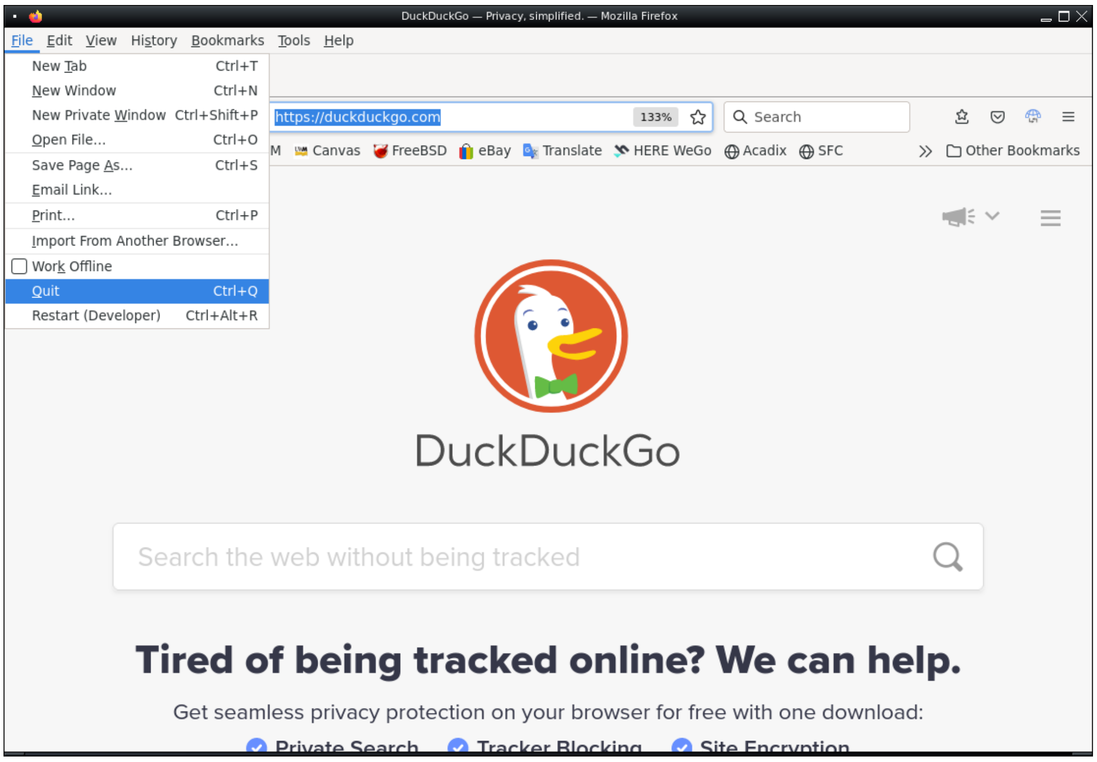

DuckDuckGo — Privacy, simplified. — Mozilla Firefox
File Edit View History Bookmarks Tools Help
New Tab Ctrl+T
New Window Ctrl+N
New Private Window Ctrl+Shift+P
Open File... Ctrl+O
Save Page As... Ctrl+S
Email Link...
Print... Ctrl+P
Import From Another Browser...
Work Offline
Quit Ctrl+Q
Restart (Developer) Ctrl+Alt+R
https://duckduckgo.com
133%
Search
Canvas FreeBSD eBay Translate HERE WeGo Acadix SFC
Other Bookmarks
DuckDuckGo
Search the web without being tracked
Tired of being tracked online? We can help.
Get seamless privacy protection on your browser for free with one download:
Private Search Tracker Blocking Site Encryption

Figure 3.1: Hot Keys

It is also difficult to automate tasks in a menu-driven system. Some systems have this capability, but most do not, and the method of automating is different for each system. Command-driven interfaces are easy to automate by placing commands in a *script*, a simple text file containing a sequence of commands that might otherwise be run directly via the keyboard. Scripting is covered in Chapter 4.

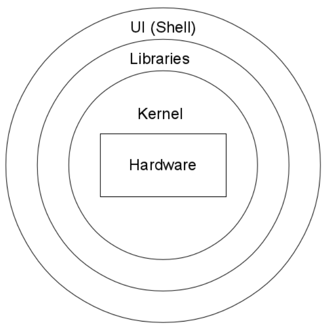
Perhaps the most important drawback of menu-driven systems is non-existence. Programming a menu system, and especially a GUI, requires a lot of grunt-work and testing. As a result, the vast majority of open source software does not and never will have a GUI interface. Open source developers generally don't have the time or programming skills to build and maintain a comprehensive GUI interface.

**Caution**
If you lack command-line skills, you will be limited to using a small fraction of available open source software.
In the tight competition for research grants, those who can use the command-line more often win.

The small investment in learning a command line interface can have a huge payoff, and yet many people try to avoid it. The result is usually an enormous amount of wasted effort dealing with limited and poorly designed custom user interfaces before eventually realizing that things would have been much easier had they learned to use the command line in the first place. It's amazing how much effort people put into avoiding effort...

A *shell* is a program that provides the command line interface. It inputs commands from the user, interprets them, and executes them. ( By "execute", we mean "run", not what happened to Socrates. ) Using a shell, you type a command, press enter, and the command is immediately executed.

The word "shell" comes from the view of Unix as three layers of software wrapped around the hardware:



*A 3-layer Model of Unix*

- The innermost layer, which handles all hardware interaction for Unix programs, is called the *kernel*, named after the core of a seed. The Unix kernel effectively hides the hardware from user programs and provides a standard API. This is what allows Unix programs to run on different kinds of hardware without modification. Application programs never "see" the hardware interface. They only see the kernel interface, which is the same regardless of hardware.
- The middle layer, the libraries, provide a wealth of standard functionality for Unix programmers to utilize. The libraries are like a huge box of Legos that can be used to build all kinds of sophisticated programs. They include basic input/output functions, math functions, character string functions, graphics functions, etc.
- The outermost layer, the CLI, is called a shell.

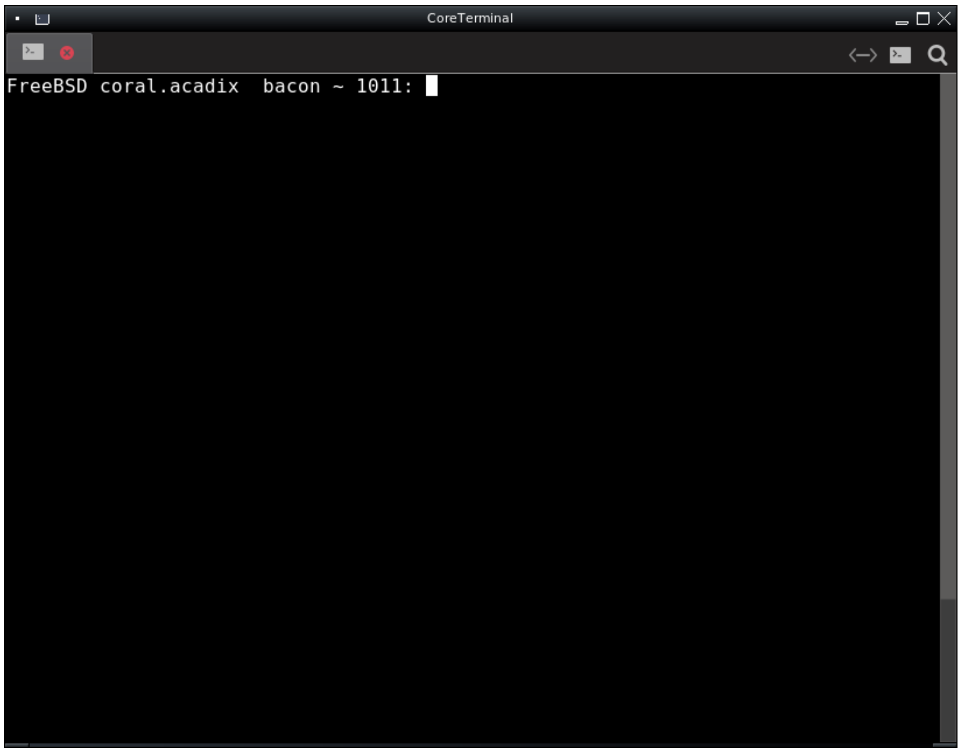
### 3.3.4 Terminals

All that is needed to use a Unix shell is a keyboard and a screen. In the olden days, these were provided by a simple hardware device called a *terminal*, which connected a keyboard and screen to the system through a simple communication cable. These terminals typically did not have a mouse or any graphics capabilities. They usually had a text-only screen of 80 columns by 24 lines, and offered limited capabilities such as moving the cursor, scrolling the screen, and perhaps a limited number of colors, usually 8 or 16.

Digital VT320 terminal

Hardware terminals lost popularity with the advent of cheap personal computers, which can perform the functions of a terminal, as well as running programs of their own. Hardware terminals are still used in some settings that require extreme security, such as banking, where a PC's capabilities would aid in stealing information by saving it to an external device or transmitting it across a network. A dumb terminal ensures that information can be sent to the user on the screen and nowhere else.

Hardware terminals have been largely replaced by *terminal emulators*. A terminal emulator is a simple program that emulates an old style terminal within a window on your desktop.

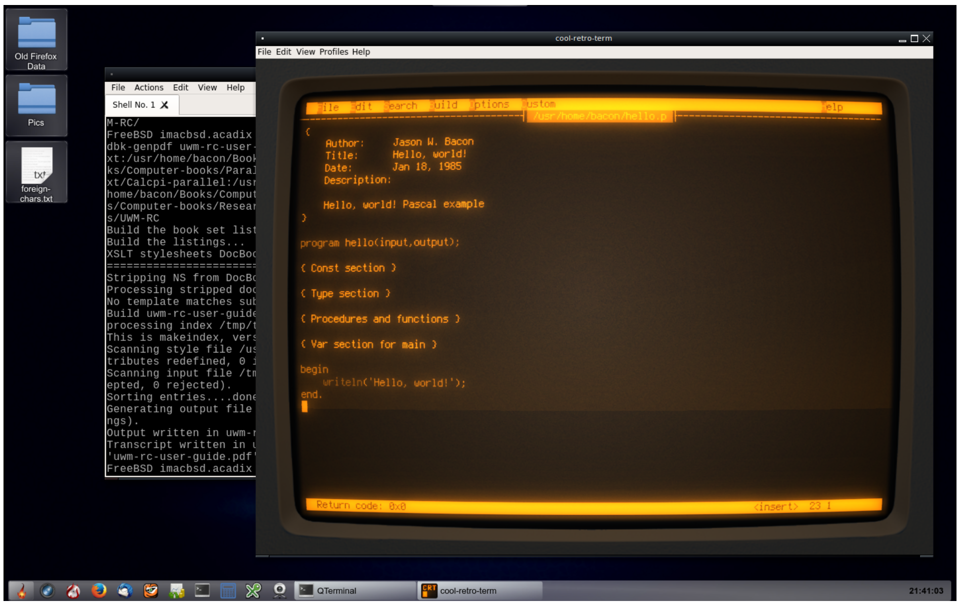


```
FreeBSD coral.acadix  bacon ~ 1011: 
```

*A Terminal emulator.*

All Unix systems come with a terminal emulator program. There are also free terminal emulators for Windows, which are discussed in Section 3.5.

For purists who *really* want to emulate a terminal, there's *Cool Retro Terminal* (CRT for short, which also happens to stand for cathode ray tube). This emulator comes complete with screen distortion and jitter to provide a genuine nostalgic 1970s experience.



```
M-RC/
FreeBSD imacbsd.acadix
dbk-genpdf uwm-rc-user-
xt:/usr/home/bacon/Book
ks/Computer-books/Paral
xt/Calcpi-parallel:/usr
home/bacon/Books/Comput
s/Computer-books/Resear
s/UWM-RC
Build the book set list
Build the listings...
XSLT stylesheets DocBoo
=======================
Stripping NS from DocBo
Processing stripped doc
No template matches sub
Build uwm-rc-user-guide
processing index /tmp/t
This is makeindex, vers
Scanning style file /us
tributes redefined, 0 i
Scanning input file /tm
epted, 0 rejected).
Sorting entries....done
Generating output file
ngs).
Output written in uwm-r
Transcript written in u
'uwm-rc-user-guide.pdf'
FreeBSD imacbsd.acadix
```

```
 ile  dit  earch  uild  ptions  ustom                                   elp
-------------------------| /usr/home/bacon/hello.p |--------------------------
{
    Author:     Jason W. Bacon
    Title:      Hello, world!
    Date:       Jan 18, 1985
    Description:

    Hello, world! Pascal example
}

program hello(input,output);

{ Const section }

{ Type section }

{ Procedures and functions }

{ Var section for main }

begin
    writeln('Hello, world!');
end.

 Return code: 0x0                                              <insert>  23 1
```

*Cool Retro Terminal*

### 3.3.5 Basic Shell Use

Once you're logged in and have a shell running in your terminal window, you're ready to start entering Unix commands.

The shell displays a *shell prompt*, such as "FreeBSD coral.acadix bacon ~ 1011:" in the image above, to indicate that it's waiting for you to enter the next command. The shell prompt can be customized by each user, so it may be different on each Unix system you use.

---

**Note**
For clarity, we primarily use the following to indicate a shell prompt in this text:

```
shell-prompt:
```

---

To enter a Unix command, you type the command on a single line, edit if necessary (using arrow keys to move around), and press **Enter** or **Return**.

We can also enter multiple Unix commands on the same line separated by semicolons.

Modern Unix shells allow commands to be extensively edited. Assuming your terminal type is properly identified by the Unix system, you can use the left and right arrow keys to move around, backspace and delete to remove characters (Ctrl+h serves as a backspace in some cases), and other key combinations to remove words, the rest of the line, etc. Learning the editing capabilities of your shell will make you a much faster Unix user, so it's a great investment of a small amount of time.

If you have access to a Unix system now, do the practice break below. This practice break is offered again in Section 3.5 for those who will be using a remote Unix system.

**Example 3.2** Practice Break

**Note** For this, and all practice breaks that follow, students should do the exercises shown. If you are reading this for a class, then these exercises are meant to be done in class. Try them on your own, do your best to understand what is happening, and ask the instructor for clarification if necessary.

Remotely log into another Unix system using the **ssh** command or PuTTY, or open a shell on your Mac or other Unix system. Then try the commands shown below.
Unix commands are below preceded by the shell prompt "shell-prompt: ". Other text refers to input to the program (command) currently running. You must exit that program before running another Unix command.
Lines beginning with '#' are comments to help you understand the text below, and not to be typed.
Don't worry if you're not clear on what these commands do. You do not need to memorize them right now. This exercise is only meant to help you understand the Unix CLI. Specific commands will be covered later.

```
# Print the current working directory
shell-prompt: pwd

# List files in the current working directory (folder)
shell-prompt: ls
shell-prompt: ls -al

# Two commands on the same line.  A ';' is the same as a newline in Unix.
shell-prompt: ls; ls /etc

# List files in the root directory
shell-prompt: ls /

# List commands in the /bin directory
shell-prompt: ls /bin

# Search the directory tree under /etc
shell-prompt: find /etc -name '*.conf'

# Create a subdirectory
shell-prompt: mkdir -p Data/IRC

# Change the current working directory to the new subdirectory
shell-prompt: cd Data/IRC

# Print the current working directory
shell-prompt: pwd

# See if the nano editor is installed
# nano is a simple text editor (like Notepad on Windows)
shell-prompt: which nano

    If this does not report "command not found", then do the following:

# Try the nano editor.  Nano is an add-on tool, not a standard tool on
# Unix systems.  Some systems will not have it installed.
shell-prompt: nano sample.txt

# Type the following text into the nano editor:

This is a text file called sample.txt.
I created it using the nano text editor on Unix.

# Then save the file (press Ctrl+o), and exit nano (press Ctrl+x).
# You should now be back at the Unix shell prompt.
# Try the "vi" editor
# vi is standard editor on all Unix system.  It is more complex than nano.
# It is good to know vi, since all Unix systems have it.
shell-prompt: vi sample.txt

    Type ’i’ to go into insert mode
    Type in some text
    Type Esc to exit insert mode and go back to command mode
    Type :w to save
    Type :q to quit

    # "ZZ" is a shortcut for ":w:q"

shell-prompt: ls

# Echo (concatenate) the contents of the new file to the terminal
shell-prompt: cat sample.txt

# Count lines, words, and characters in the file
shell-prompt: wc sample.txt

# Change the current working directory to your home directory
shell-prompt: cd
shell-prompt: pwd

# Show your login name
shell-prompt: id -un

# Show the name of the Unix system running your shell process
shell-prompt: hostname

# Show operating system and hardware info
shell-prompt: uname -a

# Today’s date
shell-prompt: date

# Display a simple calendar
shell-prompt: cal
shell-prompt: cal 2023
shell-prompt: cal nov 2018
shell-prompt: cal jan 3000

# CLI calculator with unlimited precision and many functions
shell-prompt: bc -l
scale=50
sqrt(2)
8^2
2^8
a=1
b=2
c=1
(-b+sqrt(b^2-4*a*c))/2*a
2*a
quit

# Show who is logged in and what they are running
shell-prompt: w
shell-prompt: finger

# How much disk space is used by the programs in /usr/local/bin?
```

```
shell-prompt: du -sh /usr/local/bin/

# Copy a file to the current working directory
shell-prompt: cp /etc/profile .
shell-prompt: ls

# View the copy
shell-prompt: cat profile

# View the original
shell-prompt: cat /etc/profile

# Remove the file
shell-prompt: rm profile
shell-prompt: ls

# Exit the shell (which logs you out from an ssh session)
# This can also be done by typing Ctrl+d, which is the ASCII/ISO
# character for EOT (end of transmission)
shell-prompt: exit
```

### 3.3.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a UI?
2. What is a GUI?
3. What is the difference between Unix and other operating systems with respect to the GUI?
4. How is Unix + X11 different from remote desktop systems and terminal servers?
5. What is a virtual desktop?
6. What are the two basic types of user interfaces? Which type is a GUI?
7. What is a CLI?
8. What types of applications are better suited for a menu-driven interface? Why?
9. What types of applications are better suited for a command-driven interface?
10. Which is easier to automate, a menu-driven system or a CLI? Why?
11. How many scientific programs offer a menu-driven interface? Why?
12. What is a shell?
13. What is a kernel?
14. What are libraries? What kinds of functionality do they provide?
15. What is a terminal?
16. What is a terminal emulator?
17. Do people still use hardware terminals today? Explain.
18. What is a shell prompt?

## 3.4 Still Need Windows? Don't Panic!

For those who need to run software that is only available for Windows, or those who simply haven't tried anything else yet, there are options for getting to know Unix while still using Windows for your daily work.

One option is to remotely log into a Unix system using a terminal application such as *PuTTY* on your Windows machine.

There are virtual machines (see Chapter 40) that allow us to run Windows and Unix on the same computer, at the same time. This is the best option for those who need a fully functional Unix environment on a Windows machine.

Cygwin is a compatibility layer that allows Unix software to be compiled and run on Windows. A compatibility layer is generally easier to install, but as of this writing, Cygwin has performance limitations in some areas. Purely computational software will run about as fast as it would on a real Unix system, but software that performs a lot of file input/output or other system calls can be much slower than a real Unix system, even one running in a virtual machine.

For example, installing the pkgsrc package manager from scratch, which involves running many Unix scripts and programs, required the times shown in Table 3.2. Cygwin, and the Hyper-V virtual machine were all run on the same Windows 10 host with a 2.6 GHz Core i7 processor and 4 GiB RAM. The native FreeBSD and Linux builds were run on identical 3.0 GHz Xeon servers with 16 GiB RAM, much older than the Core i7 Windows machine.

| Platform | Time |
|---|---|
| Cygwin | 71 minutes |
| FreeBSD Virtual Machine under Hyper-V | 21 minutes |
| CentOS Linux (3.0 GHz Xeon) | 6 minutes, 16 seconds |
| FreeBSD (3.0 GHz Xeon) | 5 minutes, 57 seconds |

Table 3.2: Pkgsrc Build Times

I highly recommend Cygwin as a light-duty Unix environment under Windows, for connecting to other Unix systems or developing small Unix programs. For serious Unix development or heavy computation, obtaining a real Unix system, even under a virtual machine, will produce better results.

### 3.4.1 Cygwin: Try This First

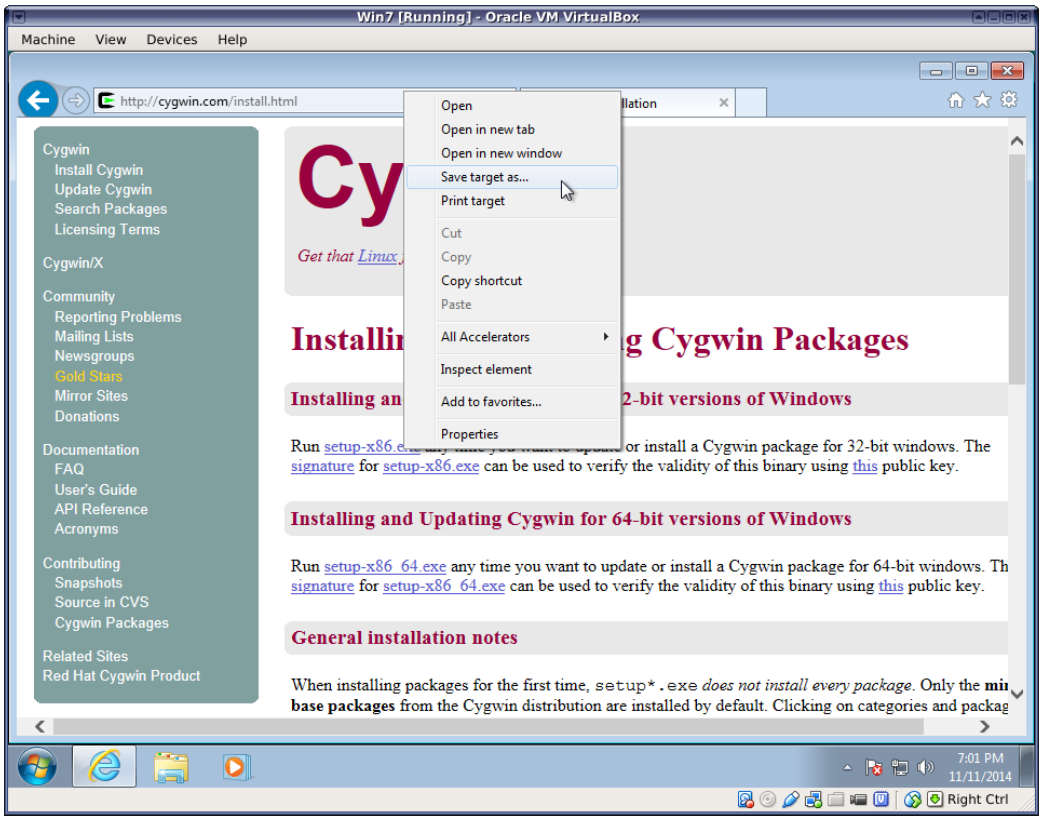
Cygwin is a free collection of Unix software, including many system tools from Linux and other Unix-compatible systems, ported to Windows. It can be installed on any typical Windows machine in about 10 minutes and allows users to experience a Unix user interface as well as run many popular Unix programs right on the Windows desktop.

Cygwin is a *compatibility layer*, another layer of software on top of Windows that translates the Unix API to the Windows API. As such, performance is not as good as a native Unix system on the same hardware, but it's more than adequate for many purposes. Cygwin may not be ideal for heavy-duty data analysis where optimal performance is required, but it is an excellent system for basic development and testing of Unix code and for interfacing with other Unix systems.

Cygwin won't break your Windows configuration, since it is completely self-contained in its own directory. Given that it's so easy to install and free of risk, there's no point wasting time wondering whether you should use Cygwin, a virtual machine, or some other method to get a Unix environment on your Windows PC. Try Cygwin first and if it fails to meet your needs, try something else.

Installing Cygwin is quick and easy:

1. Download **setup-x86_64.exe** from https://www.cygwin.com and save a copy on your desktop or some other convenient location. You will need this program to install additional packages in the future.



Cy

Get that Linux

Installi g Cygwin Packages

Installing an 2-bit versions of Windows

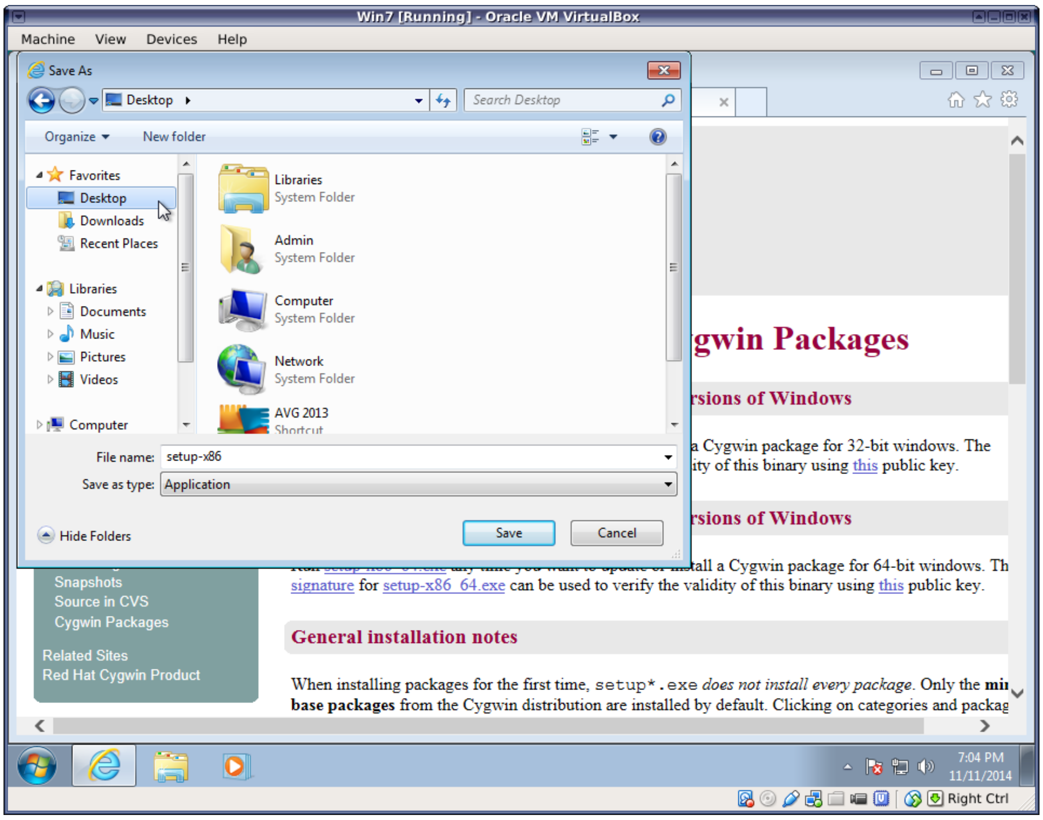
Run setup-x86.e or install a Cygwin package for 32-bit windows. The signature for setup-x86.exe can be used to verify the validity of this binary using this public key.

Installing and Updating Cygwin for 64-bit versions of Windows

Run setup-x86_64.exe any time you want to update or install a Cygwin package for 64-bit windows. Th signature for setup-x86_64.exe can be used to verify the validity of this binary using this public key.

General installation notes

When installing packages for the first time, `setup*.exe` *does not install every package*. Only the mi **base packages** from the Cygwin distribution are installed by default. Clicking on categories and packag



File name: setup-x86

Save as type: Application

gwin Packages

rsions of Windows

a Cygwin package for 32-bit windows. The ity of this binary using this public key.

rsions of Windows

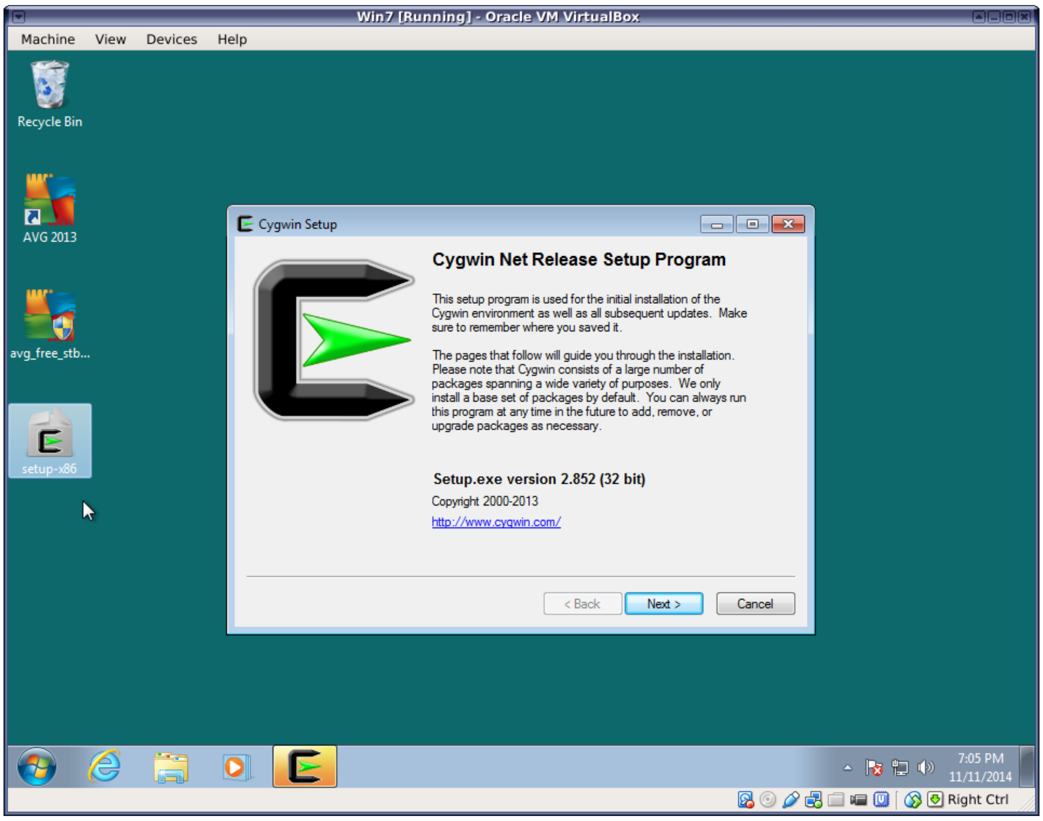
all a Cygwin package for 64-bit windows. Th signature for setup-x86_64.exe can be used to verify the validity of this binary using this public key.

General installation notes

When installing packages for the first time, `setup*.exe` *does not install every package*. Only the mi **base packages** from the Cygwin distribution are installed by default. Clicking on categories and packag

2. Run **setup-x86_64.exe** and follow the instructions on the screen.

   Unless you know what you're doing, accept the default answers to most questions. Some exceptions are noted below.



Cygwin Net Release Setup Program

This setup program is used for the initial installation of the Cygwin environment as well as all subsequent updates. Make sure to remember where you saved it.

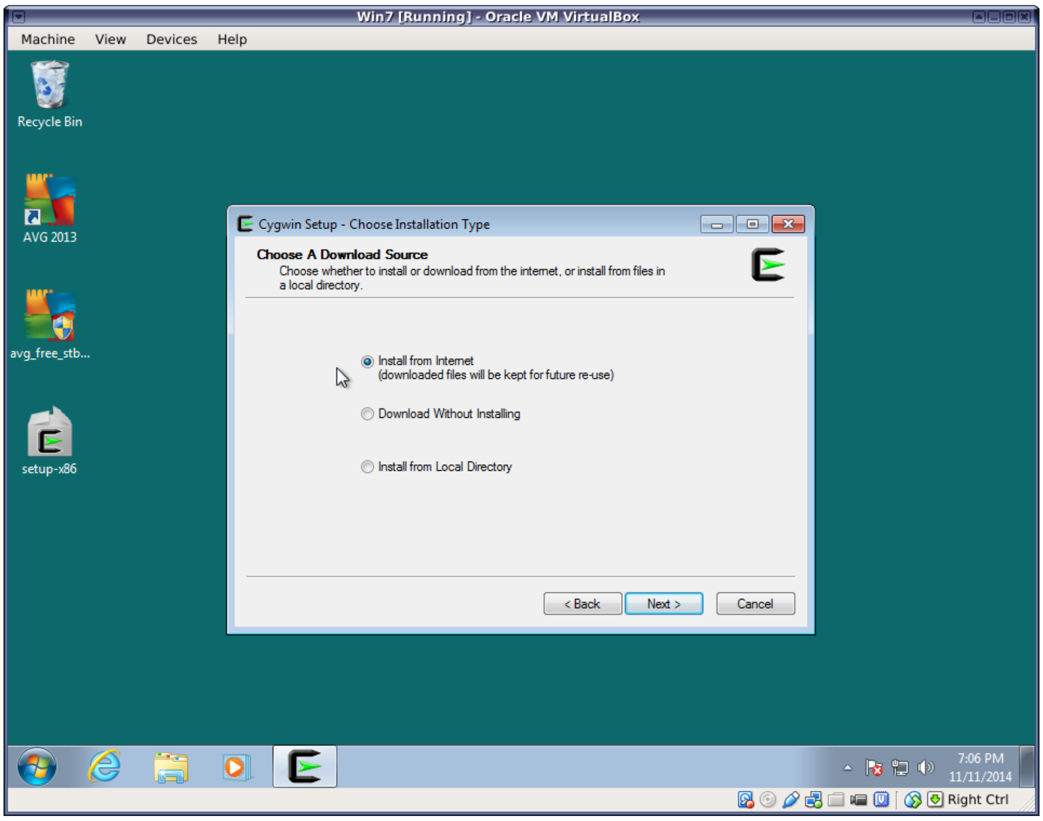
The pages that follow will guide you through the installation. Please note that Cygwin consists of a large number of packages spanning a wide variety of purposes. We only install a base set of packages by default. You can always run this program at any time in the future to add, remove, or upgrade packages as necessary.

Setup.exe version 2.852 (32 bit)

Copyright 2000-2013

http://www.cygwin.com/

3. Unless you know what you're doing, simply choose "Install from Internet".



Choose A Download Source

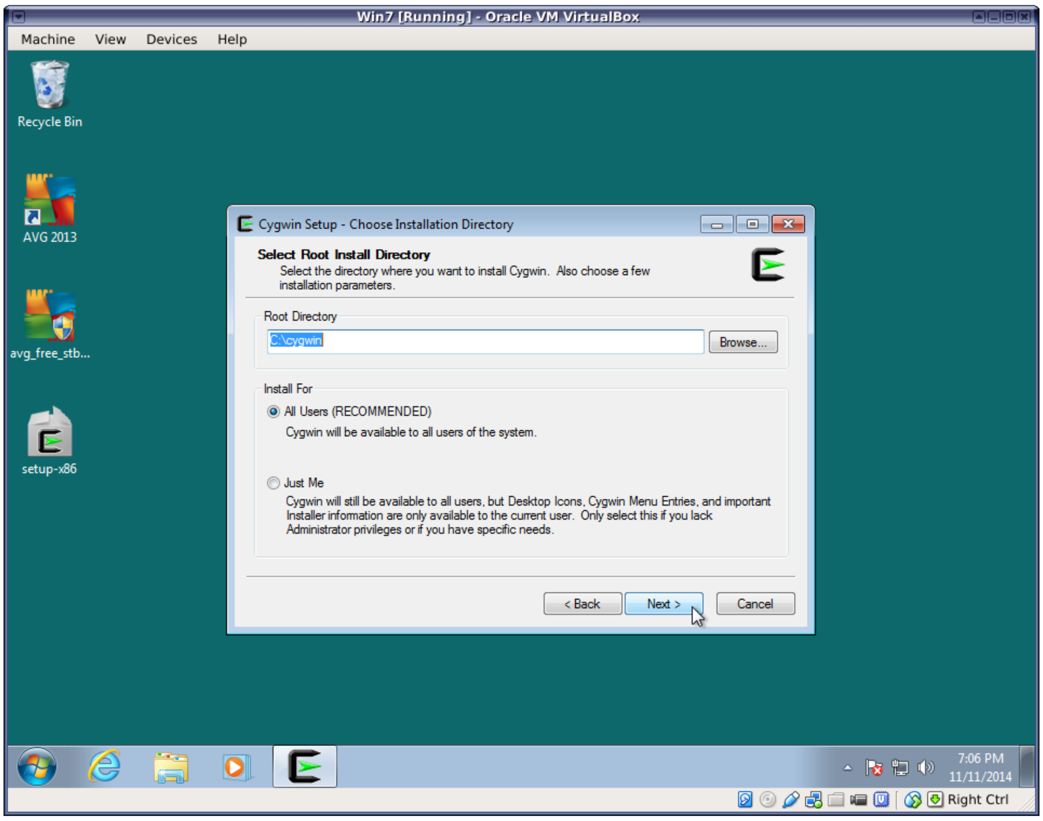
Choose whether to install or download from the internet, or install from files in a local directory.

[✓ Install from Internet (downloaded files will be kept for future re-use)]

Download Without Installing

Install from Local Directory

4. Select where you want to install the Cygwin files and whether to install for all users of this Windows machine.



Cygwin Setup - Choose Installation Directory

**Select Root Install Directory**

Select the directory where you want to install Cygwin. Also choose a few installation parameters.

Root Directory: C:\cygwin

Install For

[✓ All Users (RECOMMENDED)]

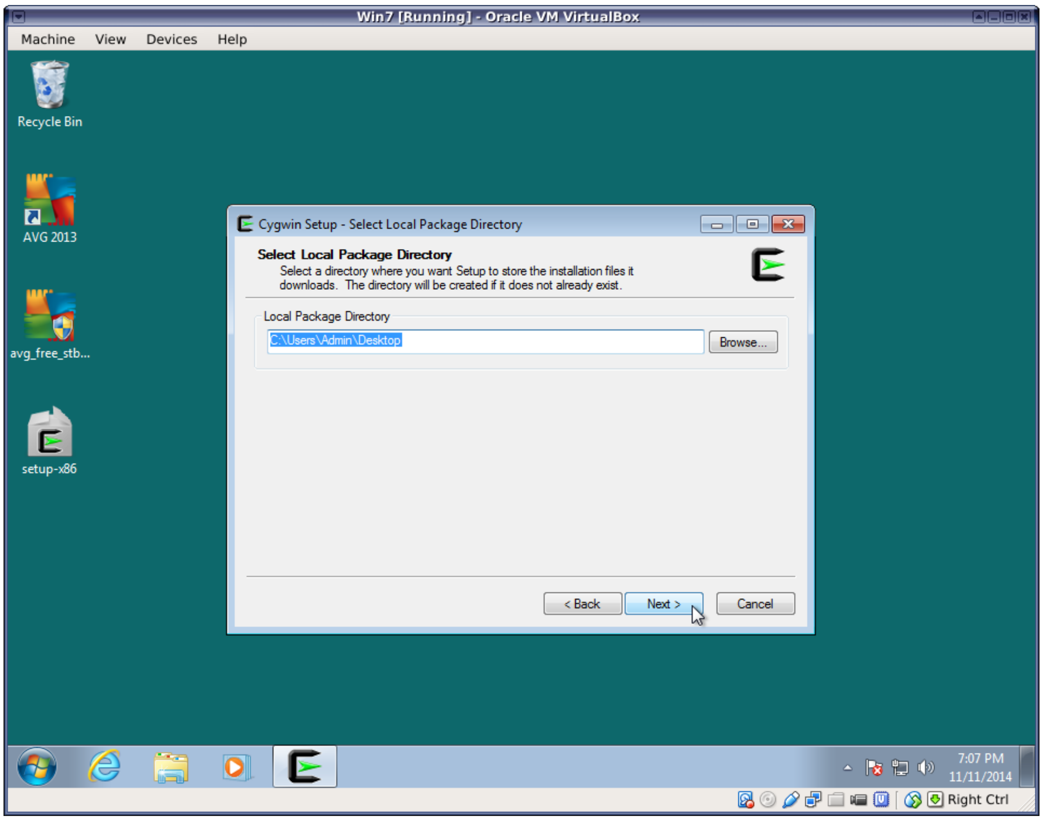
Cygwin will be available to all users of the system.

Just Me

Cygwin will still be available to all users, but Desktop Icons, Cygwin Menu Entries, and important Installer information are only available to the current user. Only select this if you lack Administrator privileges or if you have specific needs.

7:06 PM 11/11/2014

5. Select where to save downloaded packages. Again, the default location should work for most users.



Cygwin Setup - Select Local Package Directory

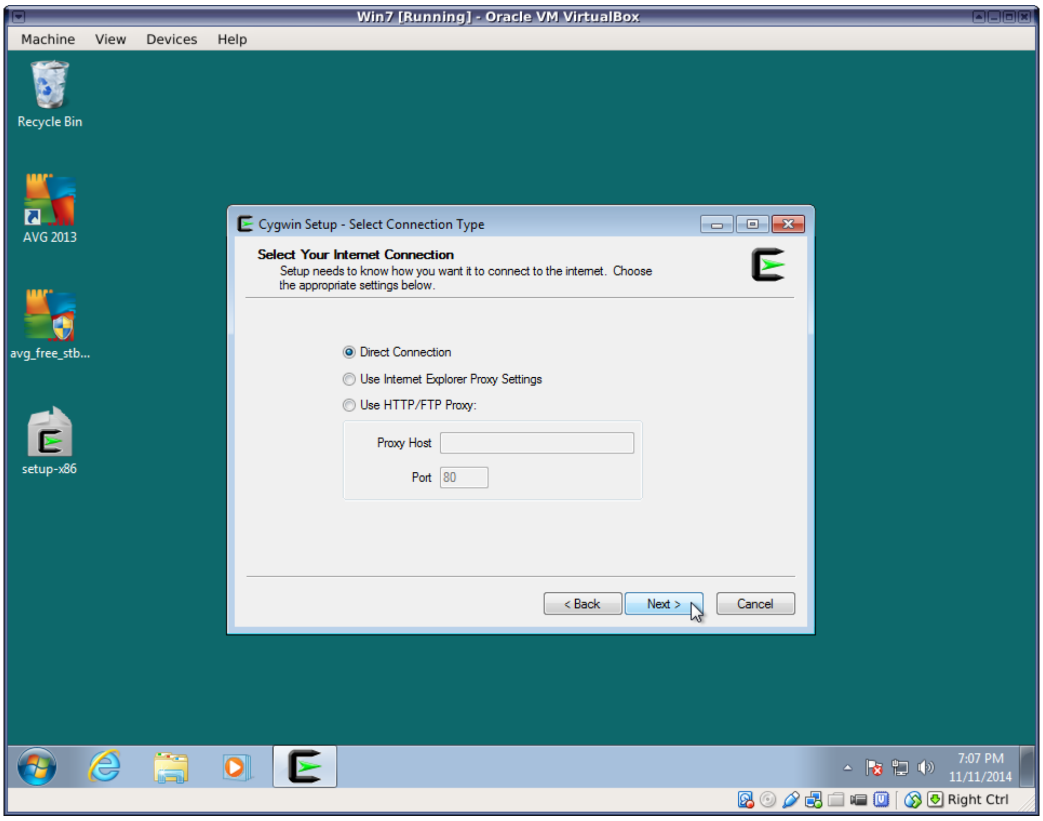
**Select Local Package Directory**

Select a directory where you want Setup to store the installation files it downloads. The directory will be created if it does not already exist.

Local Package Directory: C:\Users\Admin\Desktop

7:07 PM 11/11/2014

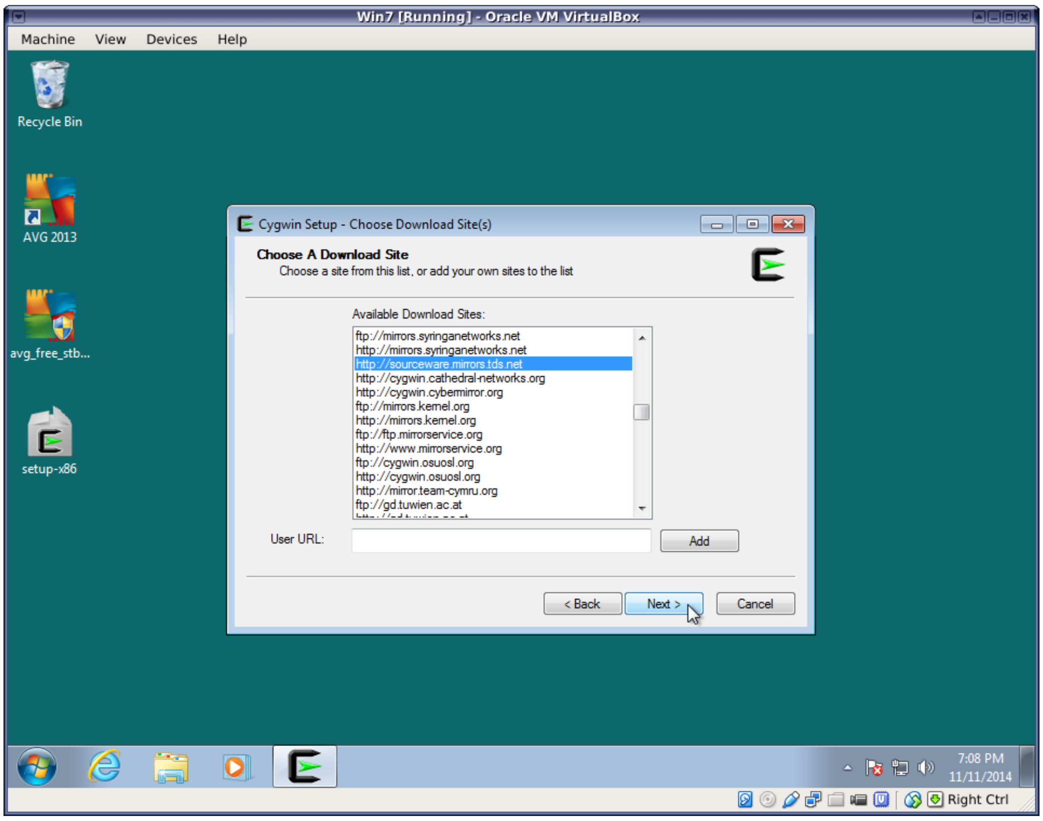
6. Select a network connection type.



**Select Your Internet Connection**
Setup needs to know how you want it to connect to the internet. Choose the appropriate settings below.

[✓ Direct Connection]
Use Internet Explorer Proxy Settings
Use HTTP/FTP Proxy:
Proxy Host
Port 80

7. Select a download site. It is very important here to select a site near you. Choosing a site far away can cause downloads to be incredibly slow. You may have to search the web to determine the location of each URL. This information is unfortunately not presented by the setup utility.



**Choose A Download Site**
Choose a site from this list, or add your own sites to the list

Available Download Sites:

```
ftp://mirrors.syringanetworks.net
http://mirrors.syringanetworks.net
http://sourceware.mirrors.tds.net
http://cygwin.cathedral-networks.org
http://cygwin.cybermirror.org
ftp://mirrors.kernel.org
http://mirrors.kernel.org
ftp://ftp.mirrorservice.org
http://www.mirrorservice.org
ftp://cygwin.osuosl.org
http://cygwin.osuosl.org
http://mirror.team-cymru.org
ftp://gd.tuwien.ac.at
```

User URL:

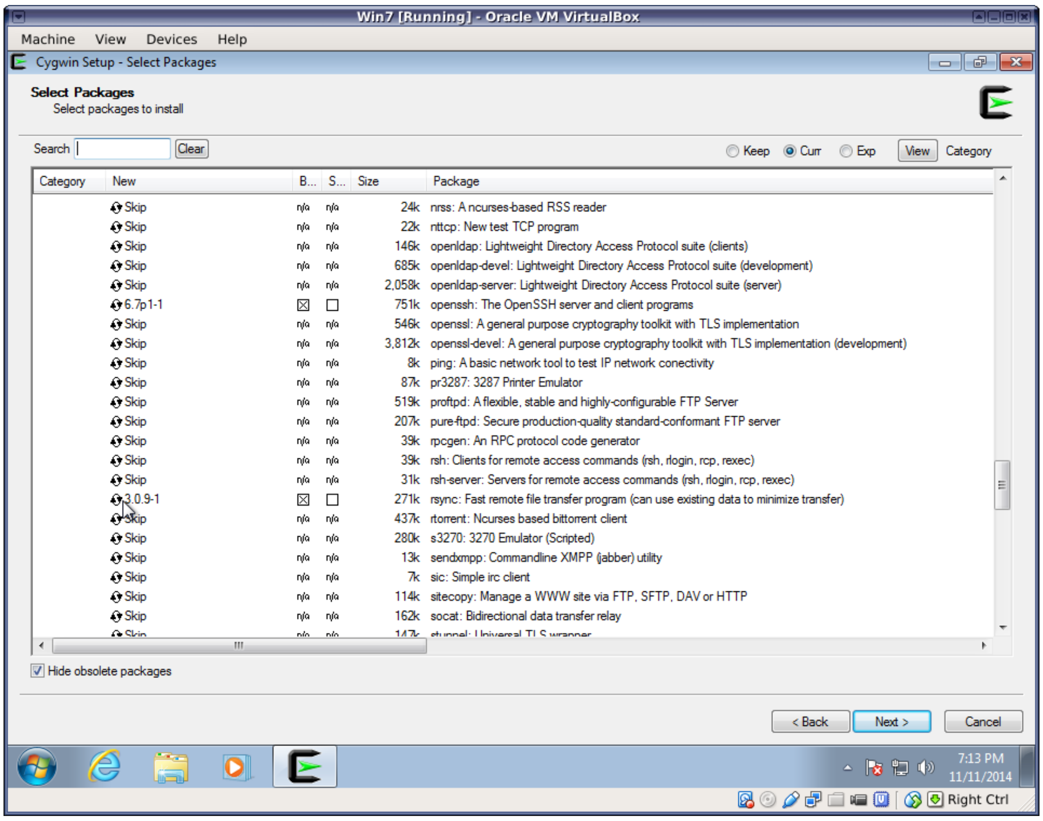
8. When you reach the package selection screen, select at least the following packages in addition to the basic installation:

- net/openssh
- net/rsync
- x11/xhost
- x11/xinit

This will install the **ssh** command as well as an X11 server, which will allow you to run graphical Unix programs on your Windows desktop. You may not need graphical capabilities immediately, but they will likely come in handy down the road.

The rsync package is especially useful if you'll be transferring large amounts of data back and forth between your Windows machine and remote servers.

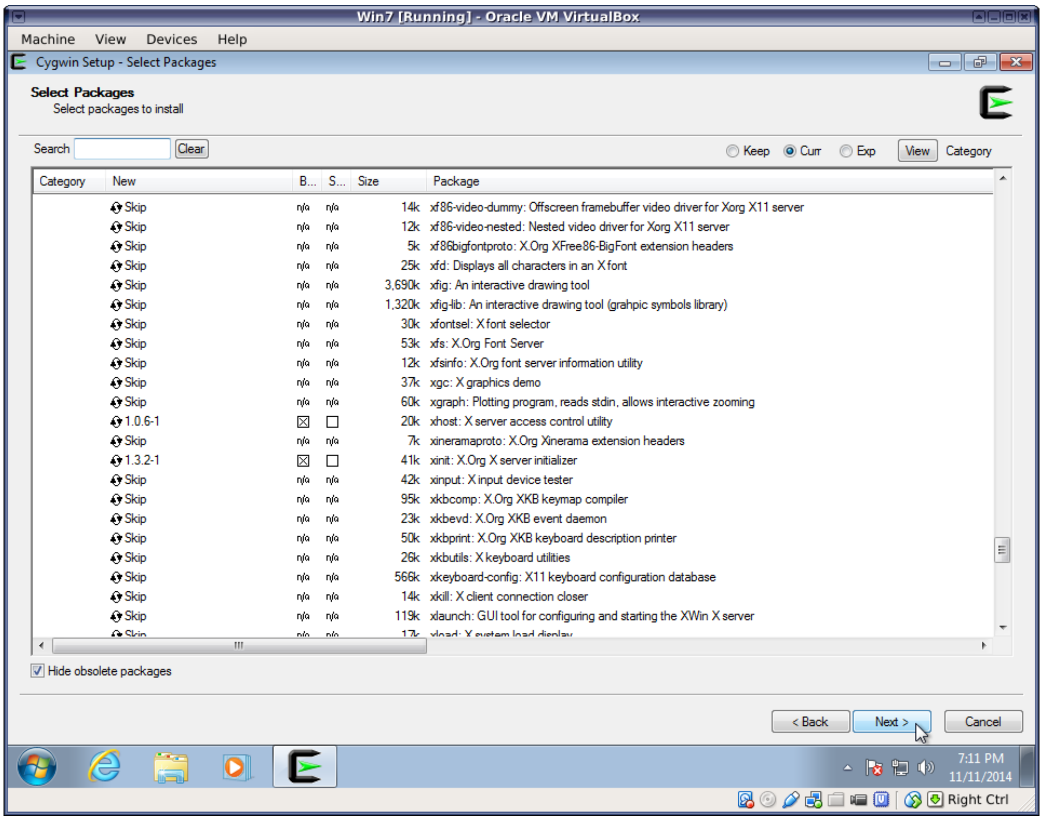
Click on the package categories displayed in order to expand them and see the packages under them.



Keep [✓ Curr] Exp

| Category | New | B... | S... | Size | Package |
|---|---|---|---|---|---|
| | Skip | n/a | n/a | 24k | nrss: A ncurses-based RSS reader |
| | Skip | n/a | n/a | 22k | nttcp: New test TCP program |
| | Skip | n/a | n/a | 146k | openldap: Lightweight Directory Access Protocol suite (clients) |
| | Skip | n/a | n/a | 685k | openldap-devel: Lightweight Directory Access Protocol suite (development) |
| | Skip | n/a | n/a | 2,058k | openldap-server: Lightweight Directory Access Protocol suite (server) |
| | 6.7p1-1 | ☒ | ☐ | 751k | openssh: The OpenSSH server and client programs |
| | Skip | n/a | n/a | 546k | openssl: A general purpose cryptography toolkit with TLS implementation |
| | Skip | n/a | n/a | 3,812k | openssl-devel: A general purpose cryptography toolkit with TLS implementation (development) |
| | Skip | n/a | n/a | 8k | ping: A basic network tool to test IP network conectivity |
| | Skip | n/a | n/a | 87k | pr3287: 3287 Printer Emulator |
| | Skip | n/a | n/a | 519k | proftpd: A flexible, stable and highly-configurable FTP Server |
| | Skip | n/a | n/a | 207k | pure-ftpd: Secure production-quality standard-conformant FTP server |
| | Skip | n/a | n/a | 39k | rpcgen: An RPC protocol code generator |
| | Skip | n/a | n/a | 39k | rsh: Clients for remote access commands (rsh, rlogin, rcp, rexec) |
| | Skip | n/a | n/a | 31k | rsh-server: Servers for remote access commands (rsh, rlogin, rcp, rexec) |
| | 3.0.9-1 | ☒ | ☐ | 271k | rsync: Fast remote file transfer program (can use existing data to minimize transfer) |
| | Skip | n/a | n/a | 437k | rtorrent: Ncurses based bittorrent client |
| | Skip | n/a | n/a | 280k | s3270: 3270 Emulator (Scripted) |
| | Skip | n/a | n/a | 13k | sendxmpp: Commandline XMPP (jabber) utility |
| | Skip | n/a | n/a | 7k | sic: Simple irc client |
| | Skip | n/a | n/a | 114k | sitecopy: Manage a WWW site via FTP, SFTP, DAV or HTTP |
| | Skip | n/a | n/a | 162k | socat: Bidirectional data transfer relay |
| | Skip | n/a | n/a | 147k | stunnel: Universal TLS wrapper |

☑ Hide obsolete packages


Win7 [Running] - Oracle VM VirtualBox
Machine View Devices Help
Cygwin Setup - Select Packages
Select Packages
Select packages to install
Search
Clear
Keep
Curr
Exp
View
Category
Hide obsolete packages
< Back
Next >
Cancel
7:11 PM
11/11/2014
Right Ctrl

| Category | New | B... | S... | Size | Package |
|---|---|---|---|---|---|
| | Skip | n/a | n/a | 14k | xf86-video-dummy: Offscreen framebuffer video driver for Xorg X11 server |
| | Skip | n/a | n/a | 12k | xf86-video-nested: Nested video driver for Xorg X11 server |
| | Skip | n/a | n/a | 5k | xf86bigfontproto: X.Org XFree86-BigFont extension headers |
| | Skip | n/a | n/a | 25k | xfd: Displays all characters in an X font |
| | Skip | n/a | n/a | 3,690k | xfig: An interactive drawing tool |
| | Skip | n/a | n/a | 1,320k | xfig-lib: An interactive drawing tool (grahpic symbols library) |
| | Skip | n/a | n/a | 30k | xfontsel: X font selector |
| | Skip | n/a | n/a | 53k | xfs: X.Org Font Server |
| | Skip | n/a | n/a | 12k | xfsinfo: X.Org font server information utility |
| | Skip | n/a | n/a | 37k | xgc: X graphics demo |
| | Skip | n/a | n/a | 60k | xgraph: Plotting program, reads stdin, allows interactive zooming |
| | 1.0.6-1 | ☒ | ☐ | 20k | xhost: X server access control utility |
| | Skip | n/a | n/a | 7k | xineramaproto: X.Org Xinerama extension headers |
| | 1.3.2-1 | ☒ | ☐ | 41k | xinit: X.Org X server initializer |
| | Skip | n/a | n/a | 42k | xinput: X input device tester |
| | Skip | n/a | n/a | 95k | xkbcomp: X.Org XKB keymap compiler |
| | Skip | n/a | n/a | 23k | xkbevd: X.Org XKB event daemon |
| | Skip | n/a | n/a | 50k | xkbprint: X.Org XKB keyboard description printer |
| | Skip | n/a | n/a | 26k | xkbutils: X keyboard utilities |
| | Skip | n/a | n/a | 566k | xkeyboard-config: X11 keyboard configuration database |
| | Skip | n/a | n/a | 14k | xkill: X client connection closer |
| | Skip | n/a | n/a | 119k | xlaunch: GUI tool for configuring and starting the XWin X server |
| | Skip | n/a | n/a | 17k | xload: X system load display |

Cygwin can also enable you to do Unix program development on your Windows machine. There are many packages providing Unix development tools such as compilers and editors, as well as libraries. The following is a small sample of common development packages:

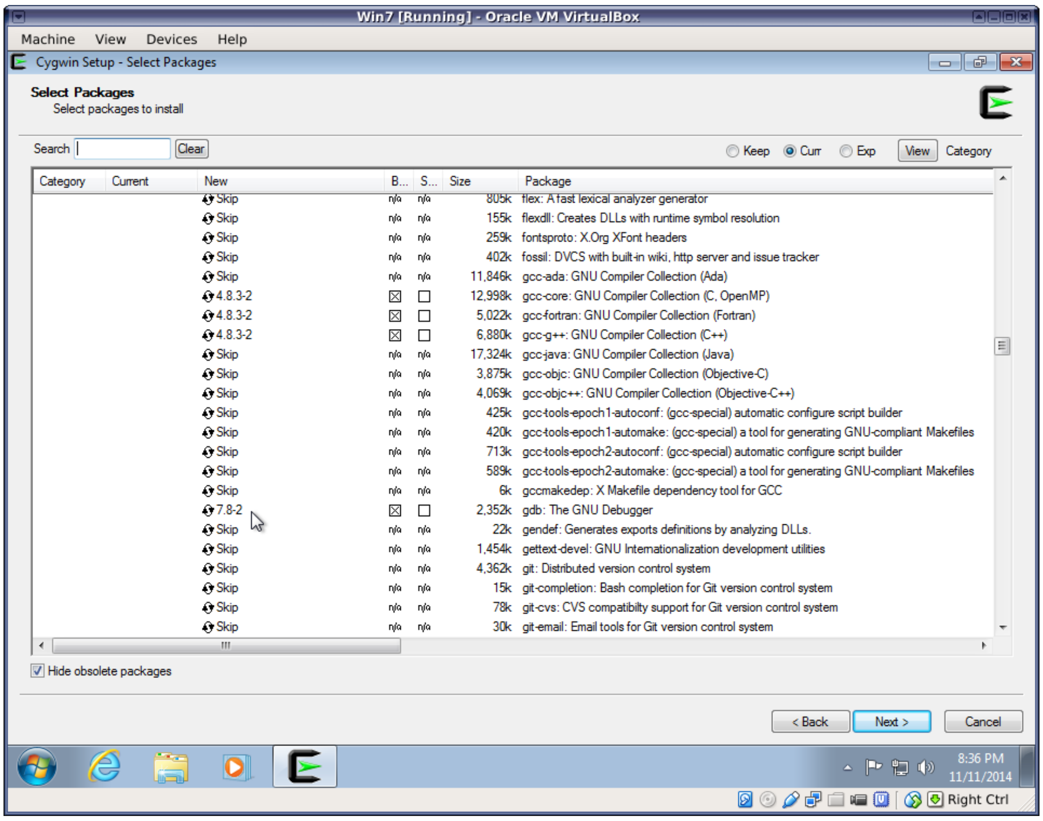
---

**Note**

Many of these programs are easier to install and update than their counterparts with a standard Windows interface. By running them under Cygwin, you are also practicing use of the Unix interface, which will make things easy for you when need to run them on a cluster or other Unix host that is more powerful than your PC.

---

- devel/clang (C/C++/ObjC compiler)
- devel/clang-analyzer (Development and debugging tool)
- devel/gcc-core (GNU Compiler Collection C compiler)
- devel/gcc-g++
- devel/gcc-gfortran
- devel/make (GNU make utility)
- editors/emacs (Text editor)
- editors/gvim (Text editor)
- editors/nano (Text editor)
- libs/openmpi (Distributed parallel programming tools)
- math/libopenblas (Basic Linear Algebra System libraries)
- math/lapack (Linear Algebra PACKage libraries)
- math/octave (Open source linear algebra system compatible with Matlab(r))
- math/R (Open source statistical language)



Win7 [Running] - Oracle VM VirtualBox
Machine View Devices Help
Cygwin Setup - Select Packages

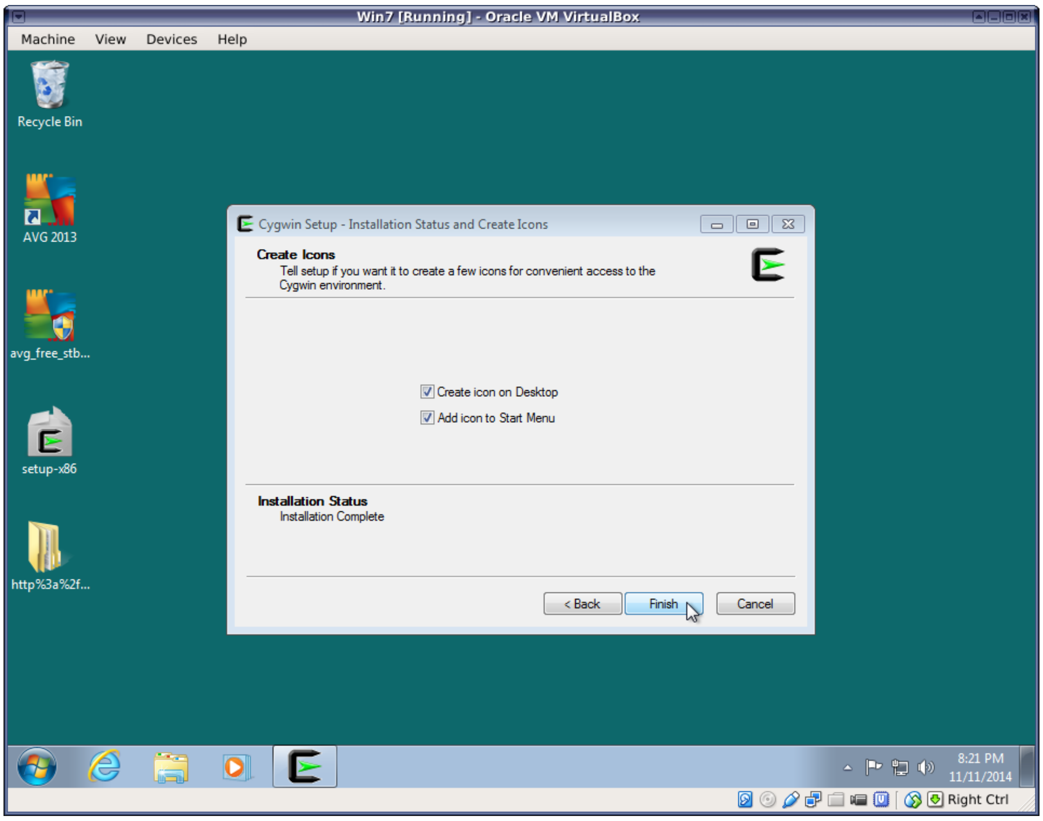
**Select Packages**
Select packages to install

Search [ ] Clear ( ) Keep (●) Curr ( ) Exp View Category

| Category | Current | New | B... | S... | Size | Package |
|---|---|---|---|---|---|---|
| | | Skip | n/a | n/a | 805k | flex: A fast lexical analyzer generator |
| | | Skip | n/a | n/a | 155k | flexdll: Creates DLLs with runtime symbol resolution |
| | | Skip | n/a | n/a | 259k | fontsproto: X.Org XFont headers |
| | | Skip | n/a | n/a | 402k | fossil: DVCS with built-in wiki, http server and issue tracker |
| | | Skip | n/a | n/a | 11,846k | gcc-ada: GNU Compiler Collection (Ada) |
| | | 4.8.3-2 | ☒ | ☐ | 12,998k | gcc-core: GNU Compiler Collection (C, OpenMP) |
| | | 4.8.3-2 | ☒ | ☐ | 5,022k | gcc-fortran: GNU Compiler Collection (Fortran) |
| | | 4.8.3-2 | ☒ | ☐ | 6,880k | gcc-g++: GNU Compiler Collection (C++) |
| | | Skip | n/a | n/a | 17,324k | gcc-java: GNU Compiler Collection (Java) |
| | | Skip | n/a | n/a | 3,875k | gcc-objc: GNU Compiler Collection (Objective-C) |
| | | Skip | n/a | n/a | 4,069k | gcc-objc++: GNU Compiler Collection (Objective-C++) |
| | | Skip | n/a | n/a | 425k | gcc-tools-epoch1-autoconf: (gcc-special) automatic configure script builder |
| | | Skip | n/a | n/a | 420k | gcc-tools-epoch1-automake: (gcc-special) a tool for generating GNU-compliant Makefiles |
| | | Skip | n/a | n/a | 713k | gcc-tools-epoch2-autoconf: (gcc-special) automatic configure script builder |
| | | Skip | n/a | n/a | 589k | gcc-tools-epoch2-automake: (gcc-special) a tool for generating GNU-compliant Makefiles |
| | | Skip | n/a | n/a | 6k | gccmakedep: X Makefile dependency tool for GCC |
| | | 7.8-2 | ☒ | ☐ | 2,352k | gdb: The GNU Debugger |
| | | Skip | n/a | n/a | 22k | gendef: Generates exports definitions by analyzing DLLs. |
| | | Skip | n/a | n/a | 1,454k | gettext-devel: GNU Internationalization development utilities |
| | | Skip | n/a | n/a | 4,362k | git: Distributed version control system |
| | | Skip | n/a | n/a | 15k | git-completion: Bash completion for Git version control system |
| | | Skip | n/a | n/a | 78k | git-cvs: CVS compatibilty support for Git version control system |
| | | Skip | n/a | n/a | 30k | git-email: Email tools for Git version control system |

☑ Hide obsolete packages

9. Most users will want to accept the default action of adding an icon to their desktop and to the Windows Start menu.



Win7 [Running] - Oracle VM VirtualBox
Machine View Devices Help
Recycle Bin
AVG 2013
avg_free_stb...
setup-x86
http%3a%2f...
Cygwin Setup - Installation Status and Create Icons

**Create Icons**
Tell setup if you want it to create a few icons for convenient access to the Cygwin environment.

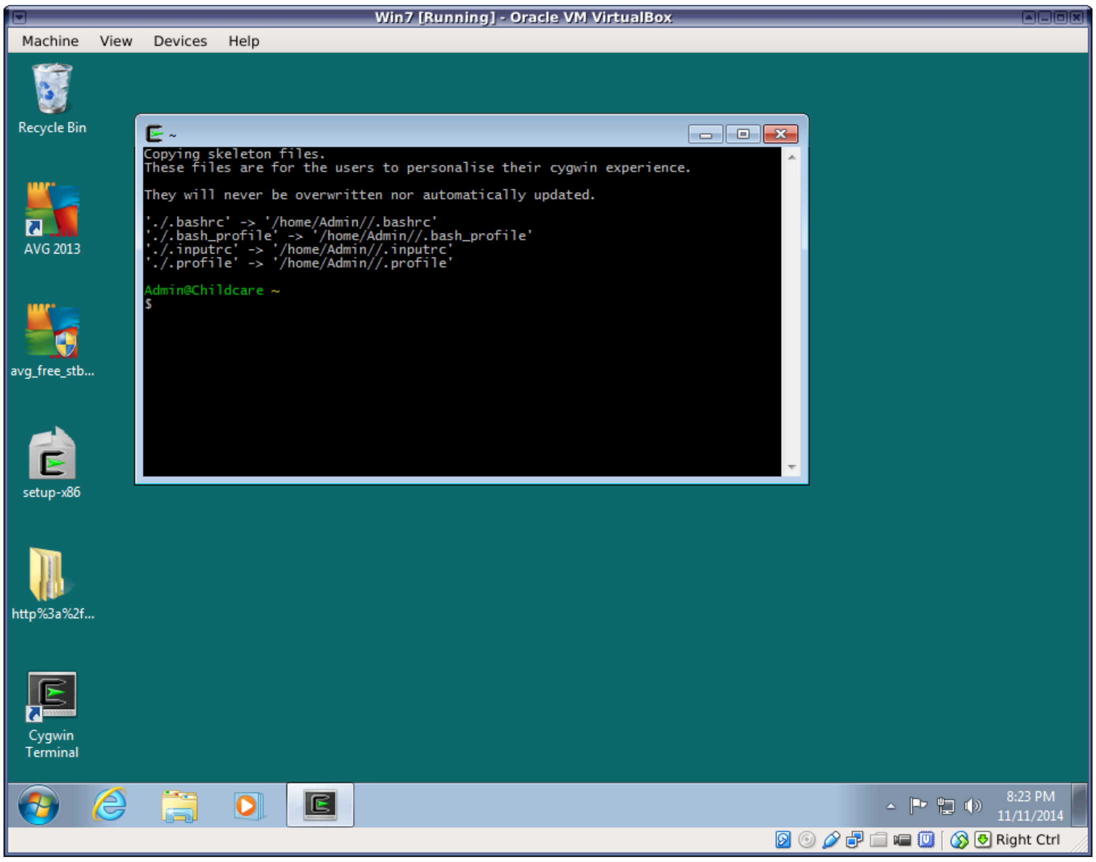
☑ Create icon on Desktop
☑ Add icon to Start Menu

**Installation Status**
Installation Complete

< Back
Finish
Cancel
8:21 PM
11/11/2014
Right Ctrl

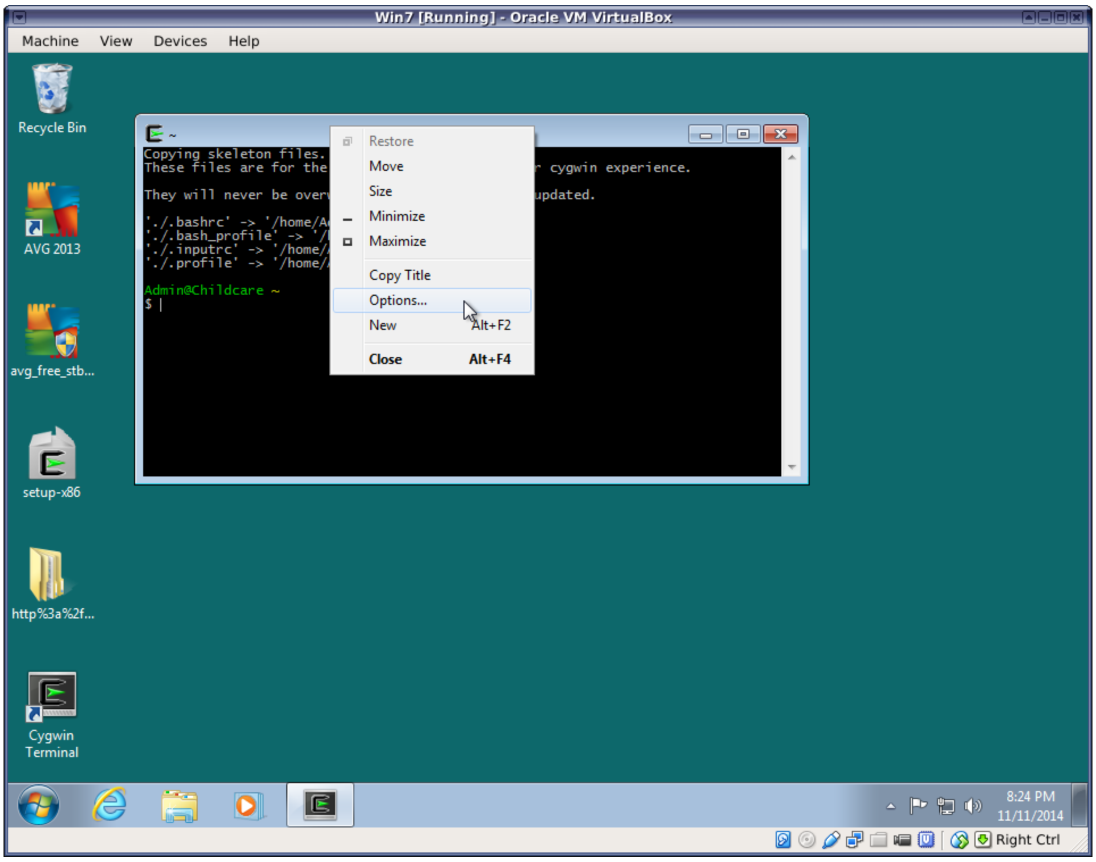
When the installation is complete, you will find Cygwin and Cygwin/X folders in your Windows program menu.

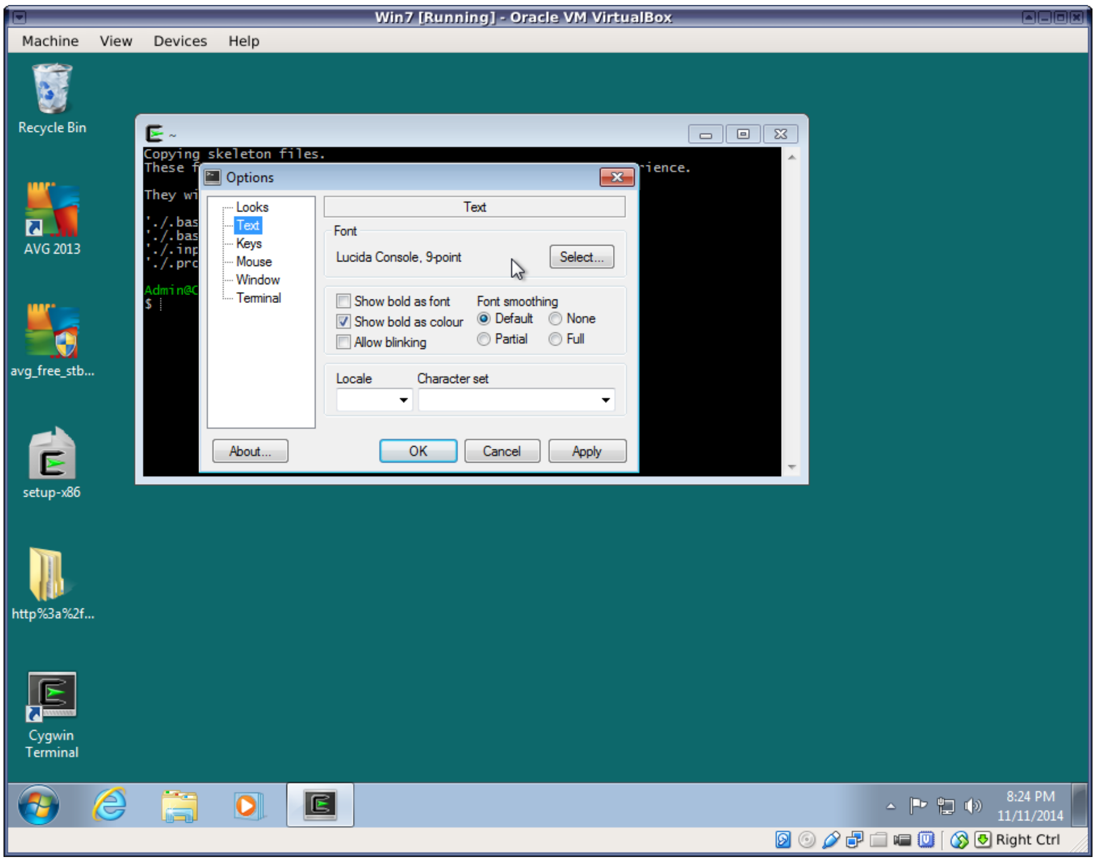
For a basic Terminal emulator, just run the Cygwin terminal:



```
Copying skeleton files.
These files are for the users to personalise their cygwin experience.

They will never be overwritten nor automatically updated.

'./.bashrc' -> '/home/Admin//.bashrc'
'./.bash_profile' -> '/home/Admin//.bash_profile'
'./.inputrc' -> '/home/Admin//.inputrc'
'./.profile' -> '/home/Admin//.profile'

Admin@Childcare ~
$
```

If you'd like to change the font size or colors of the Cygwin terminal emulator, just right-click on the title bar of the window:





```
Copying skeleton files.
These f                                               rience.

They wi

'./.bas
'./.bas
'./.inp
'./.prc

Admin@C
$
```

Text

Font: Lucida Console, 9-point — Select...

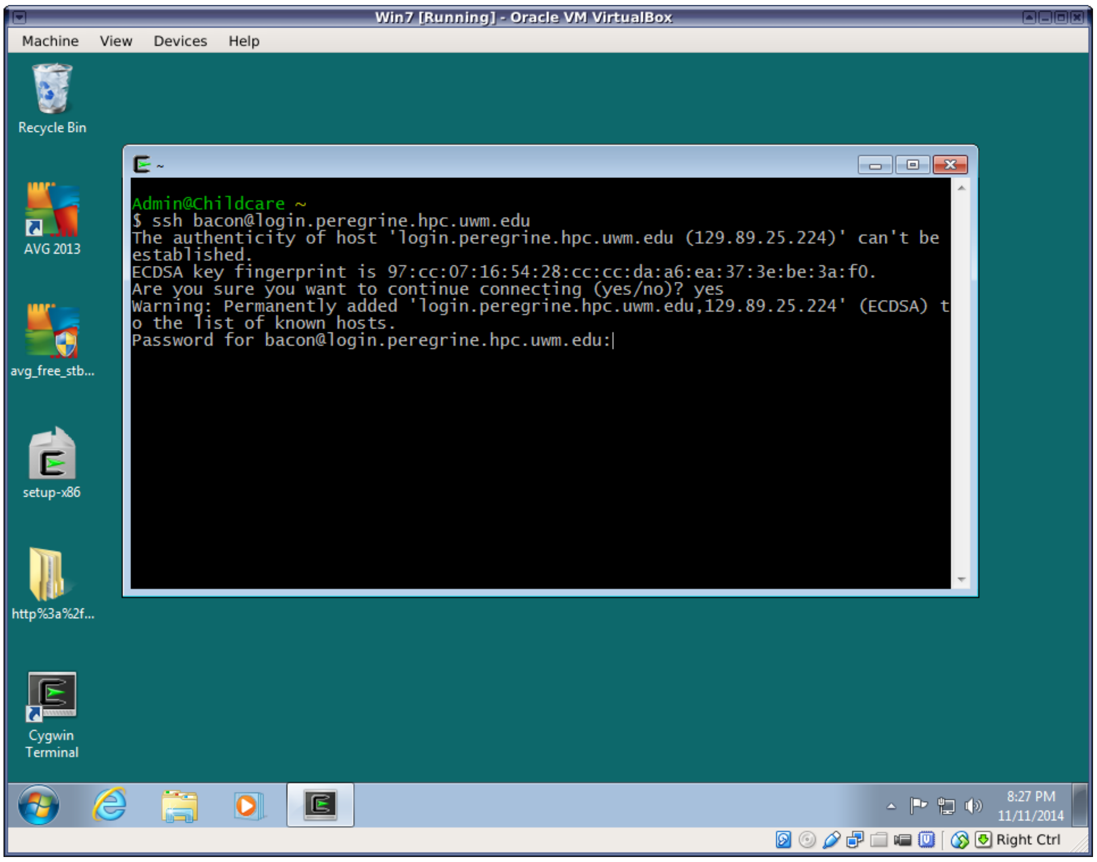
[ ] Show bold as font
[✓] Show bold as colour
[ ] Allow blinking

Font smoothing: [✓ Default] None Partial Full

Locale — Character set

Within the Cygwin terminal window, you are now running a "bash" Unix shell and can run most common Unix commands such as "ls", "pwd", etc.

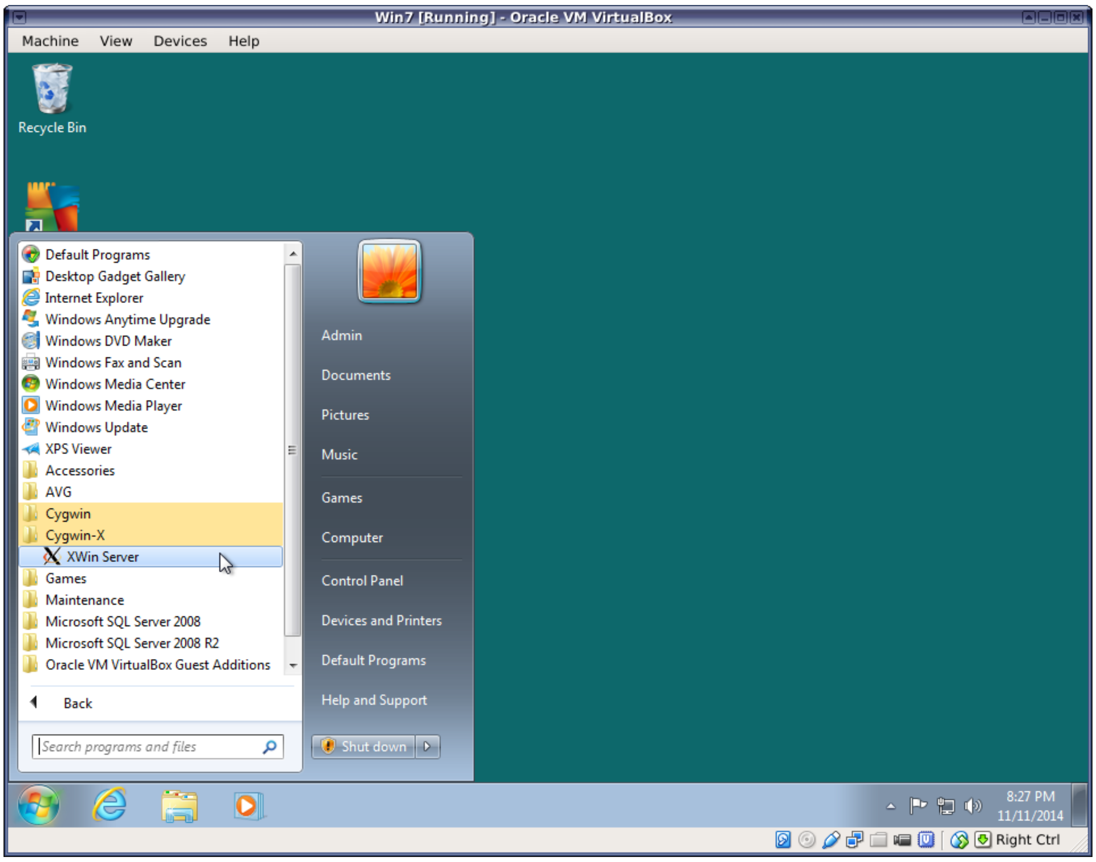
If you selected the openssh package during the Cygwin installation, you can now remotely log into other Unix machines, such as the clusters, over the network:



```
Admin@Childcare ~
$ ssh bacon@login.peregrine.hpc.uwm.edu
The authenticity of host 'login.peregrine.hpc.uwm.edu (129.89.25.224)' can't be
established.
ECDSA key fingerprint is 97:cc:07:16:54:28:cc:cc:da:a6:ea:37:3e:be:3a:f0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'login.peregrine.hpc.uwm.edu,129.89.25.224' (ECDSA) t
o the list of known hosts.
Password for bacon@login.peregrine.hpc.uwm.edu:
```

**Note** If you forgot to select the openssh package, just run the Cygwin setup program again. The packages you select when running it again will be added to your current installation.

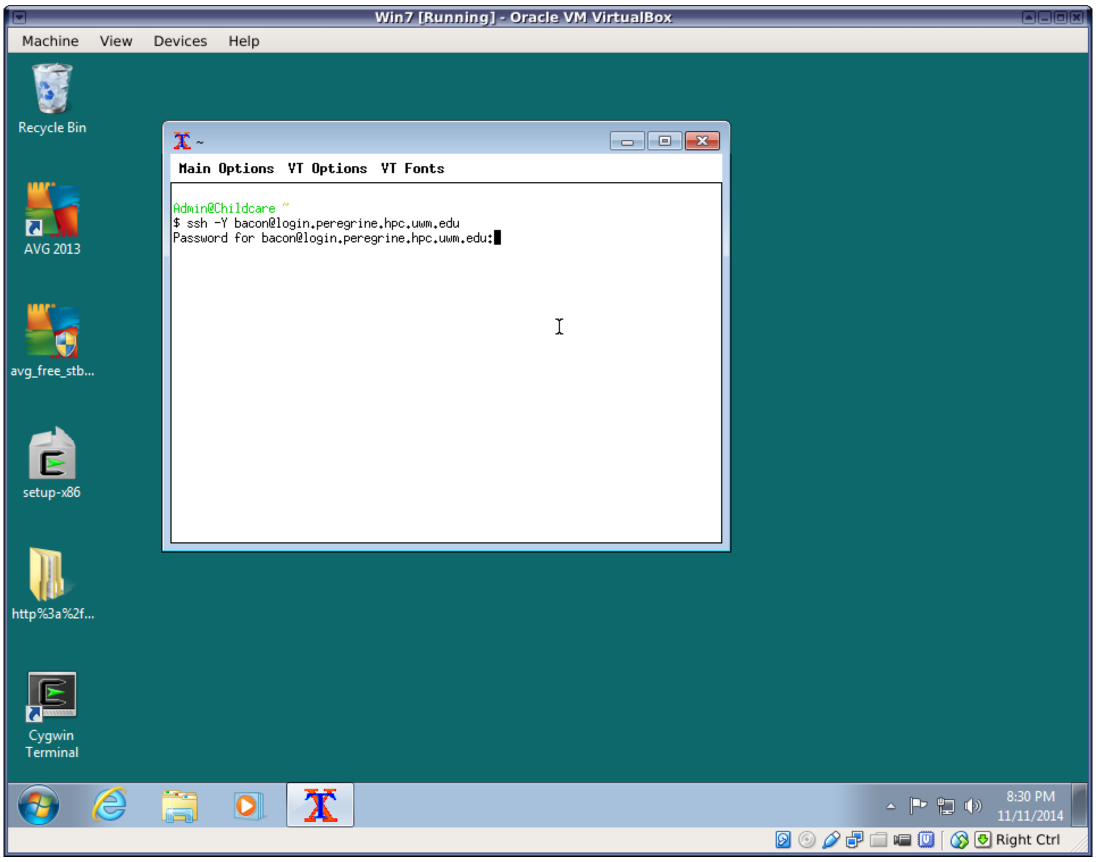
If you want to run Unix graphical applications, either on your Windows machine or on a remote Unix system, run the Cygwin/X application:



**Note** Doing graphics over a network may require a fast connection. If you are logging in from home or over a wireless connection, you may experience very sluggish rendering of windows from the remote host.

Depending on your Cygwin setup, this might automatically open a terminal emulator called "xterm", which is essentially the same as the standard Cygwin terminal, although it has a different appearance. You can use it to run all the same commands you would in the standard Cygwin terminal, including ssh. You may need to use the -X or -Y flag with ssh to enable some remote graphical programs.

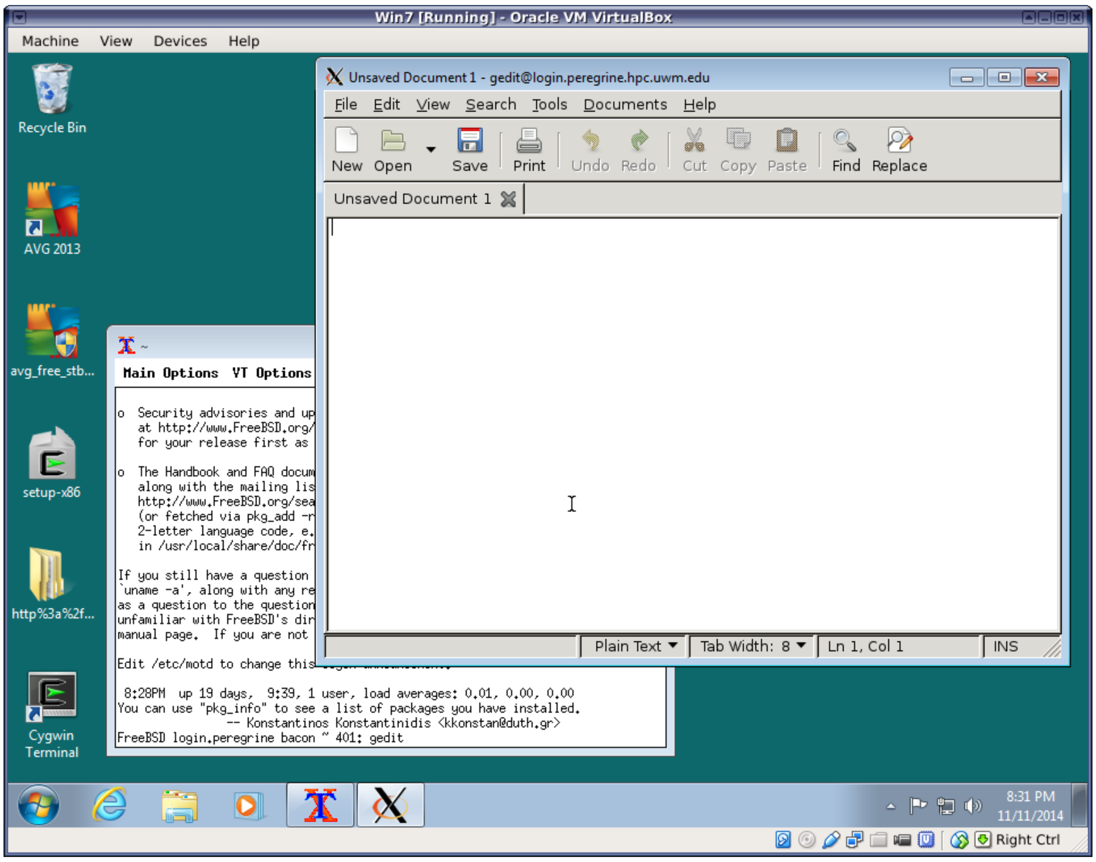
Unlike Cygwin Terminal, the xterm supplied with Cygwin/X is preconfigured to support graphical applications. See Section 3.19.3 for details.



```
Admin@Childcare ~
$ ssh -Y bacon@login.peregrine.hpc.uwm.edu
Password for bacon@login.peregrine.hpc.uwm.edu:
```

**Caution** Use of the -X and -Y flags could compromise the security of your Windows system by allowing malicious programs on the remote host to display phony windows on your PC. Use them *only* when logging into a trusted host.

Once you are logged into the remote host from the Cygwin/X xterm, you should be able to run graphical Unix programs.


Win7 [Running] - Oracle VM VirtualBox
Machine View Devices Help
Recycle Bin
AVG 2013
avg_free_stb...
setup-x86
http%3a%2f...
Cygwin Terminal
Unsaved Document 1 - gedit@login.peregrine.hpc.uwm.edu
File Edit View Search Tools Documents Help
New Open Save Print Undo Redo Cut Copy Paste Find Replace
Unsaved Document 1
Plain Text Tab Width: 8 Ln 1, Col 1 INS
Main Options VT Options
8:31 PM 11/11/2014
Right Ctrl

```
o  Security advisories and up
   at http://www.FreeBSD.org/
   for your release first as

o  The Handbook and FAQ docum
   along with the mailing lis
   http://www.FreeBSD.org/sea
   (or fetched via pkg_add -r
   2-letter language code, e.
   in /usr/local/share/doc/fr

If you still have a question
`uname -a', along with any re
as a question to the question
unfamiliar with FreeBSD's dir
manual page.  If you are not

Edit /etc/motd to change this

 8:28PM  up 19 days,  9:39, 1 user, load averages: 0.01, 0.00, 0.00
You can use "pkg_info" to see a list of packages you have installed.
                -- Konstantinos Konstantinidis <kkonstan@duth.gr>
FreeBSD login.peregrine bacon ~ 401: gedit
```

You can also run graphical applications from the standard Cygwin terminal if you update your start up script. If you are using bash (the Cygwin default shell), add the following line to your .bashrc file:

```
export DISPLAY=unix:0.0
```

You will need to run **source .bashrc** or restart your bash shell after making this change.

If you are using T shell, the line should read as follows in your .cshrc or .tcshrc:

```
setenv DISPLAY unix:0.0
```

Again, Cygwin is not the ideal way to run Unix programs on or from a Windows machine, but it is a very quick and easy way to gain access to a basic Unix environment and many Unix tools. Subsequent sections provide information about other options besides Cygwin for those with more sophisticated needs.

### 3.4.2 Windows Subsystem for Linux: Another Compatibility Layer

Windows Subsystem for Linux (WSL) is the latest in a chain of Unix compatibility products provided by Microsoft. It allows Windows to run a Linux environment under Microsoft's Hyper-V virtual machine monitor. As of this writing, the user can choose from a few different Linux distributions such as Ubuntu, Debian, SUSE, or Kali.

Differences from Cygwin:

- WSL runs actual Linux binaries (executables), whereas Cygwin allows the user to compile Unix programs into native Windows executables. Programs built under WSL can be run on a compatible Linux distribution and vice-versa. They cannot be run on Windows outside WSL. Which one you prefer depends on your specific goals. For many people, including most of us who just want to develop or run scientific programs, it makes no difference.
- WSL provides direct access to the native package collection of the chosen Linux distribution. For example, WSL users running the Debian app can install software directly from the Debian project using **apt**, just as they would on a real Debian system. The Debian package collection is much larger than Cygwin's, so if Cygwin does not have a package for software you need, WSL might be a good option.

- WSL is a virtual machine, based on Microsoft Hyper-V. It requires a substantial amount of memory and requires that virtualization features are enabled in the PC BIOS. If your Windows installation is running in a virtual machine, you must also have nested virtualization installed, and WSL performance will suffer.
- Cygwin is an independent open source project, while WSL is a Microsoft product. There are pros and cons to each. Microsoft could change or even terminate support for WSL, as it has done with previous Unix compatibility products, if it no longer appears to be in the company's interest to support it. Support for Cygwin will continue as long as the user community is willing to contribute.

### 3.4.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Describe three ways we can use Unix software on a Windows machine.
2. What is the advantage of Cygwin over a virtual machine?
3. What is the risk of using Cygwin?
4. What are two advantages of a virtual machine over Cygwin? Explain.
5. What is an advantage of Cygwin over WSL?

## 3.5 Logging In Remotely

Virtually all Unix systems allow users to log in and run programs over a network from other locations. This feature is intrinsic to Unix systems, and only disabled on certain proprietary or embedded installations. It is possible to use both GUIs and CLIs in this fashion, although GUIs may not work well over slow connections such as a slower home Internet services. Different graphical programs have vastly different bandwidth demands. Some will work fine over a DSL, cable, or WiFi connection, while others require a fast wired connection.

The command line interface, on the other hand, works comfortably on even the slowest network connections.

Logging into a Unix CLI from a remote location is usually done using *Secure Shell (SSH)*.

**Caution** Older protocols such as rlogin, rsh, and telnet, should no longer be used due to their lack of security. These protocols transport passwords over the Internet in unencrypted form, so people who manage the gateway computers they pass through can easily read them.

### 3.5.1 Unix to Unix

If you want to remotely log in from one Unix system to another, you can simply use the **ssh** command from the command line. The general syntax of the **ssh** command is:

```
ssh [flags] login-id@hostname
```

The login-id portion is your login name on the remote host. If you are logging into a campus-managed server, this is likely the same campus login ID used to log into other services such as VPN, email, Canvas, etc.

The first time you connect to each remote host, you will be asked to verify that you trust it. You must enter the full word "yes" to continue:

```
The authenticity of host 'unixdev1.ceas.uwm.edu (129.89.25.223)' can't be established.
ED25519 key fingerprint is SHA256:askjdkj2ksjfdfamnmnmw5lka7jdkjka,mksjdkssfj.
No matching host key fingerprint found in DNS.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
```

If the connection is successful, you will be asked for your password. Note that nothing will echo to your screen as you type the password. Login panels that echo a '*' or a dot for each character are less secure, since someone looking over your shoulder can see exactly how long your password is. Knowing the length reduces the parameter space they would have to search in order to guess the password.

If you plan to run graphical programs on the remote Unix system, you may need to include the `-X` (enable X11 forwarding) or `-Y` (enable trusted X11 forwarding) flag in your **ssh** command. Run **man ssh** for full details.

**Caution** Use `-X` or `-Y` only when connecting to trusted computers, i.e. those managed by you or someone you trust. These options allow the remote system to access your display, which can pose a security risk. For example, a hacker on the remote system could display a fake login panel on your screen in order to steal your login and password.

**Caution** Only ssh should be used to log into remote systems. Older commands such as rsh and telnet lack encryption and are not secure. If anyone tells you to use rsh or telnet, they should not be trusted regarding any computing issues.

Examples:

```
shell-prompt: ssh joe@unixdev1.ceas.uwm.edu
```

**Note** For licensing reasons, **ssh** may not be included in basic Linux installations, but it can be very easily added via the package management system of most Linux distributions.

If you have X11 capabilities and used -X or -Y with your ssh command, you can easily open additional terminals from the command-line if you know the name of the terminal emulator. Simply type the name of the terminal emulator, followed by an '&' to put it in the background. ( See Section 3.18.3 for a full explanation of background jobs. ) Some common terminal emulators are coreterminal, konsole, urxvt, and xterm.

```
shell-prompt: coreterminal &
```

### 3.5.2 Windows to Unix

If you're connecting to a Unix system from a Windows system, you will need to install some additional software.

#### Cygwin

The Cygwin Unix-compatibility system is free, quick and easy to install, and equips a Windows computer with most common Unix commands, including a Unix-style Terminal emulator. Once Cygwin is installed, you can open a Cygwin terminal on your Windows desktop and use the **ssh** command as shown above.

The Cygwin installation is very quick and easy and is described in Section 3.4.1.

#### PuTTY

A more limited method for remotely accessing Unix systems is to install a stand-alone terminal emulator, such as PuTTY, https://www.chiark.greenend.org.uk/~sgtatham/putty/. PuTTY has a built-in **ssh** client, and a graphical dialog box for connecting to a remote machine. For more information, see the PuTTY documentation.

### 3.5.3 Terminal Types

In rare cases, you may be asked to specify a terminal type when you log in:

```
TERM=(unknown)
```

Terminal features such as cursor movement and color changes are triggered by sending special codes (characters or character combinations called *magic sequences*) to the terminal. Pressing keys on the terminal sends codes from the terminal to the computer.

Different types of terminals use different magic sequences. PuTTY and most other terminal emulators emulate an "xterm" terminal, so if asked, just type the string "xterm" (without the quotes).

If you fail to set the terminal type, some programs such as text editors will not function. They may garble the screen and fail to recognize special keys such as arrows, page-up, etc.

You can set the terminal type after logging in, but the methods for doing this vary according to which shell you use, so you may just want to log out and remember to set the terminal type when you log back in.

**Example 3.3** Practice Break

**Note** For this, and all practice breaks that follow, students should do the exercises shown. If you are reading this for a class, then these exercises are meant to be done in class. Try them on your own, do your best to understand what is happening, and ask the instructor for clarification if necessary.

Remotely log into another Unix system using the **ssh** command or PuTTY, or open a shell on your Mac or other Unix system. Then try the commands shown below.
Unix commands are below preceded by the shell prompt "shell-prompt: ". Other text refers to input to the program (command) currently running. You must exit that program before running another Unix command.
Lines beginning with '#' are comments to help you understand the text below, and not to be typed.
Don't worry if you're not clear on what these commands do. You do not need to memorize them right now. This exercise is only meant to help you understand the Unix CLI. Specific commands will be covered later.

```
# Print the current working directory
shell-prompt: pwd

# List files in the current working directory (folder)
shell-prompt: ls
shell-prompt: ls -al

# Two commands on the same line.  A ';' is the same as a newline in Unix.
shell-prompt: ls; ls /etc

# List files in the root directory
shell-prompt: ls /

# List commands in the /bin directory
shell-prompt: ls /bin

# Search the directory tree under /etc
shell-prompt: find /etc -name '*.conf'

# Create a subdirectory
```

```
shell-prompt: mkdir -p Data/IRC

# Change the current working directory to the new subdirectory
shell-prompt: cd Data/IRC

# Print the current working directory
shell-prompt: pwd

# See if the nano editor is installed
# nano is a simple text editor (like Notepad on Windows)
shell-prompt: which nano

    If this does not report "command not found", then do the following:

# Try the nano editor.  Nano is an add-on tool, not a standard tool on
# Unix systems.  Some systems will not have it installed.
shell-prompt: nano sample.txt

# Type the following text into the nano editor:

This is a text file called sample.txt.
I created it using the nano text editor on Unix.

# Then save the file (press Ctrl+o), and exit nano (press Ctrl+x).
# You should now be back at the Unix shell prompt.

# Try the "vi" editor
# vi is standard editor on all Unix system.  It is more complex than nano.
# It is good to know vi, since all Unix systems have it.
shell-prompt: vi sample.txt

    Type 'i' to go into insert mode
    Type in some text
    Type Esc to exit insert mode and go back to command mode
    Type :w to save
    Type :q to quit

    # "ZZ" is a shortcut for ":w:q"

shell-prompt: ls

# Echo (concatenate) the contents of the new file to the terminal
shell-prompt: cat sample.txt

# Count lines, words, and characters in the file
shell-prompt: wc sample.txt

# Change the current working directory to your home directory
shell-prompt: cd
shell-prompt: pwd

# Show your login name
shell-prompt: id -un

# Show the name of the Unix system running your shell process
shell-prompt: hostname

# Show operating system and hardware info
shell-prompt: uname -a

# Today's date
shell-prompt: date
```

```
# Display a simple calendar
shell-prompt: cal
shell-prompt: cal 2023
shell-prompt: cal nov 2018
shell-prompt: cal jan 3000

# CLI calculator with unlimited precision and many functions
shell-prompt: bc -l
scale=50
sqrt(2)
8^2
2^8
a=1
b=2
c=1
(-b+sqrt(b^2-4*a*c))/2*a
2*a
quit

# Show who is logged in and what they are running
shell-prompt: w
shell-prompt: finger

# How much disk space is used by the programs in /usr/local/bin?
shell-prompt: du -sh /usr/local/bin/

# Copy a file to the current working directory
shell-prompt: cp /etc/profile .
shell-prompt: ls

# View the copy
shell-prompt: cat profile

# View the original
shell-prompt: cat /etc/profile

# Remove the file
shell-prompt: rm profile
shell-prompt: ls

# Exit the shell (which logs you out from an ssh session)
# This can also be done by typing Ctrl+d, which is the ASCII/ISO
# character for EOT (end of transmission)
shell-prompt: exit
```

### 3.5.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What must be added to Unix to allow remote access?
2. Can we run graphical programs on remote Unix systems? Elaborate.
3. Does the CLI require a fast connection for remote operation?
4. What command would you use to log into a remote system with host name "myserver.mydomain.edu" using the user name "joe", assuming you want to run a graphical X11 application?

5. What should you do if someone advises you to use rsh or telnet?
6. How can Windows users add an ssh command like the one used on Unix systems?
7. What is the purpose of the TERM environment variable? What will happen if it is not set correctly?

## 3.6 Unix Command Basics

A Unix command is built from a command name and optionally one or more command line *arguments*. Arguments can be either *flags* or data.

```
ls -a -l /etc /var
^^ ^^^^^ ^^^^^^^^^
|   |     |
|   |     Data Arguments
|   Flags
Command name
```

- The *command name* is either the filename of a program or a command built into the shell. For example, the **ls** command is a program that lists the contents of a directory. The **cd** command is part of the shell.
- Most commands have optional *flags* (sometimes called *options*) that control the behavior of the command. By convention, flags begin with a '-' character.

  **Note** Unix systems do not enforce this, but very few commands violate it. Unix programmers tend to understand the benefits of conventions and don't have to be coerced to follow them.

  The flags in the example above have the following meaning:

  `-a`: tells **ls** to show "hidden" files (files whose names begin with '.', which ls would not normally list).

  `-l`: tells **ls** to do a "long listing", which is to show lots of information about each file and directory instead of just the name.

  Single-letter flags can usually be combined, e.g. `-a -l` can be abbreviated as `-al`.

  Most newer Unix commands also support long flag names, mainly to improve readability of commands used in scripts. For example, in the Unix **zip** command, `-C` and `--preserve-case` are synonymous. Using -C saves typing, but --preserve-case is more easily understood.
- Many commands also accept one or more data arguments, which provide input data to the command, or instruct it where to send output. Such arguments may be the actual input data or they may be the names of files or directories that contain input or receive output. The `/etc` and `/var` arguments above are directories to be listed by **ls**. If no data arguments are given to **ls**, it lists the current working directory (described in Section 3.8).

For many Unix commands, the flags must come before the data arguments. A few commands require that certain flags appear in a specific order. Some commands allow flags and data arguments to appear in any order. Unix systems do not enforce any rules regarding arguments. How they behave is entirely up to the programmer writing the command. However, the vast majority of commands follow conventions, so there is a great deal of consistency in Unix command syntax.

The components of a Unix command are separated by white space (space or tab characters). Hence, if an argument contains any white space, it must be enclosed in quotes (single or double) so that it will not be interpreted as multiple separate arguments.

**Example 3.4** White space in an Argument

Suppose you have a directory called `My Programs`, and you want to see what's in it. You might try the following:

```
shell-prompt: ls My Programs
```

The above command fails because "My" and "Programs" are interpreted as two separate arguments. The **ls** command will look for two separate files or directories called "My" and "Programs". In this case, we must use quotes to bind the parts of the directory name together into a single argument. Either single or double quotes are accepted by all common Unix shells. The difference between single and double quotes is covered in Chapter 4.

```
shell-prompt: ls 'My Programs'
shell-prompt: ls "My Programs"
```

As an alternative to using quotes, we can *escape* the space by preceding it with a backslash ('\') character. This will save one keystroke if there is only one character to be escaped in the text.

```
shell-prompt: ls My\ Programs
```

**Example 3.5** Practice Break

Try the following commands:

```
shell-prompt: ls
shell-prompt: ls -al
shell-prompt: ls /etc
shell-prompt: ls -al /etc
shell-prompt: mkdir My Programs
shell-prompt: ls
shell-prompt: rmdir My
shell-prompt: rmdir Programs
shell-prompt: mkdir 'My Programs'
shell-prompt: ls
shell-prompt: ls My Programs
shell-prompt: ls "My Programs"
```

### 3.6.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the three major components of a Unix command?
2. What are the two sources of the command name?
3. How do we know whether an argument is a flag or data?
4. What is the advantage of short flags and the advantage of long flags?
5. What do data argument represent?
6. What rules does Unix enforce regarding the order of arguments?
7. What separates one Unix argument from the next?
8. Can an argument contain whitespace? If so, how?

## 3.7 Basic Shell Tools

### 3.7.1 Common Unix Shells

There are many different shells available for Unix systems. This might sound daunting if you're new to Unix, but fortunately, like most Unix tools, all the common shells adhere to certain standards. All of the common shells are derived from one of two early ancestors:

- Bourne shell (sh) is the de facto basic shell on all Unix systems, and is derived from the original Unix shell developed at AT&T.
- C shell (csh) offers mostly the same features as Bourne shell, but the two differ in the syntax of their scripting languages, which are discussed in Chapter 4. The C shell syntax is designed to be more intuitive and similar to the C language.

Most Unix commands are exactly the same regardless of which shell you are using. Differences will only become apparent when using more advanced command features or writing shell scripts, both of which we will cover later.

Common shells derived from Bourne shell include the following:

- Almquist shell (ash), used as the Bourne shell on some BSD systems.
- Korn shell (ksh), an extended Bourne shell with many added features for user-friendliness.
- Bourne again shell (bash) another extended Bourne shell from the GNU project with many added features for user-friendliness. Used as the Bourne shell on some Linux systems.
- Debian Almquist shell (dash), a reincarnation of ash which is used as the Bourne shell on Debian based Linux systems.

Common shells derived from C shell include the following:

- T shell (tcsh), and extended C shell with many added features for user-friendliness.
- Hamilton C shell, an extended C shell used primarily on Microsoft Windows.

Unix systems differ in which shells are included in the base installation, but most shells can be easily added to any Unix system using the system's package manager.

### 3.7.2 Command History

Most shells remember a configurable number of recent commands. This command history is saved both in memory and to disk, so that you can still recall this session's commands next time you log in. The exact mechanisms for recalling those commands varies from shell to shell, but some of the features common to all shells are described below.

Most modern shells support scrolling through recent commands by using the up-arrow and down-arrow keys. Only very early shells lack this capability.

**Note** This feature may not work if your TERM variable is not set properly, since the arrow keys send magic sequences that may differ among terminal types.

The **history** command lists the commands that the shell currently has in memory.

```
shell-prompt: history
```

A command consisting of an exclamation point (!) followed by any character string causes the shell to search for the most recently executed command that began with that string. This is particularly useful when you want to repeat a complicated command.

```
shell-prompt: find Programs -name '*.o' -exec rm -i '{}' \;
shell-prompt: !find
```

An exclamation point followed by a number runs the command with that history index:

```
shell-prompt: history
   385  13;42   more output.txt
   386  13:54   ls
   387  13:54   cat /etc/hosts
shell-prompt: !386
ls
Avi-admin/            Materials-Studio/     iperf-bsd
Backup@               New-cluster/          notes
Books/                Peregrine-admin/      octave-workspace
```

Tantalizing sneak preview: We can check the history for a particular pattern such as "find" as follows:

```
shell-prompt: history | grep find
```

More on the "| find" in Section 3.13.

### 3.7.3 Auto-completion

In most Unix shells, you need only type enough of a command or argument filename to uniquely identify it. At that point, pressing the TAB key will automatically fill in the rest for you. Try the following:

```
shell-prompt: touch sample.txt
shell-prompt: cat sam<Press the TAB key now>
```

If there are other files in your directory that begin with "sam", you may need to type a few additional characters before the TAB, like 'p' and 'l' before auto-completion will work.

### 3.7.4 Command-line Editing

Modern shells allow extensive editing of the command currently being entered. The key bindings for different editing features depend on the shell you are using and the current settings. Some shells offer a selection of different key bindings that correspond to Unix editors such as **vi** or **Emacs**.

See the documentation for your shell for full details. Below are some examples of default key bindings for shells such as **bash** and **tcsh**.

| **Key** | **Action** |
|---|---|
| Left arrow | Move left |
| Right arrow | Move right |
| Ctrl+a | Beginning of line |
| Ctrl+e | End of line |
| Backspace or Ctrl+h | Delete left |
| Ctrl+d | Delete current |

Table 3.3: Default Key Bindings in some Shells

### 3.7.5 Globbing (File Specifications)

There is often a need to specify a large number of files as command line arguments. Typing all of them would be tedious, so Unix shells provide a mechanism called *globbing* that allows short, simple patterns to match many file names. This allows us to type a brief specification that represents a large number (a glob) of files.

| Symbol | Matches |
|---|---|
| * | Any sequence of characters (including none) except a '.' in the first character of the filename. |
| ? | Any single character, except a '.' in the first character of the filename. |
| [string] | Any character in string |
| [c1-c2] | Any character from c1 to c2, inclusive |
| {thing1,thing2} | Thing1 or thing2 |

Table 3.4: Globbing Symbols

These patterns are built from literal text and/or special symbols called *wild cards* as shown in Table 3.4.

Normally, the shell handles these special characters, expanding globbing patterns to a list of matching file names *before* the command is executed.

If you want an argument containing special globbing characters to be sent to a command in its raw form, it must be enclosed in quotes, or each special character must be escaped (preceded by a backslash, \).

Certain commands, such as **find** need to receive the pattern as an argument and attempt to do the matching themselves rather than have it done for them by the shell. Therefore, patterns to be used by the **find** command must be enclosed in quotes.

```
shell-prompt: ls *.txt     # Lists all files ending in ".txt"
shell-prompt: ls "*.txt"   # Fails, unless there is a file called '*.txt'
shell-prompt: ls '*.txt'   # Fails, unless there is a file called '*.txt'
shell-prompt: ls .*.txt    # Lists hidden files ending in ".txt"
shell-prompt: ls [A-Za-z]* # Lists all files and directories
                           # whose name begins with a letter
shell-prompt: find . -name *.txt    # Fails
shell-prompt: find . -name '*.txt'  # List .txt files in all subdirectories
shell-prompt: ls *.{c,c++,f90}
```

**Caution** The exact behavior of character ranges such as [A-Z] may be affected by locale environment variables such as LANG, LC_COLLATE, and LC_ALL. See Section 3.16 for general information about the environment. Information about locale and collation can be found online. Behavior depends on these settings as well as which Unix shell you are using and the shell's configuration settings. Setting LANG and the LC_ variables to C or C.UTF-8 will usually ensure the behavior described above.

### 3.7.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the de facto standard shell on Unix systems?
2. How do most Unix commands differ when run under one shell such as C shell as opposed to running under another such as Bourne shell or Bourne again shell?
3. What if a shell we like or need is not present on our Unix installation?
4. How can we quickly rerun the previous command in most Unix shells?
5. Show a Unix command that lists all recent commands executed from this shell.
6. Show a Unix command that runs the last command that began with "ls".

7. Given the shell history shown below, show a Unix command that runs the last command used to log into unixdev1.

```
    984  16:48   vi .ssh/known_hosts
    985  16:50   ssh -X bacon@unixdev1.ceas.uwm.edu
    986  16:52   ssh -X -C bacon@unixdev1.ceas.uwm.edu
    987  16:58   ape
    988  16:59   ssh -X -C bacon@unixdev1.ceas.uwm.edu
```

8. How can we avoid typing a long file name or command name in most shells?
9. How can we instantly move to the beginning of the command we are currently typing?
10. How do we list all the non-hidden files in the current directory ending in ".txt"?
11. How do we list all the hidden files in the current directory ending in ".txt"?
12. How do we list all the files in /etc beginning with "hosts"?
13. How do we list all the files in the current directory starting with a lower case letter and ending in ".txt"?
14. How do we list all the files in the current directory starting with any letter and ending in ".txt"?
15. How do we list all the non-hidden files in the current directory ending with ".pdf" or ".txt"?
16. How do we list all the files in /etc and all other directories under /etc with names ending in ".conf"?
17. When are globbing patterns normally expanded to a list of files?
18. How can we include a file name in a command if the file name contains a special character such as '*' or '['?

## 3.8 Processes

A program is a file containing statements or commands in the form of source code or machine code. A *process*, in Unix terminology, is the *execution of a program*. By this we mean the running of a program, not a blindfold, a cigarette, and firing squad. A program is an object. A process is an action utilizing that object. A grocery list is like a program. A trip to the grocery store to buy what is on the list is like a process.

Unix is a multitasking system, which means that many processes can be running at any given moment, i.e. there can be many active processes.

When you log in, the system creates a new process to run your shell program. The same happens when other people log in. Hence, while everyone may be using the same shell *program*, they are all running different shell *processes*.

When you run a program (a command) from the shell, the shell creates a new process to run the program. Hence, you now have two processes running: the shell process and the command's process. The shell then normally waits for that child process to complete before printing the shell prompt again and accepting another command.

The process created by the shell to run your command is called a *child process* of the shell process. Naturally, the shell process is then called the *parent process* of the command process.

Each process is uniquely identified by an integer serial number called the *process ID*, or *PID*.

Unix systems also keep track of each process's status and resource usage, such as memory, CPU time, etc. Information about your currently running processes can be viewed using the **ps** (process status) command:

```
shell-prompt: ps
 PID TTY           TIME CMD
7147 ttys000    0:00.14 -tcsh
7438 ttys000    0:01.13 ape notes.dbk unix.dbk
7736 ttys001    0:00.13 -tcsh
```

**Example 3.6** Practice Break

Run the **ps** command. What processes do you have running?

```
shell-prompt: ps
```

What if we want to see all the processes on the system, instead of just our own? On most systems, we can add the `-a` (include other peoples' processes) and `-x` (include processes not started from a terminal) flags.

```
shell-prompt: ps -ax
```

Another useful tool is the **top** command, which monitors all processes in a system and displays system statistics and the top (most active) processes every few seconds. Note that since **top** is a full-terminal command, it will not function properly unless the TERM environment variable is set correctly.

**Example 3.7** Practice Break

Run the **top** command. What processes are using the most CPU time? Type 'q' to quit **top**.

```
shell-prompt: top
```

### 3.8.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does a process differ from a program?
2. If 10 people are logged in and using the same Unix shell, how many shell programs are there? How many shell processes?
3. What normally happens when you run a program from the shell?
4. How are processes identified in Unix?
5. Show a Unix command that lists all the processes currently running on the system.
6. Show a Unix command that monitors which processes are using the most CPU and memory resources.

## 3.9 The Unix File System

### 3.9.1 Unix Files

A Unix *file* is simply a sequence of bytes (8-bit values) stored on a disk and given a unique name. The bytes in a file may be printable characters such as letters, digits, punctuation symbols, invisible *control characters* (which cause a printer or terminal to perform actions such as backspacing or scrolling), part of a number (a typical integer or floating point number consists of 8 bytes), or other non-character, non-numeric data.

This is how Unix sees all files. It takes no interest whatsoever in the meaning of the bytes within a file. The meaning of the content is determined solely by the programs using the file.

#### Text vs Binary Files

Files are often classified as either *text* or *binary* files. All of the bytes in a text file are interpreted as ASCII/ISO characters by the programs that read or write the file, while binary files may contain both character and non-character data.

Again, Unix does not make a distinction between text and binary files. This is left to the programs that use the files.

**Example 3.8** Practice Break

Try the following commands:

```
shell-prompt: cat /etc/hosts
```

What do you see? The `/etc/hosts` file is a text file, and **cat** is used here to echo (concatenate) it to the terminal output.

Now try the following:

```
shell-prompt: cat /bin/ls
```

What do you see? The file `/bin/ls` is not a text file. It contains binary program code, not characters. The **cat** command assumes that the file is a text file and sends each byte to your terminal. The terminal tries to interpret each byte as an ASCII/ISO character and display it on the screen. Since the file does not contain a sequence of characters, it appears as nonsense on your terminal. Some of the bytes sent to the terminal may even knock it out of whack, causing it to behave strangely. If this happens, run the **reset** command to restore your terminal to its default state.

#### Unix vs. Windows Text Files

While it is the program that interprets the contents of a file, there are some conventions regarding text file format that all Unix programs follow, so that they can all manipulate the same files. Unfortunately, Windows programs follow different conventions. Unix programs assume that text files terminate each line with a control character known as a *line feed* (also known as a *newline* or *NL* for short), which is the 10th character in the standard ASCII/ISO character sets. Windows programs use both a *carriage return* or *CR* (13th character) and NL.

Text files created on Windows will contain both a CR and NL at the end of each line. Text files created on Unix will have only an NL. This can cause problems for programs on either Unix or Windows. Hence, it is not a good idea to use a Windows editor to write code for Unix systems or vice-versa.

The **dos2unix** and **unix2dos** commands can be used to clean up files that have been transferred between Unix and Windows. These programs convert text files between the Windows and Unix standards. If you've edited a text file on a non-Unix system, and are now using it on a Unix system, you can clean it up by running:

```
shell-prompt: dos2unix filename
```

The **dos2unix** and **unix2dos** commands are not standard with most Unix systems, but they are free programs that can easily be added via most package managers.

**Caution** Note that **dos2unix** and **unix2dos** should only be used on text files. They should never be used on binary files, since the contents of a binary file are not meant to be interpreted as characters such as line feeds and carriage returns.

### 3.9.2 File system Organization

#### Basic Concepts

A Unix file system contains *files* and *directories*. A file is like a document, and a directory is like a folder that contains documents and/or other directories. The terms "directory" and "folder" are interchangeable, but "directory" is the standard term used in Unix.

Directories are so called because they serve the same purpose as the directory you might find in the lobby of an office building: They are listings that keep track of what files and other directories are called and where they are located on the disk.

**Note**
Unix file systems use case-sensitive file and directory names. I.e., `Temp` is not the same as `temp`, and both can coexist in the same directory.
macOS is the only mainstream Unix system that violates this convention. The standard OS X file systems is case-preserving, but not case-sensitive. This means that if you call a file `Temp`, it will remember that the T is capital, but it can also be referred to as `temp`, `tEmp`, etc. Only one of these files can exist in a given directory at any one time.

A Unix file system can be visualized as a tree, with each file and directory contained within another directory. Figure 3.2 shows a small portion of a typical Unix file system. On a real Unix system, there are usually thousands of files and directories. Directories are shown in green and files are in yellow.

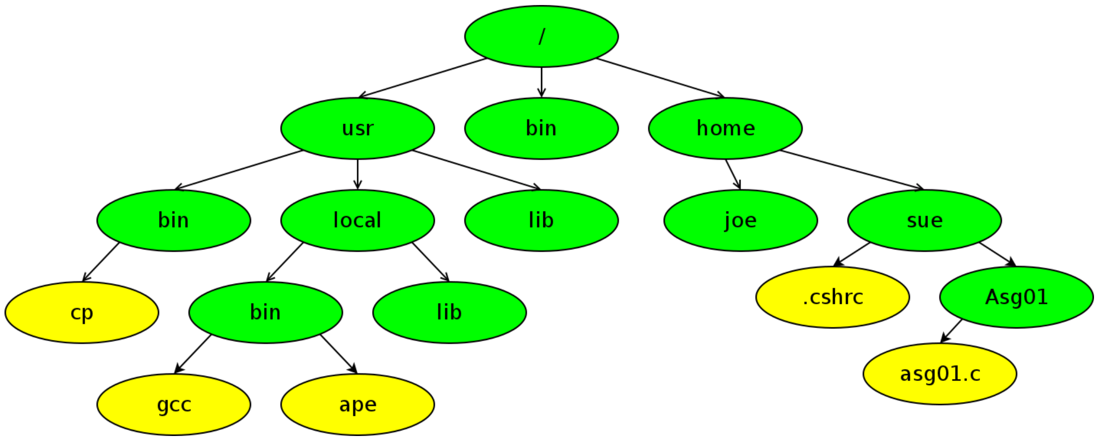


Figure 3.2: Sample of a Unix File system

Unix uses a forward slash (/) to separate directory and file names while Windows uses a backslash (\).

The one directory that is not contained within any other is known as the *root directory*, whose name under Unix is `/`. There is exactly one root directory on every Unix system. Windows systems, on the other hand, have a root directory for each disk partition such as C:\ and D:\.

The Cygwin compatibility layer works around the separate drive letters of Windows by unifying them under a common parent directory called /cygdrive. Hence, for Unix commands run under Cygwin, /cygdrive/c is equivalent to c:\, /cygdrive/d is equivalent to d:\, and so on. This allows Cygwin users to do things like search multiple Windows drive letters with a single command starting in /cygdrive.

Unix file system trees are fairly standardized, but most have some variation. For instance, all Unix systems have a /bin and a /usr/bin, which contain standard Unix commands. Not all of them have /home or /usr/local. Many Linux systems install commands from add-on packages into /usr/bin, mixing them with the standard Unix commands that are essential to the basic functioning of the system. Other systems such as most BSDs keep them separated in /usr/local/bin or /usr/pkg/bin.

The root directory is the *parent* of /bin and /home and an *ancestor* of all other files and directories.

The /bin and /home directories are *subdirectories*, or *children* of /. Likewise, /home/joe and /home/sue are subdirectories of /home, and grandchildren of /.

All of the files in and under /home comprise a *subtree* of /home.

The children of a directory, all of its children, and so on, are known as *descendants* of the directory. All files and directories on a Unix system, except /, are descendants of /.

Each user has a *home directory*, which can be arbitrarily assigned, but is generally a child of /home on many Unix systems or of /Users on macOS. Most or all of a user's files and subdirectories are found under their home directory. In the example above, /home/joe is the home directory for user joe, and /home/sue is the home directory for user sue.

In some situations, a home directory can be referred to as ~ or ~user. For example, user joe can refer to his home directory as ~, ~/, or ~joe, while he can only refer to sue's home directory as ~sue.

#### Absolute Path Names

The *absolute path name*, also known as *full path name*, of a file or directory denotes the complete path from / (the root directory) to the file or directory of interest. It is the path we would "walk" from the root directory (/) to the file or directory of interest. For example, the absolute path name of Sue's .cshrc file is /home/sue/.cshrc, and the absolute path name of the ape command is /usr/local/bin/ape. To walk the directory tree, we would start in / and progress from there:

```
Start in /
Go to    /usr
Go to    /usr/local
Go to    /usr/local/bin
End at   /usr/local/bin/ape
```

The absolute path name is the only way to uniquely identify a file or directory in the file system.

**Note** An absolute path name always begins with '/' or a '~', noting that '~' is shorthand for a path that begins with a '/' such as /home/joe or /Users/joe.

**Example 3.9** Practice Break

Try the following commands:

```
shell-prompt: ls
shell-prompt: ls /etc
shell-prompt: cat /etc/hosts
shell-prompt: ls ~
```

#### Current Working Directory

Every Unix *process* has an attribute called the *current working directory*, or *CWD*. This is the directory that the process is currently "in". When you first log into a Unix system, the shell process's CWD is set to your home directory.

**Note** It is important to understand that the CWD is a property of each *process*, not of a user or a program.

The **pwd** command prints the CWD of the shell process. The **cd** command changes the CWD of the shell process. Running **cd** with no arguments sets the CWD to your home directory, much like clicking your heels together three times to get back to Kansas.

**Example 3.10** Practice Break

Try the following commands:

```
shell-prompt: pwd
shell-prompt: cd /
shell-prompt: pwd
shell-prompt: cd
shell-prompt: pwd
```

Many commands, such as **ls**, use the CWD as a default if you don't provide a directory name on the command line. For example, if the CWD is `/home/joe`, then the following commands are the same:

```
shell-prompt: ls
shell-prompt: ls /home/joe
shell-prompt: ls ~joe
```

### Relative Path Names

Whereas an absolute path name denotes the path from / to a file or directory, the *relative path name* denotes the path from the CWD to a file or directory.

Any path name that does not begin with a '/' or '~' is interpreted as a relative path name. The absolute path name is then derived by appending the relative path name to the CWD. For example, if the CWD is `/etc`, then the relative path name `hosts` refers to the absolute path name `/etc/hosts`, and the relative path name of `/etc/ssh/ssh_config` is `ssh/ssh_config`.

absolute path name = CWD + "/" + relative path name

**Note** Since the CWD is a property of each process, a relative path name is not the same for all processes. Relative path names for the same file may be different for different processes, or for the same process before and after it changes its CWD. For example the meaning of the relative path name `bin` is `/bin` when CWD is / and `/usr/bin` when CWD is `/usr`.

**Note** Relative path names are handled at the lowest level of the operating system, by the Unix kernel. This means that they can be used anywhere: in shell commands, in C or Fortran programs, etc.

When you run a program from the shell, the new process inherits the CWD from the shell. Hence, you can use relative path names as arguments in any Unix command, and they will use the CWD inherited from the shell process. For example, the two **cat** commands below have the same effect.

```
shell-prompt: cd /etc        # Set shell's CWD to /etc
shell-prompt: cat hosts      # Inherits CWD from shell, so hosts = /etc/hosts
shell-prompt: cat /etc/hosts # Same effect as above
```

**Wasting Time**

The **cd** command is one of the most overused Unix commands. Many people use it where it is completely unnecessary and actually results in significantly more typing than needed. Don't use **cd** if it is actually more work than using an absolute path name as an argument. For example, consider the sequence of commands:

```
shell-prompt: cd /etc
shell-prompt: more hosts
shell-prompt: cd
```

The same effect could have been achieved much more easily using the following single command:

```
shell-prompt: more /etc/hosts
```

**Note** In almost all cases, absolute path names and relative path names are interchangeable. You can use either type of path name as a command line argument, or within a program.

**Example 3.11** Practice Break

Try to predict the results of the following commands before running them:

```
shell-prompt: cd
shell-prompt: pwd
shell-prompt: cd /etc
shell-prompt: pwd
shell-prompt: cat hosts
shell-prompt: cat /etc/hosts
shell-prompt: cd
shell-prompt: pwd
shell-prompt: cat hosts
```

Why does the last command result in an error?

### Avoid Absolute Path Names

The relative path name is potentially much shorter than the equivalent absolute path name. Using relative path names also makes code more portable.

Suppose you have a project contained in the directory `/Users/joe/Thesis` on your Mac. Now suppose you want to work on the same project on an HPC cluster, where there is no `/Users` directory, and you have to store it in `/share1/joe/Thesis`.

The absolute path name of every file and directory under `Thesis` will be different on the cluster than it is on your Mac. This can cause major problems if you were using absolute path names in your scripts, programs, and makefiles. Statements like the following will have to be changed in order to run the program on a different computer.

```
infile = fopen("/Users/joe/Thesis/Inputs/input1.txt", "r");
```

```
sort /Users/joe/Thesis/Inputs/names.txt
```

**Note** No program should ever have to be altered just to make it run on a different computer. Changes like these are a source of *regressions* (new program bugs).

While the absolute path names change when you move the Thesis directory, the path names relative to the `Thesis` directory remain the same. For this reason, absolute path names should be avoided.

The statements below will work on any computer as long as the program or script is running with `Thesis` as the CWD. It does not matter where the `Thesis` directory is located, so long as the `Inputs` directory is its child.

```
infile = fopen("Inputs/input1.txt", "r");
```

```
sort Inputs/names.txt
```

### Special Directory Names

In addition to absolute path names and relative path names, there are a few special symbols for directories that are commonly referenced:

| Symbol | Refers to |
|---|---|
| . | The current working directory |
| .. | The parent of the current working directory |
| ~ | Your home directory |
| ~user | user's home directory |

Table 3.5: Special Directory Symbols

The '.' notation for CWD is useful for copying files to CWD and other commands that require a target directory name.

```
shell-prompt: cp /etc/hosts .
```

It is also useful if a mishap occurs, leading to the creation of a file whose name begins with a special character such as '-' or '~'. If we have a file called "-file.txt", we cannot remove it with **rm -file.txt**, since the **rm** command will think the '-' indicates a flag argument. To get around this, we simply need to make the argument not begin with a '-'. We can either use the absolute path name of the file, e.g. `/home/joe/-file.txt` or `./-file.txt`. `./path` is exactly the same as `path`.

The ".." notation refers to the parent of the CWD and allows for relative path names that are not *under* the CWD. For example, if the CWD is `/home/joe`, then the relative path of `/home/sue/.cshrc` is `../sue/.cshrc` and the relative path name of `/etc/hosts` is `../../etc/hosts`. We can "walk" a relative path such as `../../etc/hosts` just as we walk an absolute path:

```
Start at /home/joe       (.)
Go to    /home           (..)
Go to    /               (../..)
Go to    /etc            (../../etc)
End at   /etc/hosts      (../../etc/hosts)
```

Note that `/home/joe/../sue/.cshrc` (`/home/joe + / + ../sue/.cshrc`) is a valid absolute path name, but it can be shortened to `/home/sue/.cshrc`. We can always remove a `../` along with the path component to the left of it, such as `joe/../`. Likewise, `/home/joe/../../etc/hosts` can be reduced to `/home/../etc/hosts` and further to `/etc/hosts`.

**Example 3.12** Practice Break

Try the following commands and see what they do:

```
shell-prompt: cd
shell-prompt: pwd
shell-prompt: ls
shell-prompt: ls ~
shell-prompt: ls .
shell-prompt: mkdir Data Scripts
shell-prompt: cp /etc/hosts .
shell-prompt: mv hosts Data
shell-prompt: ls Data
shell-prompt: ls ./Data
shell-prompt: cd Data
shell-prompt: cd ../Scripts
shell-prompt: ls ..
shell-prompt: ls ../Data
shell-prompt: more ../Data/hosts
shell-prompt: rm ../Data/hosts
shell-prompt: ls ~/Data
shell-prompt: ls /bin
shell-prompt: cd ..
shell-prompt: pwd
```

### 3.9.3 Ownership and Permissions

#### Overview

Every file and directory on a Unix system has inherent access control features based on a simple system:

- Every file and directory belongs to an individual user and to a group of users.
- There are 3 types of permissions which are controlled separately from each other:
  - Read
  - Write (modify)
  - Execute (e.g. run a file if it's a program)
- Read, write, and execute permissions can be granted or denied separately for each of the following:
  - The individual who owns the file (user)
  - The group that owns the file (group)
  - All other users on the system (a hypothetical group known as "world" (other)

Execute permissions on a file mean that the file can be executed as a script or a program by typing its name. It does not mean that the file actually contains a script or a program: It is up to the owner of the file to set the execute permissions appropriately for each file.

Execute permissions on a directory mean that permitted users can **cd** into it. Users only need read permissions on a directory to list it or access a file within it, but they need execute permissions in order for their processes to make it the CWD.

Unix systems provide this access using 9 on/off switches (bits) associated with each file.

#### Viewing Permissions

If you do a long listing of a file or directory, you will see the ownership and permissions:

```
shell-prompt: ls -l
drwx------   2 joe    users      512 Aug  7 07:52 Desktop/
drwxr-x---  39 joe    users     1536 Aug  9 22:21 Documents/
drwxr-xr-x   2 joe    users      512 Aug  9 22:25 Downloads/
-rw-r--r--   1 joe    users    82118 Aug  2 09:47 bootcamp.pdf
```

The leftmost column shows the type of object and the permissions for each user category.

A '-' in the leftmost character means a regular file, 'd' means a directory, 'l' means a link. etc. Running **man ls** will reveal all the codes.

The next three characters are, in order, read, write and execute permissions for the owner (joe).

The next three after that are permissions for members of the owning group (users).

The next three are permissions for world (other).

A '-' in a permission bit column means that the permission is denied for that user or set of users and an 'r', 'w', or 'x' means that read, write, or execute is permitted.

The next three columns show the number of links (different path names for the same file), the individual and group ownership of the file or directory. The remaining columns show the size, the date and time it was last modified, and name. In addition to the 'd' in the first column, directory names are followed by a '/' if the **ls** is so configured.

You can see above that Joe's `Desktop` directory is readable, writable, and executable for Joe, and completely inaccessible to everyone else.

Joe's `Documents` directory is readable, writable and executable for Joe, and readable and executable for members of the group "users". Users not in the group "users" cannot access the Documents directory at all.

Joe's `Downloads` directory is readable and executable to anyone who can log into the system.

The file `bootcamp.pdf` is readable by group and world, but only writable by Joe. It is not executable by anyone, which makes sense because a PDF file is not a program.

#### Setting Permissions

Users cannot change individual ownership on a file, since this would allow them to subvert disk quotas and do other malicious acts by placing their files under someone else's name. Only the *superuser* (the system administrator) can change the individual ownership of a file or directory.

Every user has a primary group and may also be a member of supplementary groups. Users can change the group ownership of a file to any group that they belong to using the **chgrp** command, which requires a group name as the second argument and one or more path names following the group:

```
shell-prompt: chgrp group path [path ...]
```

All sharing of files on Unix systems is done by controlling group ownership and file permissions.

File permissions are changed using the **chmod** command:

```
shell-prompt: chmod permission-specification path [path ...]
```

The permission specification has a symbolic form, and a raw form, which is an octal number.

The symbolic form consists of any of the three user categories 'u' (user/owner), 'g' (group), and 'o' (other/world) followed by a '+' (grant) or '-' (revoke), and finally one of the three permissions 'r', 'w', or 'x'.

To add read and execute (cd) permissions for group and world on the Documents directory:

```
shell-prompt: chmod go+rx Documents
```

Sometimes it is impossible to express the changes we want to make in one simple specification. In that case, we can use a compound specification, two or more basic specs separated by commas. Remember that white space indicates the end of an argument, so we cannot have any white space next to the comma.

To revoke all permissions for world on the Documents directory and grant read permission for the group:

```
shell-prompt: chmod o-rwx,g+r Documents
```

Disable write permission for everyone, including the owner, on bootcamp.pdf. This can be used to prevent the owner from accidentally deleting an important file.

```
shell-prompt: chmod ugo-w bootcamp.pdf
```

Run **man chmod** for additional information.

The raw form for permissions uses a 3-digit octal number to represent the 9 permission bits. This is a quick and convenient method for computer nerds who can do octal/binary conversions in their head.

```
shell-prompt: chmod 644 bootcamp.pdf   # 644 = 110100100 = rw-r--r--
shell-prompt: chmod 750 Documents      # 750 = 111101000 = rwxr-x---
```

**Caution** NEVER make any file or directory world-writable. Doing so allows any other user to modify it, which is a serious security risk. A malicious user could use this to install a Trojan Horse program under your name, for example.

By default, new files you create are owned by you and your primary group. If you are a member of more than one group and wish to share a directory with one of your supplementary groups, it may also be helpful to set a special flag on the directory so that new files created in it will have the same group as the directory, rather than your primary group. Then you won't have to remember to chmod every new file you create.

```
shell-prompt: chmod g+s Shared-research
```

**Example 3.13** Practice Break

Try the following commands, and try to predict the output of each **ls** before you run it.

```
shell-prompt: touch testfile
shell-prompt: ls -l
shell-prompt: chmod go-rwx testfile
shell-prompt: ls -l
shell-prompt: chmod o+rw testfile
shell-prompt: ls -l
shell-prompt: chmod g+rwx testfile
shell-prompt: ls -l
shell-prompt: rm testfile
```

Now set permissions on testfile so that it is readable, writable, and executable by you, only readable by the group, and inaccessible to everyone else.

### 3.9.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a file in the viewpoint of Unix?
2. What is the difference between a text file and a binary file?
3. What will happen if you echo a binary file to your terminal?
4. What is the difference between Windows and Unix text files?
5. How can we convert text files between the Unix and Windows standards?
6. What is a directory?
7. What does it mean that Unix filenames are case-sensitive?
8. What is a root directory?
9. How many root directories does a Unix system have? How many does Windows have?
10. What is contained in the /bin and /usr/bin directories?
11. What is a subdirectory?
12. What is a home directory?
13. What is an absolute path name and how do we recognize one?
14. What is the absolute path name of Sue's asg01.c in the tree diagram in this section?
15. Of what is the CWD a property?
16. Show a Unix command that prints the CWD of a shell process.
17. Show a Unix command that sets the CWD of a shell process to /tmp.
18. Show a Unix command that sets the CWD of a shell process to our home directory?
19. What is a relative path name and how to we recognize one?
20. Is a relative path name unique? Prove your answer with an example.
21. How does Unix determine the absolute path name from a relative path name?
22. If the CWD of a process is /usr/local, what is the absolute path name of "bin/ape"?
23. If the CWD of a process is /usr/local, what is the relative path name of /usr/local/lib/libxtend.a?
24. If the CWD of a process is /usr/local, what is the relative path name of /usr/bin?
25. If the CWD of a process is /usr/local, what is the relative path name of /etc/motd?
26. Where does a new process get its initial CWD?
27. Why should we avoid using absolute path names in programs and scripts?
28. Show a Unix command that lists the contents of the parent directory of CWD.
29. If the CWD of a process is /home/bob/Programs, what is the relative path name of /home/bob/Data/input1.txt?
30. How do we remove a file called "~sue" in the CWD?
31. What are the three user categories that can be granted permissions on a file or directory?

32. What does it mean to set execute permission on a file? On a directory?
33. Given the following ls -l output, who can do what to bootcamp.pdf?

```
-rw-r-----   1 joe    users    82118 Aug  2 09:47 bootcamp.pdf
```

34. How would we allow users who are not in the owning group to read bootcamp.pdf?
35. How would we allow members of the group to read and execute the program "simulation" and at the same time revoke all access to other users?
36. Show a Unix command that makes the directory "MyScripts" world writable.
37. Show a Unix command that changes the group ownership of the directory "Research" to the group "smithlab".
38. Assuming your primary group is "joe", show a Unix command that configures the directory Research form the previous question so that new files you create in it will be owned by "smithlab" instead of "joe"?

## 3.10 Unix Commands and the Shell

**Before You Begin** You should have a basic understanding of Unix processes, files, and directories. These topics are covered in Section 3.8 and Section 3.9.

Unix commands fall into one of two categories:

- Internal commands are part of the shell.

  No new process is created when you execute an internal command. The shell simply carries out the execution of internal commands by itself.

- External commands are programs separate from the shell. The command name of an external command is actually the name of an *executable file*, i.e. a file containing the program or script. For example, when you run the **ls** command, you are executing the program contained in the file `/bin/ls`.

  When you run an external command, the shell locates the program file, loads the program into memory, and creates a new (child) process to execute the program. The shell then normally waits for the child process to end before prompting you for the next command.

### 3.10.1 Internal Commands

Commands are implemented internally only when it is necessary or when there is a substantial benefit. If all commands were part of the shell, the shell would be enormous and require too much memory.

One command that must be internal is the **cd** command, which changes the CWD of the shell process. The **cd** command cannot be implemented as an external command, since the CWD is a property of the process, as described in Section 3.9.2.

We can prove this using *Proof by Contradiction*. If the **cd** command were external, it would run as a child process of the shell. Hence, running **cd** would create a child process, which would inherit CWD from the shell process, alter its copy of CWD, and then terminate. The CWD of the parent, the shell process, would be unaffected.

Expecting an external command to change your CWD for you would be akin to asking one of your children to go to take a shower for you. Neither is capable of affecting the desired change. Likewise, any command that alters the state of the shell process must be implemented as an internal command.

### 3.10.2 External Commands

Most commands are external, i.e. programs separate from the shell. As a result, they behave the same way regardless of which shell we use to run them.

The executable files containing external commands are kept in certain directories, most of which are called `bin` (short for "binary", since most executable files are binary files containing machine code). The most essential commands required for the Unix system to function are kept in `/bin` and `/usr/bin`. The location of optional add-on commands varies, but a typical location is `/usr/local/bin`. Debian and Redhat Linux mix add-on commands with core system commands in `/usr/bin`. BSD systems keep them separate directories such as `/usr/local/bin` or `/usr/pkg/bin`.

**Example 3.14** Practice Break

1. Use **which** under C shell family shells to find out whether the following commands are internal or external. Use **type** under Bourne family shells (bash, ksh, dash, zsh). You can use either command under either shell, but will get better results if you follow the advice above. (Try both and see what happens.)

```
shell-prompt: which cd
shell-prompt: which cp
shell-prompt: which exit
shell-prompt: which ls
shell-prompt: which pwd
```

2. Use **ls** to find out what commands are located in `/bin` and `/usr/bin`.

### 3.10.3 Getting Help

In the dark ages before Unix, when programmers wanted to look up a command or function, they actually had to get out of their chairs and walk somewhere to pick up a typically ring-bound printed manual to flip through. This resembled physical activity, which most computer scientists find terrifying.

The Unix designers saw the injustice of this situation and set out to rectify it. They imagined a Utopian world where they could sit in the same chair for ten hours straight without ever taking our eyes off the monitor or their fingers off the keyboard, happily subsisting on coffee and potato chips.

**Aside**
If there is one trait that best defines an engineer it is the ability to concentrate on one subject to the complete exclusion of everything else in the environment. This sometimes causes engineers to be pronounced dead prematurely. Some funeral homes in high-tech areas have started checking resumes before processing the bodies. Anybody with a degree in electrical engineering or experience in computer programming is propped up in the lounge for a few days just to see if he or she snaps out of it.
-- The Engineer Identification Test (Anonymous)

And so, online documentation was born. On Unix systems, all common Unix commands are documented in detail on the Unix system itself, and the documentation is accessible via the command line (you do not need a GUI to view it, which is important when using a dumb terminal to access a remote system). Whenever you want to know more about a particular Unix command, you can find out by typing **man command-name**. For example, to learn all about the **ls** command, type:

```
shell-prompt: man ls
```

The **man** covers virtually every common command, as well as other topics. It even covers itself:

```
shell-prompt: man man
```

The **man** command displays a nicely formatted document known as a *man page*. It uses a file viewing program called **more**, which can be used to browse through text files very quickly. Table 3.6 shows the most common keystrokes used to navigate a man page. For complete information on navigation, run:

| **Key** | **Action** |
|---|---|
| h | Show key commands |
| Space bar | Forward one page |
| Enter/Return | Forward one line |
| b | Back one page |
| / | Search |

Table 3.6: Common hot keys in **more**

```
shell-prompt: man more
```

Man pages include a number of standard sections, such as SYNOPSIS, DESCRIPTION, and SEE ALSO, which helps you identify other commands that might be of use.

Man pages do not always make good tutorials. Sometimes they contain too much detail, and they are often not well-written for novice users. If you're learning a new command for the first time, you might want to consult a Unix book or the WEB. The man pages will provide the most detailed and complete reference information on most commands, however.

The **apropos** command is used to search the man page headings for a given topic. It is equivalent to **man -k**. For example, to find out what man pages exist regarding Fortran, we might try the following:

```
shell-prompt: apropos sine
FreeBSD moray.acadix  bacon ~ 1002: apropos sine
acos, acosf, acosl(3) - arc cosine functions
acosh, acoshf, acoshl(3) - inverse hyperbolic cosine functions
asin, asinf, asinl(3) - arc sine functions
asinh, asinhf, asinhl(3) - inverse hyperbolic sine functions
cos, cosf, cosl(3) - cosine functions
cosh, coshf, coshl(3) - hyperbolic cosine functions
cospi, cospif, cospil(3) - half-cycle cosine functions
Role::Tiny(3) - Roles: a nouvelle cuisine portion size slice of Moose
sin, sinf, sinl, sincosl(3) - sine functions
sincos, sincosf, sincosl(3) - sine and cosine functions
sinh, sinhf, sinhl(3) - hyperbolic sine function
sinpi, sinpif, sinpil(3) - half-cycle sine functions
```

or

```
shell-prompt: man -k sine
```

The **whatis** is similar to **apropos** in that it lists short descriptions of commands. However, **whatis** only lists those commands with the search string in their name or short description, whereas **apropos** attempts to list everything related to the string.

```
shell-prompt: whatis sin
sin, sinf, sinl, sincosl(3) - sine functions
```

The **info** command is an alternative to man that uses a non-graphical hypertext system instead of flat files. This allows the user to navigate extensive documentation more efficiently. The **info** command has a fairly high learning curve, but it is very powerful, and is often the best option for documentation on a given topic. Some open source software ships documentation in info format and provides a man page (converted from the info files) that actually has less information in it.

```
shell-prompt: info gcc
```

**Example 3.15** Practice Break

1. Find out how to display a '/' after each directory name and a '*' after each executable file when running **ls**.
2. Use **apropos** to find out what Unix commands to use with bzip files.

### 3.10.4 Some Useful Unix Commands

Most Unix commands have short names which are abbreviations or acronyms for what they do. ( pwd = print working directory, cd = change directory, ls = list, ... ) Unix was originally designed for people with good memories and poor typing skills. Some of the most commonly used Unix commands are described below.

---

**Note** This section is meant to serve as a quick reference, and to inform new readers about which commands they should learn. There is much more to know about these commands than we can cover here. For full details about any of the commands described here, consult the **man** pages, **info** pages, or the WEB.

---

This section uses the same notation conventions as the Unix man pages:

- Optional arguments are shown inside [].
- The 'or' symbol (|) between two items means one or the other.
- An ellipses (...) means optionally more of the same.
- "file" means a filename is required and a directory name is not allowed. "directory" means a directory name is required, and a filename is not allowed. "path" means either a filename or directory name is acceptable.

#### File and Directory Management

---

**Note** Run these commands in the exact order presented. Some depend on successful completion of previous commands.

---

**ls** lists files in CWD or a specified file or directory.

```
shell-prompt: ls [path ...]
```

```
shell-prompt: ls           # List CWD
shell-prompt: ls /etc      # List /etc directory
```

**mkdir** creates one or more directories.

```
shell-prompt: mkdir [-p] path name [path name ...]
```

The `-p` flag indicates that mkdir should attempt to create any parent directories in the path that don't already exist. If not used, **mkdir** will fail unless all but the last component of the path already exist.

```
shell-prompt: ls
shell-prompt: mkdir Temp
shell-prompt: ls                        # Should see Temp now
shell-prompt: mkdir Temp2/C/MPI      # Should fail
shell-prompt: mkdir -p Temp2/C/MPI
shell-prompt: ls Temp2
```

**cp** copies one or more files.

```
shell-prompt: cp source-file destination-file
shell-prompt: cp source-file [source-file ...] destination-directory
```

If there is only one source filename, then destination can be either a filename or a directory.

```
shell-prompt: cd
shell-prompt: touch file            # Create file if it doesn't exist
shell-prompt: cp file file.bak      # Make a backup copy
shell-prompt: ls                    # Should see file and file.bak
```

If there are multiple source files, then destination must be a directory. If destination is a filename, and the file exists, it will be overwritten.

```
shell-prompt: cp /etc/hosts* hosts  # Should fail
shell-prompt: cp /etc/hosts* Temp   # Should work if directory Temp exists
shell-prompt: ls Temp
```

**mv** moves or renames files or directories.

```
shell-prompt: mv source destination
shell-prompt: mv source [source ...] destination-directory
```

```
shell-prompt: mv file.bak file.bk
shell-prompt: ls
```

If multiple sources are given, destination must be a directory.

```
shell-prompt: mv file file.bk file2     # Should fail
shell-prompt: mv file file.bk Temp      # Should work if directory Temp exists
shell-prompt: ls
shell-prompt: ls Temp
```

**rm** removes one or more files.

```
shell-prompt: rm file [file ...]
```

```
shell-prompt: cd Temp
shell-prompt: ls
shell-prompt: rm hosts*
shell-prompt: ls
shell-prompt: rm file*
shell-prompt: ls
```

**Caution** Removing files with **rm** is not like dragging them to the trash. Once files are removed by **rm**, they cannot be recovered.

If there are multiple hard links to a file, removing one of them only removes the link, and remaining links are still valid.

**Caution** Removing the path name to which a symbolic link points will render the symbolic link invalid. It will become a *dangling link*.

**srm** (secure rm) removes files securely, erasing the file content and directory entry so that the file cannot be recovered. Use this to remove files that contain sensitive data. This is not a standard Unix command, but a free program that can be easily installed on most systems via a package manager.

**df** shows the free disk space on all currently mounted partitions.

```
shell-prompt: df
```

**ln** link files or directories.

```
shell-prompt: ln source-file destination-file
shell-prompt: ln -s source destination
```

The **ln** command creates another path name for the same file. Both names refer to the same file, so changes made through one name (e.g. using nano) appear in the other.

Each file in a typical Unix file system is described by a structure called an *inode*. The inode contains *metadata*, i.e. information about the file other than its content, such as the file’s ownership, permissions, last modification time, and the locations of the disk blocks (chunks of disk space) containing the file’s content.

Without `-s`, a standard directory entry, known as a *hard link* is created. A hard link is a directory entry that points directly to the inode of the file. In fact, such a directory entry contains little more than the file’s name and the location of the inode. Every file must have at least one hard link to it. For this reason, removing a file is also known as "unlinking".

```
shell-prompt: touch file
shell-prompt: ln file file.hardlink
shell-prompt: ls -l
```

To create a second hard link, the source cannot be a directory, and the source and destination path names must be in the same file system. There is no harm in trying to create a hard link. If it fails, you can do a soft link instead.

```
shell-prompt: ln /etc .          # Should fail
shell-prompt: ln -s /etc .
shell-prompt: ls
shell-prompt: ls etc             # List the link
shell-prompt: ls etc/            # List contents of the directory
```

File systems under Windows appear as different drive letters, such as C: or D:. Under Unix, each file system is *mounted* to a specific directory. The main file system is mounted to `/` and the rest are mounted to subdirectories. The **df** command will list file systems and their mount points within the directory tree. For example, in the **df** output below, `/` and `/data` are separate file systems. The disk `ada0` is divided into three *partitions*. Partition 2, called `ada0p2`, contains a file system which is mounted on `/`. Partitions 0 and 1 are used by the operating system for other purposes. The second disk, `ada1`, has a file system on partition 0, which is mounted on `/data`.

```
shell-prompt: df
Filesystem          Size    Used   Avail Capacity  Mounted on
/dev/ada0p2         447G    266G    146G    64%    /
/dev/ada1p0         978G    172G    729G    20%    /data
```

Everything under `/data` and only things under `/data` are on `ada1p0`. Hence, we cannot create a hard line to `/data/joe/Research/notes.txt` in `/home/joe`, which is on `ada0p2`.

```
# This will fail.
# You cannot run this command, since the partitions are hypothetical
# You can try linking something from a different filesystem based on
# your own "df" output if you like.
shell-prompt: ln /data/joe/Research/notes.txt ~joe
```

With `-s`, a *symbolic link*, or *soft link* is created. A symbolic link is not a standard directory entry, but a pointer to another path name. It is a directory entry that points to another directory entry rather than the inode of the file. Symbolic links to not have to be in the same file system as the source.

```
# This will work
shell-prompt: ln -s /data/joe/Research/notes.txt ~joe
```

**rmdir** removes one or more empty directories.

```
shell-prompt: rmdir directory [directory ...]
```

**rmdir** will fail if a directory is not completely empty. You may also need to check for hidden files using **ls -a directory**. To remove a directory and everything under it, use **rm -r directory**.

```
shell-prompt: cd
shell-prompt: rmdir Temp2                # Should fail
shell-prompt: rmdir Temp2/C
```

```
shell-prompt: rmdir Temp2/C/MPI
shell-prompt: rm -r Temp2
shell-prompt: rmdir Temp                # Should tail
shell-prompt: rm -r Temp
shell-prompt: ls
```

**du** reports the disk usage of a directory and everything under it.

```
shell-prompt: du [-s] [-h] path
```

The `-s` (summary) flag suppresses output about each file in the subtree, so that only the total disk usage of the directory is shown. The `-h` asks for human-readable output with gigabytes followed by a G, megabytes by an M, etc.

```
shell-prompt: du -sh /etc
```

**Note**

The **du** command does *not* add up file content sizes. It adds up the disk space used by each file. In an uncompressed file system, space used is rounded up to a multiple of the block size (commonly 4096 bytes). In a compressed file system, space used is a multiple of blocks used after compression, which can be significantly smaller than the file content. This is often the case with the ZFS file system, which is standard on FreeBSD and Solaris-based systems such as OpenIndiana. "Fluffy" text files that compress easily, such as genomic data, may require only a small fraction of their content size in disk space on ZFS. This make ZFS a great choice for housing genomic data.

#### Shell Internal Commands

As mentioned previously, internal commands are part of the shell, and serve to control the shell itself. Below are some of the most common internal commands.

**cd** changes the current working directory of the shell process.

```
shell-prompt: cd [directory]
```

**pushd** changes CWD and saves the old CWD on a stack so that we can easily return.

```
shell-prompt: pushd directory
```

Users often encounter the need to temporarily go to another directory, run a few commands, and then come back to the current directory.

The **pushd** command is a very useful alternative to **cd** that helps in this situation. It performs the same operation as **cd**, but it records the starting CWD by adding it to the top of a stack of CWDs. You can then return to where the last **pushd** command was invoked using **popd**. This saves you from having to retype the path name of the directory to which you want to return. This is like leaving a trail of bread crumbs in the woods to retrace your path back home, except the pushd stack will not get eaten by birds and squirrels, and you won't end up in a witch's soup pot.

**Example 3.16** Practice Break

Try the following sequence of commands:

```
shell-prompt: pwd          # Check starting point
shell-prompt: pushd /etc
shell-prompt: more hosts
shell-prompt: pushd /home
shell-prompt: ls
shell-prompt: popd         # Back to /etc
shell-prompt: pwd
shell-prompt: more hosts
shell-prompt: popd         # Back to starting point
shell-prompt: pwd
```

**exit** terminates the shell process.

```
shell-prompt: exit
```

This is the most reliable way to exit a shell. In some situations you could also type **logout** or simply press Ctrl+d, which sends an EOT character (end of transmission, ASCII/ISO character 4) to the shell.

### Simple Text File Processing

**cat** echoes the contents of one or more text files.

```
shell-prompt: cat file [file ...]
```

```
shell-prompt: cat /etc/hosts
```

The **vis** and **cat -v** commands display invisible characters in a visible way. For example, carriage return characters present in Windows files are normally not shown by most Unix commands. The vis and cat -v commands will show them as '^M' (representing Control+M, which is what you would type to produce this character).

```
shell-prompt: cat sample.txt
This line contains a carriage return.
shell-prompt: vis sample.txt
This line contains a carriage return.\^M
shell-prompt: cat -v sample.txt
This line contains a carriage return.^M
```

**head** shows the top N lines of one or more text files.

```
shell-prompt: head -n # file [file ...]
```

If the flag -n followed by an integer number N is given, the top N lines are shown instead of the default of 10.

```
shell-prompt: head -n 5 /etc/hosts
```

The **head** command can also be useful for generating small test inputs. Suppose you're developing a new program or script that processes genomic sequence files in FASTA format. Real FASTA files can contain millions of sequences and take a great deal of time to process. For testing new code, we don't need much data, and we want the test to complete in a few seconds rather than hours. We can use **head** to extract a small number of sequences from a large FASTA file for quick testing. Since FASTA files have alternating header and sequence lines, we must always choose a multiple of 2 lines. We use the output redirection operator (>) to send the head output to a file instead of the terminal screen. Redirection is covered in Section 3.13.

```
# You cannot run this command unless you have a file called
# reall-big.fasta in the CWD
shell-prompt: head -n 1000 really-big.fasta > small-test.fasta
```

**tail** shows the bottom N lines of one or more text files.

```
shell-prompt: tail -n # file [file ...]
```

Tail is especially useful for viewing the end of a large file that would be cumbersome to view with **more**.

If the flag -n followed by an integer number N is given, the bottom N lines are shown instead of the default of 10.

```
shell-prompt: tail -n 5 /etc/hosts
```

The **diff** command shows the differences between two text files. This is most useful for comparing two versions of the same file to see what has changed. Also see **cdiff**, a specialized version of **diff**, for comparing C source code.

The `-u` flag asks for *unified diff* output, which shows the removed text (text in the first file by not the second) preceded by '-', the added text (text in the second file but not the first) preceded by a '+', and some unchanged lines for context. Most people find this easier to read than the default output format.

```
shell-prompt: printf "1\n2\n3\n" > input1.txt
shell-prompt: printf "2\n3\n4\n" > input2.txt
shell-prompt: diff input1.txt input2.txt
shell-prompt: diff -u input1.txt input2.txt
shell-prompt: rm input1.txt input2.txt
```

#### Text Editors

There are more text editors available for Unix systems than any one person is aware of. Some are terminal-based, some are graphical, and some have both types of interfaces.

All Unix systems support running graphical programs from remote locations, but many graphical programs require a fast connection (100 megabits/sec) or more to function comfortably.

Knowing how to use a terminal-based text editor is therefore a very good idea, so that you're prepared to work on a remote Unix system over a slow connection if necessary. Some of the more common terminal-based editors are described below.

**vi** (visual editor) is the standard text editor for all Unix systems. Most users either love or hate the vi interface, but it's a good editor to know since it is available on every Unix system.

**nano** is an extremely simplistic text editor that is ideal for beginners. It is a rewrite of the **pico** editor, which is known to have many bugs and security issues. Neither editor is standard on Unix systems, but both are free and easy to install. These editors entail little or no learning curve, but are not sophisticated enough for extensive programming or scripting.

**emacs** (Edit MACroS) is a more sophisticated editor used by many programmers. It is known for being hard to learn, but very powerful. It is not standard on most Unix systems, but is free and easy to install.

**ape** is a menu-driven, user-friendly IDE (integrated development environment), i.e. programmer's editor. It has an interface similar to PC and Mac programs, but works on a standard Unix terminal. It is not standard on most Unix systems, but is free and easy to install. **ape** has a small learning curve, and advanced features to make programming much faster.

**Eclipse** is a popular open-source graphical IDE written in Java, with support for many languages. It is sluggish over a slow connection, so it may not work well on remote systems over ssh.

#### Networking

**hostname** prints the network name of the machine.

```
shell-prompt: hostname
```

This is often useful when you are working on multiple Unix machines at the same time (e.g. via **ssh**), and forgot which window applies to each machine.

#### Identity and Access Management

**passwd** changes your password. It asks for your old password once, and the new one twice (to ensure that you don't accidentally set your password to something you don't know because your finger slipped). Unlike many graphical password programs, **passwd** does not echo anything for each character typed. Even allowing someone to see the length of your password is a bad idea from a security standpoint.

```
# This may not work on systems using an authentication service
# rather than local passwords
shell-prompt: passwd
```

The **passwd** command is generally only used for setting local passwords on the Unix machine itself. Many Unix systems are configured to authenticate users via a remote service such as *Lightweight Directory Access Protocol* (*LDAP*) or *Active Directory* (*AD*). Changing LDAP or AD passwords may require using a web portal to the LDAP or AD server instead of the **passwd** command.

#### Terminal Control

**clear** clears your terminal screen (assuming the TERM environment variable is properly set).

```
shell-prompt: clear
```

**reset** resets your terminal to its default state. This is useful when your terminal has been corrupted by bad output, such as when attempting to view a binary file with **cat**.

Terminals are controlled by *magic sequences*, sequences of invisible control characters sent from the host computer to the terminal amid the normal output. Magic sequences move the cursor, change the color, change the international character set, etc. Binary files contain random data that sometimes by chance contain magic sequences that could alter the mode of your terminal. If this happens, running **reset** will usually correct the problem. If not, you will need to log out and log back in.

```
shell-prompt: reset
```

Table 3.7 provides a quick reference for looking up common Unix commands. For details on any of these commands, run **man command** (or **info command** on some systems).

| **Synopsis** | **Description** |
|---|---|
| ls [file\|directory] | List file(s) |
| cp source-file destination-file | Copy a file |
| cp source-file [source-file ...] directory | Copy multiple files to a directory |
| mv source-file destination-file | Rename a file |
| mv source-file [source-file ...] directory | Move multiple files to a directory |
| ln source-file destination-file | Create another name for the same file. (source and destination must be in the same file system) |
| ln -s source destination | Create a symbolic link to a file or directory |
| rm file [file ...] | Remove one or more files |
| rm -r directory | Recursively remove a directory and all of its contents |
| mkdir directory | Create a directory |
| rmdir directory | Remove a directory (the directory must be empty) |
| od/hexdump | Show the contents of a file in octal/hexadecimal |
| sort | Sort text files based on flexible criteria |
| uniq | Echo files, eliminating adjacent duplicate lines. |
| diff | Show differences between text files. |
| cmp | Detect differences between binary files. |
| cdiff | Show differences between C programs. |
| date | Show the current date and time. |
| cal | Print a calendar for any month of any year. |
| printenv | Print environment variables. |

Table 3.7: Unix Commands

### 3.10.5 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What types of commands have to be internal to the shell? Give one example and explain why it must be internal.
2. How can you find a list of the basic Unix commands available on your system?
3. How can you find out whether the **grep** command is internal or external, and where it is located?
4. What kind of suffering did computer users have to endure in order to read documentation before the Unix renaissance? How did Unix put an end to such suffering?

5. Show a Unix command that helps us learn about all the command-line flags available for the **tail** command.
6. Show a Unix command that copies the file /tmp/sample.txt to the CWD.
7. Show a Unix command that copies all files in /tmp whose names begin with "sample" and end with ".txt" to the CWD.
8. Show a Unix command that moves all the files in the CWD whose names end with ".py" to a subdirectory of the CWD called "Python".
9. Show a Unix command that creates another file name in the CWD called test-input.txt for the existing file ./Data/input.txt.
10. What is a hard link?
11. What is a symbolic link?
12. What do we get when we remove the path name to which a symbolic link points?
13. What limitations do hard links have that soft links do not have?
14. How do we create a new directory /home/joe/Data/Project1 if the Data directory does not exist and the CWD is /home/joe?
15. How do we remove the directory ./Data if it is empty? If it is not empty?
16. Show a Unix command that tells us how much disk space is available in each file system.
17. Show a Unix command that tells us how much space is used by the directory ./Data.
18. Show a sequence of Unix commands that change CWD to /tmp, then to /etc and then return to the original CWD.
19. How do we exit the shell?
20. Show a Unix command that tells us if there are carriage returns in graph.py.
21. Show a Unix command that displays the first 20 lines of output.txt.
22. Show a Unix command that displays the last 20 lines of output.txt.
23. Show a Unix command that displays what has changed between analysis.c.old and analysis.c.
24. Which text editor is available on all Unix systems?
25. Show a Unix command that tells us the name of the machine running our shell.
26. Show a Unix command to the remote server unixdev1.ceas.uwm.edu as user joe in order to run commands on it.
27. Show a Unix command to change our local password.
28. How do we change our password for a Unix system that relies on LDAP or AD?
29. Show a Unix command that clears the terminal display.
30. Show a Unix command to reset the terminal mode to default settings.

## 3.11 POSIX and Extensions

Unix-compatible systems generally conform to standards published by the International Organization for Standardization (ISO), the Open Group, and the IEEE Computer Society.

The primary standard used for this purpose is *POSIX*, the Portable Operating System standard based on UnIX. Programs and commands that conform to the POSIX standard will work on any Unix system. Therefore, developing your programs and scripts according to POSIX will prevent the need for even minor changes when porting from one Unix variant to another.

Nevertheless, many common Unix programs have been enhanced beyond the POSIX standard to provide conveniences. Fortunately, most such programs are open source and can therefore be easily installed on most Unix systems. Features that do not conform to the POSIX standard are known as *extensions*. Extensions are often described according to their source, e.g. BSD extensions that come from BSD Unix variants or GNU extensions that come from the GNU software project.

Many standard commands such as awk, make, and sed, may contain extensions that depend on the specific operating system. For example, BSD systems use the BSD versions of awk, make, and sed, which contain BSD extensions, while GNU/Linux systems use the GNU versions of awk, make, and sed, which contain GNU extensions.

When installing GNU software on BSD systems, the GNU version of the command is usually prefixed with a 'g', to distinguish it from the native BSD command. For example, on FreeBSD, "make" and "awk" are the BSD implementations and "gmake" and "gawk" would be the GNU implementations. Likewise, on GNU/Linux systems, BSD commands would generally be prefixed with a 'b' or 'bsd'. The "make" and "tar" commands on GNU/Linux would refer to GNU versions and the BSD versions would be "bmake" and "bsdtar".

All of them will support POSIX features, so if you use only POSIX features, they will behave the same way. If you use GNU or other extensions, you should use the GNU command, e.g. gawk instead of awk.

| **Program** | **Example of extensions** |
|---|---|
| BSD Tar | Support for extracting ISO and Apple DMG files |
| GNU Make | Various "shortcut" rules for compiling multiple source files |
| GNU Awk | Additional built-in functions |

Table 3.8: Common Extensions

### 3.11.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is POSIX and why is it important?
2. What is an extension?
3. Does the use of extensions always prevent things from working on other Unix systems?

## 3.12 Subshells

Commands placed between parentheses are executed in a new child shell process rather than the shell process that received the commands as input.

This can be useful if you want a command to run in a different directory or with other alterations to its environment, without affecting the current shell process.

```
shell-prompt: (cd /etc; ls)
```

Since the commands above are executed in a new shell process, the shell process that printed "shell-prompt: " will not have its current working directory changed. This command has the same net effect as the following:

```
shell-prompt: pushd /etc
shell-prompt: ls
shell-prompt: popd
```

### 3.12.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show a single Unix command that runs pwd and produces the output "/etc", without changing the CWD of the shell process.

## 3.13 Redirection and Pipes

### 3.13.1 Device Independence

Many operating systems that came before Unix treated each input or output device differently. Each time a new device became available, programs would have to be modified in order to access it. This is intuitive, since the devices all look different and perform different functions.

The Unix designers realized that this is actually unnecessary and a waste of programming effort, so they developed the concept of *device independence*. What this means is that *Unix treats virtually every input and output device exactly like an ordinary file.* All input and output, whether to/from a file on a disk, a keyboard, a mouse, a scanner, or a printer, is simply a stream of bytes to be input or output *using the same tools*.

Most I/O devices are actually accessible as a *device file* in `/dev`. For example, the primary CD-ROM might be `/dev/cd0`, the main disk `/dev/ad0`, the keyboard `/dev/kbd0`, and the mouse `/dev/sysmouse`.

A user with sufficient permissions can view input coming from these devices using the same Unix commands we use to view a file:

```
shell-prompt: cat /dev/kbd0
shell-prompt: more /dev/cd0
```

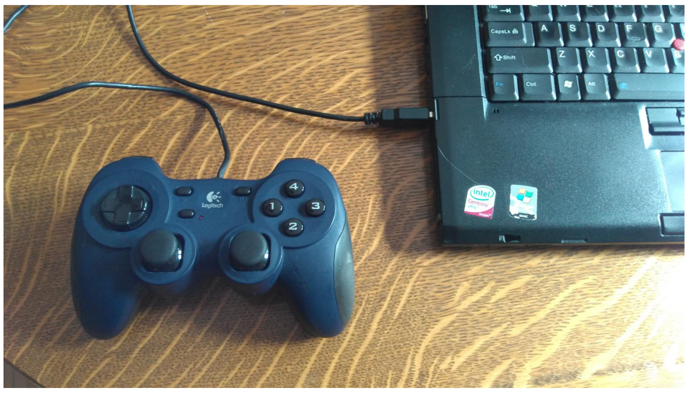
In fact, data are often recovered from corrupted file systems or accidentally deleted files by searching the raw disk partition as a file using standard Unix commands such as grep!

```
shell-prompt: grep string /dev/ad0s1f
```

A keyboard sends text data, so `/dev/kbd0` is like a text file. Many other devices send binary data, so using **cat** to view them would output gibberish. To see the raw input from a mouse as it is being moved, we could instead use **hexdump**, which displays the bytes of input as numbers rather than characters:

```
shell-prompt: hexdump /dev/sysmouse
```

Some years ago while mentoring my son's robotics team, as part of a side project, I reverse-engineered a USB game pad so I could control a Lego robot via Bluetooth from a laptop. Thanks to device-independence, no special software was needed to figure out the game pad's communication protocol.



After plugging the game pad into my FreeBSD laptop, the system creates a new UHID (USB Human Interface Device) under `/dev`. The **dmesg** command shows the name of the new device file:

```
ugen1.2: <vendor 0x046d product 0xc216> at usbus1
uhid0 on uhub3
uhid0: <vendor 0x046d product 0xc216, class 0/0, rev 1.10/3.00, addr 2> on usbus1
```

One can then view the input from the game pad using **hexdump**:

```
FreeBSD manatee.acadix  bacon ~ 410: hexdump /dev/uhid0
0000000 807f 7d80 0008 fc04 807f 7b80 0008 fc04
0000010 807f 7780 0008 fc04 807f 6780 0008 fc04
0000020 807f 5080 0008 fc04 807f 3080 0008 fc04
0000030 807f 0d80 0008 fc04 807f 0080 0008 fc04
0000060 807f 005e 0008 fc04 807f 005d 0008 fc04
0000070 807f 0060 0008 fc04 807f 0063 0008 fc04
0000080 807f 006c 0008 fc04 807f 0075 0008 fc04
0000090 807f 0476 0008 fc04 807f 1978 0008 fc04
00000a0 807f 4078 0008 fc04 807f 8c7f 0008 fc04
00000b0 807f 807f 0008 fc04 807f 7f7f 0008 fc04
00000c0 807f 827f 0008 fc04 807f 847f 0008 fc04
00000d0 807f 897f 0008 fc04 807f 967f 0008 fc04
00000e0 807f a77f 0008 fc04 807f be80 0008 fc04
00000f0 807f d980 0008 fc04 807f f780 0008 fc04
0000100 807f ff80 0008 fc04 807f ff83 0008 fc04
0000110 807f ff8f 0008 fc04 807f ff93 0008 fc04
```

To understand these numbers, we need to know a little about hexadecimal, base 16. This is covered in detail in Chapter 14. In short, it works the same as decimal, but we multiply by powers of 16 rather than 10, and digits go up to 15 rather than 9. Digits for 10 through 15 are A, B, C, D, E, and F. The largest possible 4-digit number is therefore FFFF_16. 8000_16 is in the middle of the range.

```
0000_16 =  0 * 16^3 +  0 * 16^2 +  0 * 16^1 +  0 * 16^0 = 0_10
8000_16 =  8 * 16^3 +  0 * 16^2 +  0 * 16^1 +  0 * 16^0 = 32,678_10
FFFF_16 = 15 * 16^3 + 15 * 16^2 + 15 * 16^1 + 15 * 16^0 = 65,535_10
```

It was easy to see that moving the right joystick up resulted in lower numbers in the 3rd and 7th columns, while moving down increased the values. Center position sends a value around 8000 (hexadecimal), fully up is around 0, fully down is ffff.

It was then easy to write a small program to read the joystick position from the game pad (by simply opening /dev/uhid0 like any other file) and send commands over Bluetooth to the robot, adjusting motor speeds accordingly. The Bluetooth interface is simply treated as an output file.

### 3.13.2 Redirection

Since I/O devices and files are interchangeable, Unix shells can provide a facility called *redirection* to easily interchange them for *any process* without the process even knowing it.

Redirection depends on the notion of a *file stream*. You can think of a file stream as a hose connecting a program to a particular file or device, as shown in Figure 3.3. Redirection simply disconnects the hose from the default file or device (such as the keyboard or terminal screen) and connects it to another file or device chosen by the user.

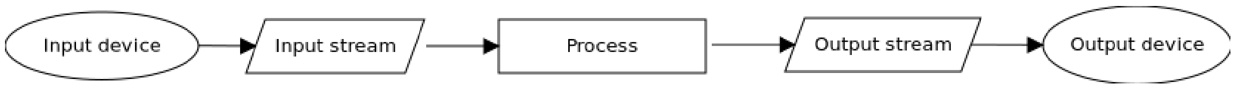


Figure 3.3: File streams

Every Unix process has three standard streams that are open from the moment the process is born. The standard streams for a shell process are normally connected to the terminal, as shown in Table 3.9 and Figure 3.4.

| Stream | Purpose | Default Connection |
|---|---|---|
| Standard Input | User input | Terminal keyboard |
| Standard Output | Normal output | Terminal screen |
| Standard Error | Errors and warnings | Terminal screen |

Table 3.9: Standard Streams

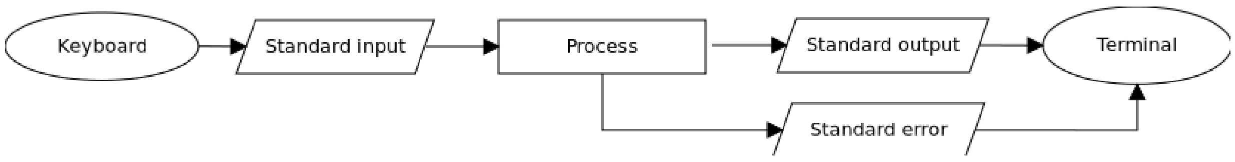


Figure 3.4: Standard streams

Redirection in the shell allows any or all of the three standard streams to be disconnected from the terminal and connected to a file or other I/O device. It uses special operator characters within the commands to indicate which stream(s) to redirect and where. The basic redirection operators shells are shown in Table 3.10.

| Operator | Shells | Redirection type |
|---|---|---|
| < | All | Standard Input |
| > | All | Standard Output (overwrite) |
| >> | All | Standard Output (append) |
| 2> | Bourne-based | Standard Error (overwrite) |
| 2>> | Bourne-based | Standard Error (append) |
| >& | C shell-based | Standard Output and Standard Error (overwrite) |
| >>& | C shell-based | Standard Output and Standard Error (append) |

Table 3.10: Redirection Operators

**Note** Memory trick: A redirection operator is an arrow that points in the direction of data flow.

```
shell-prompt: ls > listing.txt          # Overwrite with listing of .
shell-prompt: ls /etc >> listing.txt    # Append listing of /etc
```

In the examples above, the **ls** process sends its output to `listing.txt` instead of the terminal, as shown in Figure 3.5.

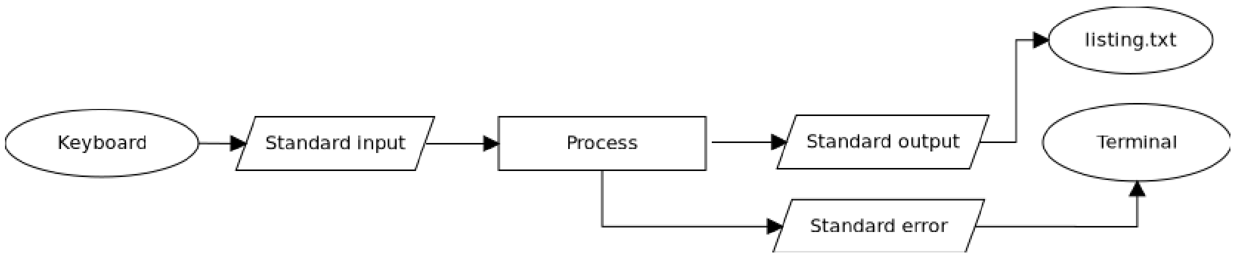


Figure 3.5: Redirecting standard output

However, the filename `listing.txt` is *not* an argument to the **ls** process. The **ls** process never even knows about this output file. The redirection is handled by the shell and the shell removes "> listing.txt" and ">> listing.txt" from these commands *before* *executing them*. So, the first **ls** receives no arguments, and the second receives only `/etc`. Most programs have no idea whether their output is going to a file, a terminal, or some other device. They don't need to know and they don't care.

**Caution**
Using output redirection (>, 2>, or >&) in a command will normally overwrite (clobber) the file that you're redirecting to, even if the command itself fails. Be very careful not to use output redirection accidentally. This most commonly occurs when a careless user meant to use input redirection, but pressed the wrong key.
The moment you press Enter after typing a command containing "> filename", filename will be erased! Remember that the shell performs redirection, not the command, so filename is clobbered before the command is even executed.
If `noclobber` is set for the shell, output redirection to a file that already exists will result in an error. The `noclobber` option can be overridden by appending a ! to the redirection operator in C shell derivatives or a | in Bourne shell derivatives. For example, >! can be used to force overwriting a file in csh or tcsh, and >| can be used in sh, ksh, or bash.

More often than not, we want to redirect both normal output and error messages to the same place. This is why C shell and its derivatives use a combined operator that redirects both at once.

```
shell-prompt: find /etc >& all-output.txt
```

The same effect can be achieved with Bourne-shell derivatives using another operator that redirects one stream to another stream. In particular, we redirect the standard output (stream 1) to a file (or device) and at the same time redirect the standard error (stream 2) to stream 1.

```
shell-prompt: find /etc > all-output.txt 2>&1
```

In Bourne family shells, we can separately redirect the standard output with > and the standard error with 2>:

```
shell-prompt: find /etc > list.txt 2> errors.txt
```

If we want to separate standard output and standard error in a C shell or T shell session, we can use a subshell under which the **find** command redirects only the standard output. The output from the subshell process will then only contain the standard error left over from **find**, which we can redirect with &>:

```
shell-prompt: (find /etc > list.txt) >& errors.txt
```

If a program takes input from the standard input, we can redirect input from a file as follows:

```
shell-prompt: command < input-file
```

For example, the "bc" (binary calculator) command is an arbitrary-precision calculator that inputs numerical expressions from the standard input and writes the results to the standard output. It's a good idea to use the `--mathlib` flag with **bc** for more complete functionality.

```
shell-prompt: bc --mathlib
3.14159265359 * 4.2 ^ 2 + sqrt(30)
60.89491440932
quit
```

In the example above, the user entered "3.14159265359 * 4.2 ^ 2 + sqrt(30)" and "quit" and the bc program output "60.89491440932". We could instead place the input shown above in a file using any text editor, such as nano or vi, or even using **cat** with keyboard input and output redirection as a primitive editor:

```
shell-prompt: cat > bc-input.txt
3.14159265359 * 4.2 ^ 2 + sqrt(30)
quit
(Type Ctrl+d to signal the end of input to the cat process)
shell-prompt: cat bc-input.txt
3.14159265359 * 4.2 ^ 2 + sqrt(30)
quit
```

Now that we have the input in a file, we can feed it to the **bc** process using input redirection instead of retyping it on the keyboard:

```
shell-prompt: bc --mathlib < bc-input.txt
60.29203070318
```

### 3.13.3 Special Files in /dev

The standard streams themselves are represented as device files on Unix systems. This allows us to redirect one stream to another without modifying a program, by appending the stream to one of the device files `/dev/stdout` or `/dev/stderr`. For example, if a program sends output to the standard output and we want to send it instead to the standard error, we could do something like the following:

```
shell-prompt: printf "Oops!" >> /dev/stderr
```

If we would like to simply discard output sent to the standard output or standard error, we can redirect it to `/dev/null`. For example, to see only error messages (standard error) from myprog, we could do the following:

```
shell-prompt: ./myprog > /dev/null
```

To see only normal output and not error messages, assuming Bourne shell family:

```
shell-prompt: ./myprog 2> /dev/null
```

In C shell family:

```
shell-prompt: ( find /etc > output.txt ) >& /dev/null ; cat output.txt
```

The device `/dev/zero` is a readable file that produces a stream of zero bytes.

The device `/dev/random` is a readable file that produces a stream of random integers in binary format. We can use the **dd** command, a bit copy program, to copy a fixed number of bytes from one file to another. We specify the input file with "if=", output with "of=", block size with "bs=", and the number of blocks with "count=". Total data copied will be block-size * count.

```
shell-prompt: dd if=/dev/random of=random-data bs=1000000 count=10
```

**Note** The block size indicates the size of the memory buffer used to store each chunk of the file. Make it large enough to keep the number of disk reads/writes low, but not so large that it will use a significant portion of available memory. A block size of a gigabyte may stress the system's memory resources, and you won't see much improvement in speed using block sizes more than several kibibytes.

### 3.13.4 Pipes

Very often, we want to use the output of one program as input to another. Such a thing could be done using redirection, as shown below:

```
shell-prompt: ls > listing.txt
shell-prompt: more listing.txt
```

The same task can be accomplished in one command using a *pipe*. A pipe redirects one of the standard streams, just as redirection does, but to or from another process instead of a file or device. In other words, we can use a pipe to send the standard output and/or standard error of one process directly to the standard input of another process.

A pipe is constructed by placing the pipe operator (|) between two commands. The whole chain of commands connected by pipes is called a *pipeline*.

**Example 3.17** Simple Pipe

The command below uses a pipe to redirect the standard output of an **ls** process directly to the standard input of a **more** process.

```
shell-prompt: ls | more
```

Since a pipe runs multiple processes in the same shell, it is necessary to understand the concept of *foreground* and *background* processes, which are covered in detail in Section 3.18.

Multiple processes can output to a terminal at the same time, although the results would obviously be chaos in most cases.

In contrast to output, only one process can be receiving input from the keyboard, however. It would be a remarkable coincidence if the same input made sense to two different programs.

The *foreground process* running under a given shell process is defined as the process that receives the input from the terminal. This is the only difference between a foreground process and a background process.

When running a pipeline command, the *last* command in the pipeline becomes the foreground process. All others run in the background, i.e. do not use the standard input device inherited from the shell process. Hence, when we run:

```
shell-prompt: ls | more
```

It is the **more** command that receives input from the keyboard. The more command has its standard input redirected from the standard output of **ls**, and the standard input of the **ls** command is effectively disabled.

**Note** The **more** command is somewhat special: Since its standard input is used to receive input from the pipe, it opens another stream to connect to the keyboard so that it can still get user input, such as pressing the space bar for another screen, etc.

This is such a common practice that Unix has defined the term *filter* to apply to programs that can be used in this way. A filter is any command that can receive input from the standard input and send output to the standard output. Many Unix commands are designed to accept a file name as an argument, but to use the standard input and/or standard output if no filename arguments are provided.

**Example 3.18** Filters

The **more** command is commonly used as a filter. It can read a file whose name is provided as an argument, but will use the standard input if no argument is provided. Hence, the following two commands have the same effect:

```
shell-prompt: more names.txt
shell-prompt: more < names.txt
```

The only difference between these two commands is that in the first, the **more** process receives `names.txt` as a command line argument, opens the file itself (creating a new file stream), and reads from the new stream (not the standard input stream). In the second instance, the *shell* process opens `names.txt` and connects the standard input stream of the **more** process to it. The **more** process then uses another stream to read user input from the keyboard.
Using the filtering capability of more, we can paginate the output of any command:

```
shell-prompt: ls | more
shell-prompt: find . -name '*.c' | more
shell-prompt: sort names.txt | more
```

We can string any number of commands together using pipes. The only limitations are imposed by the memory requirements of the processes in the pipeline. For example, the following pipeline sorts the names in names.txt, removes duplicates, filters out all names not beginning with 'B', and shows the first 100 results one page at a time.

```
shell-prompt: sort names.txt | uniq | grep '^B' | head -n 100 | more
```

To see lines 101 through 200 of a file output.txt:

```
shell-prompt: head -n 200 output.txt | tail -n 100
```

One more useful tool worth mentioning is the **tee** command. The **tee** command is a simple program that reads from its standard input and writes to both the standard output and to one or more files whose names are provided on the command line. This allows you to view the output of a program on the screen and save it to a file at the same time.

```
shell-prompt: ls | tee listing.txt
```

Recall that Bourne-shell derivatives do not have combined operators for redirecting standard output and standard error at the same time. Instead, we redirect the standard output to a file or device, and redirect the standard error to the standard output using `2>&1`.

We can use the same technique with a pipe, but there is one more condition: For technical reasons, the `2>&1` must come *before* the pipe.

```
shell-prompt: ls | tee listing.txt 2>&1    # Won't work
shell-prompt: ls 2>&1 | tee listing.txt    # Will work
```

The **yes** command (much like Jim Carrey in "Yes Man") produces a stream of y's followed by newlines. It is meant to be piped into a program that prompts for y's or n's in response to yes/no questions, so that the program will receive a yes answer to all of its prompts and run without user input.

```
shell-prompt: yes | ./myprog
```

The **yes** command can actually print any response we want, via a command line argument. To answer 'n' to every prompt, we could do the following:

```
shell-prompt: yes n | ./myprog
```

In cases where the response isn't always the same, we can feed a program a arbitrary sequence of responses using redirection or pipes. Be sure to add a newline (\n) after each response to simulate pressing the Enter key:

```
shell-prompt: printf "y\nn\ny\n" | ./myprog
```

Or, to save the responses to a file for repeated use:

```
shell-prompt: printf "y\nn\ny\n" > responses.txt
shell-prompt: ./myprog < responses.txt
```

### 3.13.5 Misusing Pipes

**Aside**
It's important to learn from the mistakes of others, because we don't have time to make them all ourselves.

Users who don't fully understand Unix and processes often fall into bad habits that can potentially be costly. There are far too many such habits to cover here: One could write a separate 1,000-page volume called "Favorite Bad Habits of Unix Users". As a less painful alternative, we'll explore one common bad habit in detail and try to help you understand how to spot others. Our feature habit of the day is the use of the **cat** command at the head of a pipeline:

```
shell-prompt: cat names.txt | sort | uniq > outfile
```

So what's the alternative, what's wrong with using **cat** this way, what's the big deal, why do people do it, and how do we know it's a problem?

1. The alternative:

   Most commands used downstream of **cat** in situations like this (e.g. **sort**, **grep**, **more**, etc.) are capable of reading a file directly if given the filename as an argument:

```
shell-prompt: sort names.txt | uniq > outfile
```

Even if they don't take a filename argument, we can always use simple redirection instead of a pipe:

```
shell-prompt: sort < names.txt | uniq > outfile
```

2. The problem:

   - Using **cat** this way just adds overhead in exchange for no benefit. Pipes are helpful when you have to perform multiple processing steps in sequence. By running multiple processes at the same time instead of one after the other, we can improve resource utilization. For example, while **sort** is waiting for disk input, **uniq** can use the CPU. Better yet, on a computer with multiple cores, the processes can utilize two cores at the same time.

     However, the **cat** command doesn't do any processing at all. It just reads the file and feeds the bytes into the first pipe.

     In using **cat** this way, here's what happens:

     (a) The **cat** command reads blocks from the file into a file input buffer.
     (b) It then copies the input buffer, one byte at a time, to its standard output buffer, without processing the data in any way. It just senselessly moves data (through a proverbial straw) from one memory buffer to another.
     (c) When the standard output buffer is full, it is copied to the pipe, which is yet another memory buffer.
     (d) Characters in the pipe buffer are copied to the standard input buffer of the next command (e.g. **sort**).
     (e) The **sort** can finally begin processing the data.

     This is like pouring a drink into a glass, then moving it to a second glass using an eye dropper, then pouring it into a third glass and finally a fourth glass before actually drinking it.

     It's much simpler and less wasteful for the **sort** command to read directly from the file.

   - Using a pipe this way also prevents the downstream command from optimizing disk access. A program such as **sort** might use a larger input buffer size to reduce the number of disk reads. Reading fewer, larger blocks from disk can keep the latency incurred for each disk operation from adding up, thereby reducing run time. This is not possible when reading from a pipe, which is a fixed-size memory buffer.

3. What's the big deal?

   Usually, this is not much of a problem. Wasting a few seconds or minutes on your laptop won't hurt anyone. However, sometimes mistakes like this one are incorporated into HPC cluster jobs using hundreds of cores for weeks at a time. In that case, it could increase run time by several days, delaying the work of other users who have jobs waiting in the queue, as well as your own. Not to mention, the wasted electricity could cost the organization hundreds of dollars and create additional pollution.

4. Why do people do things like this?

   By far the most common response I get when asking people about this sort of thing is: "[Shrug] I copied this from an example on the web. Didn't really think about it."

   Occasionally, someone might think they are being clever by doing this. They believe that this speeds up processing by splitting the task into two processes, hence utilizing multiple cores, one running **cat** to handle the disk input and another dedicated to **sort** or whatever command is downstream. However, this strategy only helps if both processes are *CPU-bound*, i.e. they spend more time using the CPU than performing input and output. This is not the case for the **cat** command.

   One might also think it helps by overlapping disk input and CPU processing, i.e. **cat** can read the next block of data while **sort** is processing the current one. This may have worked a long time ago using slow disks and unsophisticated operating systems, but it only backfires with modern disks and modern Unix systems that have sophisticated disk buffering.

   In reality, this strategy only increases the amount of CPU time used, and almost always increases run time.

5. Detection:

   Detecting performance issues is pretty easy. The most common tool is the **time** command.

```
shell-prompt: time fgrep GGTAGGTGAGGGGCGCCTCTAGATCGGAAGAGCACACGTCTGAACTCCAGTCA test.vcf ↩
    > /dev/null
2.539u 6.348s 0:09.86 89.9% 92+173k 35519+0io 0pf+0w
```

We have to be careful when using **time** with a pipeline, however. Depending on the shell and the time command used (some shells have in internal implementation), it may not work as expected. We can ensure proper function by wrapping the pipeline in a separate shell process, which is then timed:

```
shell-prompt: time sh -c "cat test.vcf | fgrep  ↩
    GGTAGGTGAGGGGCGCCTCTAGATCGGAAGAGCACACGTCTGAACTCCAGTCA > /dev/null"
2.873u 17.008s 0:13.68 145.2%   33+155k 33317+0io 0pf+0w
```

Table 3.11 compares the run times (wall time) and CPU time of the direct **fgrep** and piped **fgrep** shown above three different operating systems.

All runs were performed on otherwise idle system. Several trials were run to ensure reliable results. Times from the first read of `test.vcf` were discarded, since subsequent runs benefit from disk buffering (file contents still in memory from the previous read). The wall time varied significantly on the CentOS system, with the piped command running in less wall time for a small fraction of the trials. The times shown in the table are typical. Times for FreeBSD and MacOS were fairly consistent.

Note that there is significant variability between platforms that should not be taken too seriously. These tests were not run on identical hardware, so they do not tell us anything about relative operating system performance.

We can also collect other data using tools such as **top** to monitor CPU and memory use and **iostat** to monitor disk activity. These commands are covered in more detail in Section 3.14.15 and Section 3.14.16.

| **System specs** | **Pipe wall time** | **No pipe wall time** | **Pipe CPU time** | **No pipe CPU time** |
|---|---|---|---|---|
| CentOS 7 i7 2.8GHz | 33.43 | 29.50 | 13.59 | 8.45 |
| FreeBSD Phenom 3.2GHz | 13.01 | 8.90 | 18.76 | 8.43 |
| MacBook i5 2.7GHz | 81.09 | 81.35 | 84.02 | 81.20 |

Table 3.11: Run times of pipes with cat

### 3.13.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does device independence simplify life for Unix users? Give an example.
2. Show an example Unix command that displays the input from a mouse as it is being moved or clicked.
3. What are the standard streams associated with every Unix process? To what file or device are they connected by default?
4. Show a Unix command that saves the output of **ls -l** to a file called long-list.txt.
5. Show a Unix command that appends the output of **ls -l /etc** to a file called long-list.txt.
6. Show a Unix command that discards the normal output of **ls -l /etc** and shows the error messages on the terminal screen.
7. Show a Bourne shell command that saves the output of **ls -al /etc** to output.txt and any error messages to errors.txt.
8. Show a C shell command that saves the output and errors of **ls -al /etc** to all-output.txt.
9. How does **more list.txt** differ from **more < list.txt**?
10. Show a Unix command that creates a 1 gigabyte file called `new-image` filled with 0 bytes.
11. What are two major advantages of pipes over redirecting to a file and then reading it?
12. Show a Unix command that lists all the files in and under /etc, sorts them, and paginates the output.

13. What is a foreground process?
14. Which program in the following pipeline runs in the foreground?

    ```
    shell-prompt: find /etc | sort | more
    ```

15. What is a filter program?
16. What is the maximum number of commands allowed in a Unix pipeline?
17. Show a Unix command that prints a long listing of /usr/local/bin to the terminal and at the same time saves it to the file local-bin.txt.
18. Do the same as above, but include any error messages in the file as well. Show the command for both C shell and Bourne shell.
19. Is it a good idea to feed files into a pipe using **cat**, rather than have the next command read them directly? Why or why not?

    ```
    Example: Which command below is more efficient?
    cat file.txt | sort | uniq
    sort file.txt | uniq
    ```

## 3.14 Power Tools for Data Processing

### 3.14.1 Introduction

Congratulations on reaching the holy land of Unix data processing. It has often been said that if you know Unix well, you may never need to write a program. The tools provided by Unix often contain all the functionality you need to process your data. They are like a box of Legos from which we can construct a machine to perform almost any data analysis imaginable from the Unix shell.

Most of these tools function as filters, so they can be incorporated into pipelines. Most also accept filenames as command-line arguments for simpler use cases.

In this section, we'll introduce some of the most powerful tools that are heavily used by researchers to process data files. This will certainly reduce, if not eliminate, the need to write your own programs for many projects. This is only an introduction to make you aware of the available tools and the power they can give you.

For more detailed information, consult the man pages and other sources. Some tools, such as **awk** and **sed**, have entire books written about them, in case you want to explore in-depth.

However, do not set out to learn as much as you can about these tools. Set out to learn as much as you *need*. The ability to show off your vast knowledge is not the ability to achieve. Knowledge is not wisdom. Wisdom is doing. Learn what you need to accomplish today's goals as elegantly as possible, and then do it. You will learn more from this doing than from any amount of studying. You will develop problem solving skills and instincts, which are far more valuable than encyclopedic knowledge.

Never stop wondering if there might be an even more elegant solution. Albert Einstein was once asked what was his goal in life. His response: "To simplify." Use the tools presented here to simplify your research and by extension, your life. With this approach can achieve great things without great effort and spend your time savoring the wonders and mysteries of your work rather than memorizing facts that might come in handy one day.

### 3.14.2 Grep

**Grep** shows lines in one or more text streams that match a given *regular expression* (RE). It is an acronym for Global Regular Expression Print (or Pattern or Parser if you prefer).

```
shell-prompt: grep expression [file ...]
```

The expression is often a simple string, but can represent RE patterns as described in detail by **man re_format** on FreeBSD. There are also numerous web pages describing REs.

Using simple strings or REs, we can search any file stream for lines containing information of interest. By knowing how to construct REs that represent the information you seek, you can easily identify patterns in your data.

REs resemble globbing patterns, but they are not the same. For example, ’*’ by itself in a globbing pattern means any sequence of 0 or more characters. In an RE, ’*’ means 0 or more of the preceding character. ’*’ in globbing is expressed as ’.*’ in an RE. Some of the most common RE patterns are shown in Table 3.12.

| **Pattern** | **Meaning** |
|---|---|
| . | Any single character |
| * | 0 or more of the preceding character |
| + | 1 or more of the preceding character |
| [] | One character in the set or range of the enclosed characters (same as globbing) |
| ˆ | Beginning of the line |
| $ | End of the line |
| .* | 0 or more of any character |
| [a-z]* | 0 or more lower-case letters |

Table 3.12: RE Patterns

Consider the following C program:

```
#include <stdio.h>
#include <sysexits.h>
#include <math.h>

int     main(int argc,char *argv[])

{
    puts("Hello!");
    printf("The square root of the # 2 is %f.\n", sqrt(2.0));
    printf("The natural log of the # 2 is %f.\n", log(2.0));

    return EX_OK;
}
```

The command below shows all lines containing a call to the printf() function. We use quotes around the string because the shell will try to interpret the ’(’ without them.

```
shell-prompt: grep 'printf(' prog1.c
    printf("The square root of 2 is %f.\n", sqrt(2.0));
    printf("The natural log of 2 is %f.\n", log(2.0));
```

We might also wish to show all lines containing *any* function call in prog1.c. Since we are looking for any function name rather than one particular name, we cannot use a simple string and must construct a regular expression. Variable and function names begin with a letter or underscore and may contain any number of letters, underscores, or digits after that. So our RE must require a letter or underscore for the first character and then accept zero or more letters, digits, or underscores after that. We will also require an argument list (anything between () is good enough for our purposes) and a semicolon to terminate the statement.

```
shell-prompt: grep '[a-zA-Z_][a-zA-Z0-9_]*(.*);' prog1.c
    puts("Hello!");
    printf("The square root of 2 is %f.\n", sqrt(2.0));
    printf("The natural log of 2 is %f.\n", log(2.0));
```

The following shows lines that have a ’#’ in the first column, which represents a preprocessor directive in C or C++:

```
shell-prompt: grep '^#' prog1.c
```

```
#include <stdio.h>
#include <sysexits.h>
#include <math.h>
```

Without the '^' we match a '#' anywhere in the line:

```
shell-prompt: grep '#' prog1.c
#include <stdio.h>
#include <sysexits.h>
#include <math.h>
    printf("The square root of the # 2 is %f.\n", sqrt(2.0));
    printf("The natural log of the # 2 is %f.\n", log(2.0));
```

**Note** Since REs share many special characters with globbing patterns, we must enclose the RE in quotes to prevent the shell from treating it as a globbing pattern.

**Note** If we want to match a special character such as '.' or '*' literally, we must escape it (preceded it with a '\'). For example, to locate method calls in a Java program, which have the form object.method(arguments);, we could use the following:

```
shell-prompt: grep '[a-zA-Z_][a-zA-Z0-9_]*\.[a-zA-Z_][a-zA-Z0-9_]*\(.*\);' prog1.java
```

As an example of searching data files, rather than program code, suppose we would like to find all the lines containing contractions in text file. This would consist of some letters, followed by an apostrophe, followed by more letters. Since the apostrophe is the same character as the single quotes we might use to enclose the RE, we either need to escape it (with a '\') or use double quotes to enclose the RE.

```
shell-prompt: grep '[a-zA-Z][a-zA-Z]*\'[a-zA-Z][a-zA-Z]*'
shell-prompt: grep "[a-zA-Z][a-zA-Z]*'[a-zA-Z][a-zA-Z]*"
```

Another example would be searching for DNA sequences in a genome. We might use this to locate adapters, artificial sequences added to the ends of DNA fragments for the sequencing process, in our sequence data. Sequences are usually stored one per line in a text file in FASTA format. A common adapter sequence is "CTGTCTCTTATA".

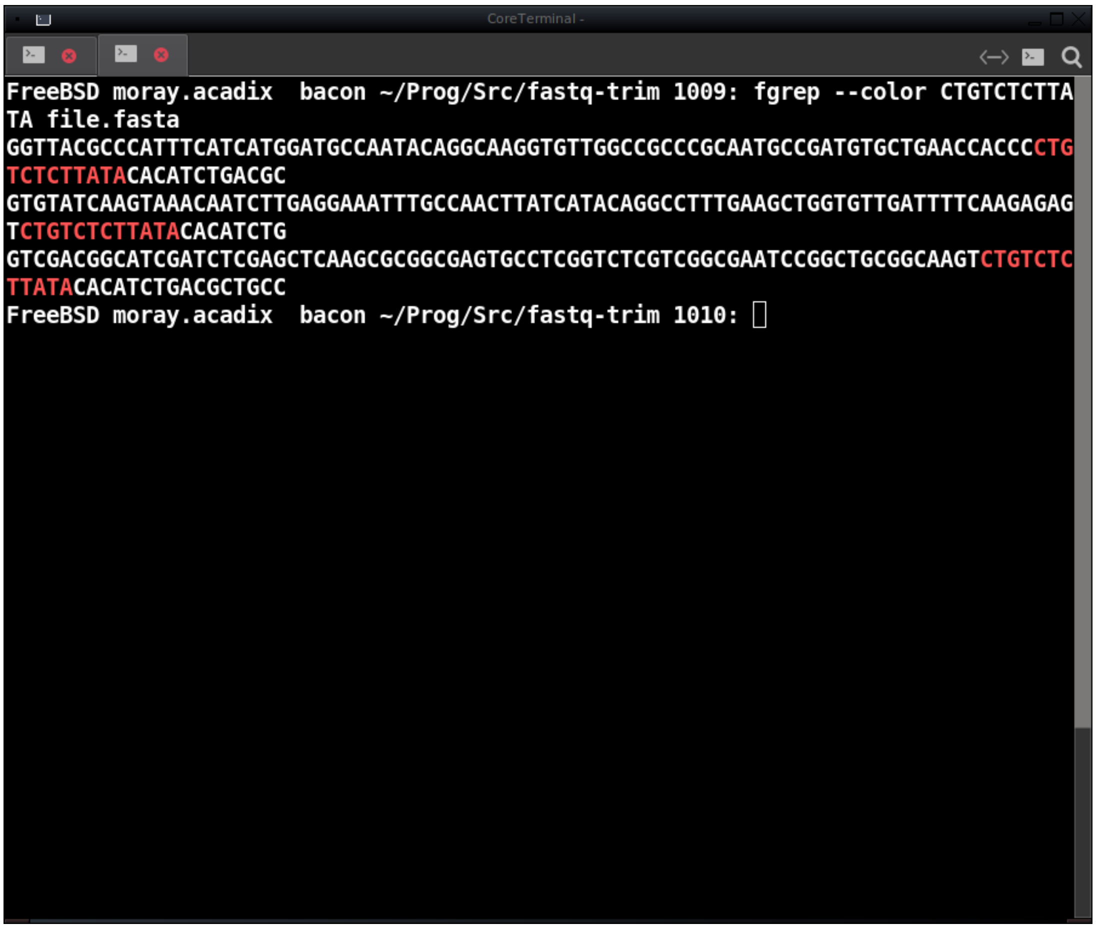
**Note** We can speed up processing by using **grep --fixed-strings** or **fgrep** instead of a regular grep. This uses a more efficient simple string comparison instead of the more complex regular expression matching.

```
shell-prompt: fgrep CTGTCTCTTATA file.fasta
GCGGCCAACACCTTGCCTGTATTGGCATCCATGATGAAATGGGCGTAACCCTGTCTCTTATACACATCTCCGAG
AAAGGCCTGTATGATAAGTTGGCAAATTTCCTCAAGATTGTTTACTTGATACACCTGTCTCTTATACACATCTC
GACCGAGGCACTCGCCGCGCTTGAGCTCGAGATCGATGCCGTCGACCTGTCTCTTATACACATCTCCGAGCCCA
AAAAAATCCCTCCGAAGCATTGTAGGTTTCCATGCTGTCTCTTATACACATCTCCGAGCCCACGAGACTCCTGA
```

DNA sequences sometimes have variations, such as *single nucleotide polymorphisms*, or *SNPs*, where one nucleotide varies in different individuals. Suppose the sequence we're looking for might have either an C or a G in the 5th position. We can use an RE to accommodate this:

```
shell-prompt: grep CTGT[CG]TCTTATA file.fasta
```

It's hard to see the pattern we were looking for in this output. To solve this problem, we can colorize any matched patterns using the `--color` flag as shown in Figure 3.6.



```
FreeBSD moray.acadix  bacon ~/Prog/Src/fastq-trim 1009: fgrep --color CTGTCTCTTA
TA file.fasta
GGTTACGCCCATTTCATCATGGATGCCAATACAGGCAAGGTGTTGGCCGCCCGCAATGCCGATGTGCTGAACCACCCCTG
TCTCTTATACACATCTGACGC
GTGTATCAAGTAAACAATCTTGAGGAAATTTGCCAACTTATCATACAGGCCTTTGAAGCTGGTGTTGATTTTCAAGAGAG
TCTGTCTCTTATACACATCTG
GTCGACGGCATCGATCTCGAGCTCAAGCGCGGCGAGTGCCTCGGTCTCGTCGGCGAATCCGGCTGCGGCAAGTCTGTCTC
TTATACACATCTGACGCTGCC
FreeBSD moray.acadix  bacon ~/Prog/Src/fastq-trim 1010:
```

Figure 3.6: Colorized grep output

There is an extended version of regular expressions that is not supported by the normal **grep** command. Extended REs include things like alternative strings, which are separated by a '|'. For example, we might want to search for either of two adapter sequences. To enable extended REs, we use **egrep** or **grep --extended-regexp**.

```
shell-prompt: egrep 'CTGTCTCTTATA|AGATCGGAAGAG' file.fasta
```

Extended REs also support the '+' modifier to indicate 1 or more of the previous character, e.g. '[a-z]+' is shorthand for '[a-z][a-z]*'.

The **grep** family of commands are very often used as filters in pipelines. If no file name argument is provided, they will read from the standard input, like most Unix commands.

The `-l, --files-with-matches` flag tells **grep** to merely report the names of files that contain a match. This is often used to generate a list of file names for use with another command.

**Example 3.19** Practice Break

```
shell-prompt: ls /usr/bin | grep '^z'
```

### 3.14.3 Awk

**AWK**, an acronym for Aho, Weinberger, and Kernighan (the original developers of the program), is an extremely powerful tool for processing tabular data. Like **grep**, it supports RE matching, but unlike **grep**, it can process individual columns, called *fields*, in the data. It also includes a flexible scripting language that closely resembles the C language, so we can perform highly sophisticated processing of whole lines or individual fields.

Awk can be used to automate many of the same tasks that researchers often perform manually in a spreadsheet program such as LibreOffice Calc or MS Excel.

There are multiple implementations of awk. The most common are "The one true awk", evolved from the original awk code and used on many BSD systems. Gawk, the GNU project implementation, is used on most Linux systems. Mawk is an independent implementation that tends to outperform the others. It is available in most package managers. Awka is an awk-to-C translator that can convert most awk scripts to C for maximize performance.

Fields by default are separated by white space, i.e. space or tab characters. However, **awk** allows us to specify any set of separators using an RE following the `-F` flag or embedded in the script, so we can process tab-separated (.tsv) files, comma-separated (.csv) files, or any other data that can be broken down into columns.

An awk script consists of one or more lines containing a *pattern* and an *action*. The action is enclosed in curly braces, like a C code block.

```
pattern { action }
```

The pattern is used to select lines from the input, usually using a relational expression such as those found in an **if** statement. The action determines what to do when a line is selected. If no pattern is given, the action is applied to every line of input. If no action is given, the default is to print the line.

In both the pattern and the action, we can refer to the entire line as $0. $1 is the first field: all text up to but not including the first separator. $2 is the second field: all text between the first and second separators. And so on...

It is very common to use awk "one-liners" on the command-line, without actually creating an awk script file. In this case, the awk script is the first argument to **awk**, usually enclosed in quotes to allow for white space and special characters. The second argument is the input file to be processed by the script.

For example, the file `/etc/passwd` contains colon-separated fields including the username ($1), user ID ($3), primary group ID ($4), full name ($5), home directory ($6), and the user's shell program ($7). To see a list of full names for every line, we could use the following simple command, which has no pattern (so it processes every line) and an action of printing the fifth field:

```
shell-prompt: awk -F : '{ print $5 }' /etc/passwd
Jason Bacon
D-BUS Daemon User
TCG Software Stack user
Avahi Daemon User
...
```

To see a list of usernames and shells:

```
shell-prompt: awk -F : '{ print $1, $6 }' /etc/passwd
bacon /bin/tcsh
messagebus /usr/sbin/nologin
_tss /usr/sbin/nologin
avahi /usr/sbin/nologin
...
```

Many data files used in research computing are tabular, with one of the most popular formats being TSV (tab-separated value) files. The *General Feature Format*, or *GFF* file is a TSV file format for describing features of a genome. The first field contains the sequence ID (such as a chromosome number) on which the feature resides. The third field contains the feature type, such as "gene" or "exon". The fourth and fifth fields contain the starting and ending positions withing the sequence. The ninth field contains "attributes", such as the globally unique feature ID and possibly the feature name and other information, separated by semicolons. If we just want to see the locations and attributes of all the genes in a genome and their names, we could use the following:

```
shell-prompt:  awk '$3 == "gene" { print $1, $4, $5, $9 }' file.gff3
1 3073253 3074322 ID=gene:ENSMUSG00000102693;Name=4933401J01Rik
1 3205901 3671498 ID=gene:ENSMUSG00000051951;Name=Xkr4
...
```

Awk uses largely the same comparison operators and C and similar languages. One additional **awk** operator that is often useful is ~, which means "contains".

```
# Locate all features whose type contains "RNA".  In a typical GFF3 file,
# this could include mRNA, miRNA, ncRNA, etc.
shell-prompt:  awk '$3 ~ "RNA" { print $1, $4, $5, $9 }' file.gff3
1 3073253 3074322 ID=gene:ENSMUSG00000102693;Name=4933401J01Rik
1 3205901 3671498 ID=gene:ENSMUSG00000051951;Name=Xkr4
...
```

Suppose we want to extract specific attributes from the semicolon-separated attributes field, such as the gene ID and gene name, as well as count the number of genes in the input. This will require a few more awk features.

The gene ID is always the first attribute in the field, assuming the feature is a gene. Not every gene has a name, so we will need to scan the attributes for this information. Awk makes this easy. We can break the attributes field into an array of strings using the `split()` function. We can then use a loop to search the attributes for one beginning with "Name=".

To count the genes in the input, we need to initialize a count variable before we begin processing the file, increment it for each gene found, and print it after processing is finished. For this we can use the special patterns BEGIN and END, which allow us to run an action before and after processing the input.

We will use the C-like `printf()` function to format the output. The basic `print` statement always adds a newline, so it does not allow us to print part of a line and finish it with an subsequent print statement.

Since this is a multiline script, we will save it in a file called `gene-info.awk` and run it using the `-f` flag, which tells awk to get the script from a file rather than the command-line.

```
shell-prompt: awk -f gene-info.awk file.gff3
```

**Caution** Awk can be finicky about the placement of curly braces. To avoid problems, always place the opening brace ({) for an action on the same line as the pattern.

```
BEGIN {
    gene_count = 0;
}

$3 == "gene" {
    # Separate attributes into an array
    split($9, attributes, ";");

    # Print location and feature ID
    printf("%s %s %s %s", $1, $4, $5, attributes[1]);

    # Look for a name attribute and print it if it exists
    # With the for-in loop, c gets the SUBSCRIPT of each element in the
    # attributes array
    for ( c in attributes )
    {
        # See if first 5 characters of the attribute are "Name="
        if ( substr(attributes[c], 1, 5) == "Name=" )
            printf(" %s", attributes[c]);
    }

    # Terminate the output line
    printf("\n");

    # Count this gene
    ++gene_count;
}

END {
    printf("\nGenes found = %d\n", gene_count);
}
```

As we can see, we can do some fairly sophisticated data processing with a very short **awk** script. There is very little that awk cannot do conveniently with tabular data. If a particular task seems like it will be difficult to do with awk, don't give up too easily. Chances are, with a little thought and effort, you can come up with an elegant awk script to get the job done.

That said, there are always other options for processing tabular data. Perl is a scripting language especially well suited to text processing, with its powerful RE handling capabilities and numerous features. Python has also become popular for such tasks in recent years.

Awk is highly efficient, and processing steps performed with it are rarely a bottleneck in an analysis pipeline. If you do need better performance than awk provides, there are C libraries that can be used to easily parse tabular data, such as libxtend. Libxtend includes a set of DSV (delimiter-separated-value) processing functions that make it easy to read fields from files in formats like TSV, CSV, etc. Once you have read a line or an individual field using libxtend's DSV functions, you now have the full power and performance of C at your disposal to process it in minimal time.

Full coverage of awk's capabilities is far beyond the scope of this text. Readers are encouraged to explore it further via the awk man page and one of the many books available on the language.

**Example 3.20** Practice Break

```
shell-prompt: awk -F : '{ print $1 }' /etc/passwd
shell-prompt: awk -F : '$1 == "root" { print $0 }' /etc/passwd
```

### 3.14.4 Cut

The **cut** command is used to select columns from a file, either by byte position, character position, or like **awk**, delimiter-separated columns. Note that characters in the modern world may be more than one byte, so bytes and characters are distinguished here.

To extract columns by byte or character position, we use the `-b` or `-c` option followed by a list of positions. The list is comma-separated and may contain individual positions or ranges denoted with a '-'. For example, to extract character positions 1 through 10 and 21 through 26 from every line of file.txt, we could use the following:

```
shell-prompt: cut -c 1-10,21-26 file.txt
```

For delimiter-separated columns, we use `-d` to indicate the delimiter. The default is a tab character alone, not just any white space. The `-w` flag tells cut to accept any white space (tab or space) as the delimiter. The `-f` is then used to indicate the fields to extract, much like `-c` is used for character positions. Output is separated by the same delimiter as the input.

For example, to extract the username, userid, groupid, and full name (fields 1, 3, 4, and 5) from /etc/passwd, we could use the following:

```
shell-prompt: cut -d : -f 1,3-5 /etc/passwd
...
ganglia:102:102:Ganglia User
nagios:181:181:Nagios pseudo-user
webcamd:145:145:Webcamd user
```

The above is equivalent to the following awk command:

```
shell-prompt: awk -F : '{ printf("%s:%s:%s:%s\n", $1, $3, $4, $5); }' /etc/passwd
```

**Example 3.21** Practice Break

```
shell-prompt: cut -d : -f 1,3-5 /etc/passwd
```

### 3.14.5 Sed

The **sed** command is a stream editor. It makes changes to a file stream with no interaction from the user. It is probably most often used to make simple text substitutions, though it can also do insertions and deletions of lines and parts of lines, even selecting lines by number or based on pattern matching much like **grep** and **awk**. A basic substitution command takes the following format:

```
sed -e ’s|pattern|replacement|g’ input-file
```

Pattern is any regular expression, like those used in **grep** or **awk**. Replacement can be a fixed string, but also takes some special characters, such as &, which represents the string matched by pattern. It can also be empty if you simply want to remove occurrences of pattern from the text.

The characters enclosing pattern and replacement are arbitrary. The ’|’ character is often used because it stands out among most other characters. If either pattern or replacement contains a ’|’, simply use a different separator, such as ’/’. The ’g’ after the pattern means "global". Without it, **sed** will only replace the first occurrence of pattern in each line. With it, all matches are replaced.

```
shell-prompt: cat fox.txt
The quick brown fox jumped over the lazy dog.
shell-prompt: sed -e ’s|fox|worm|g’ fox.txt
The quick brown worm jumped over the lazy dog.
shell-prompt: sed -e ’s/brown //g’ -e ’s|fox|&y worm|g’ fox.txt
The quick foxy worm jumped over the lazy dog.
```

Using `-E` in place of `-e` causes **sed** to support extended regular expressions.

By default, sed sends output to the standard output stream. The `-i` flag tells **sed** to edit the file in-place, i.e. replace the original file with the edited text. This flag should be followed by a filename extension, such as ".bak". The original file will then be saved to filename.bak, so that you can reverse the changes if you make a mistake. The extension can be an empty string, e.g. ” if you are sure you don’t need a backup of the original.

**Caution**

There is a rare portability issue with **sed**. GNU **sed** requires that the extension be nestled against the `-i`:

```
shell-prompt: sed -i.bak -e ’s|pattern|replacement|g’ file.txt
```

Some other implementations require a space between the `-i` and the extension, which is more orthodox among Unix commands:

```
shell-prompt: sed -i .bak -e ’s|pattern|replacement|g’ file.txt
```

FreeBSD’s **sed** accepts either form. You must be aware of this in order to ensure that scripts using **sed** are portable. The safest approach is not to use the `-i` flag, but simply save the output to a temporary file and then move it:

```
shell-prompt: sed -e ’s|pattern|replacement|g’ file.txt > file.txt.tmp
shell-prompt: mv file.txt.tmp file.txt
```

This way, it won’t matter which implementation of **sed** is present when someone runs your script.

Sed is a powerful and complex tool that is beyond the scope of this text. Readers are encouraged to consult books and other documentation to explore further.

**Example 3.22** Practice Break

```
shell-prompt: printf "The quick brown fox jumped over the lazy dog." > fox.txt
shell-prompt: cat fox.txt
shell-prompt: sed -e ’s|fox|worm|g’ fox.txt
shell-prompt: sed -e ’s/brown //g’ -e ’s|fox|&y worm|g’ fox.txt
```

### 3.14.6 Sort

The **sort** command sorts text data line by line according to one or more *keys*. A key indicates a *field* (usually a column separated by white space or some other delimiter) and the type of comparison, such as lexical (like alphabetical, but including non-letters) or numeric.

If no keys are specified, sort compares entire lines lexically. The `--key` followed by a field number restricts comparison to that field. Fields are numbered starting with 1. This can be used in conjunction with the `--field-separator` flag to specify a separator other than the default white space. The `--numeric-sort` flag must be used to perform integer comparison rather than lexical. The `--general-numeric-sort` flag must be used to compare real numbers.

```
shell-prompt: cat ages.txt
Bob Vila        23
Joe Piscopo     27
Al Gore         19
Ingrid Bergman  26
Mohammad Ali    22
Ram Das         9
Joe Montana     25

shell-prompt: sort ages.txt
Al Gore         19
Bob Vila        23
Ingrid Bergman  26
Joe Montana     25
Joe Piscopo     27
Mohammad Ali    22
Ram Das         9

shell-prompt: sort --key 2 ages.txt
Mohammad Ali    22
Ingrid Bergman  26
Ram Das         9
Al Gore         19
Joe Montana     25
Joe Piscopo     27
Bob Vila        23

shell-prompt: sort --key 3 --numeric-sort ages.txt
Ram Das         9
Al Gore         19
Mohammad Ali    22
Bob Vila        23
Joe Montana     25
Ingrid Bergman  26
Joe Piscopo     27
```

The **sort** command can process files of any size, regardless of available memory. If a file is too large to fit in memory, it is broken into smaller pieces, which are sorted separately and saved to temporary files. The sorted temporary files are then merged.

The **uniq** command, which removes adjacent lines that are identical, is often used after sorting to remove redundancy from data. Note that the **sort** command also has a `--unique` flag, but it does not behave the same as the **uniq** command. The `--unique` flag compares keys, while the **uniq** command compares entire lines.

**Example 3.23** Practice Break

Using your favorite text editor, enter a few names from the example above into a file called ages.txt.

```
shell-prompt: cat ages.txt
shell-prompt: sort ages.txt
shell-prompt: sort --key 2 ages.txt
shell-prompt: sort --key 3 --numeric-sort ages.txt
shell-prompt: du -sm * | sort -n    # Determine biggest directories
```

### 3.14.7 Tr

The **tr** (translate) command is a simple tool for performing character conversions and deletions in a text stream. A few examples are shown below. See the **tr** man page for details.

We can use it to convert individual characters in a text stream. In this case, it takes two string arguments. Characters in the Nth position in the first string are replaced by characters in the Nth position in the second string:

```
shell-prompt: cat fox.txt
The quick brown fox   jumped over the lazy dog.
shell-prompt: tr 'xl' 'gh' < fox.txt
The quick brown fog   jumped over the hazy dog.
```

There is limited support for character sets enclosed in square brackets [], similar to regular expressions, including predefined sets such as [:lower:] and [:upper:]:

```
shell-prompt: tr '[:lower:]' '[:upper:]' < fox.txt
THE QUICK BROWN FOX   JUMPED OVER THE LAZY DOG.
```

We can use it to "squeeze" repeated characters down to one in a text stream. This is useful for compressing white space:

```
shell-prompt: tr -s ' ' < fox.txt
The quick brown fox jumped over the lazy dog.
```

The **tr** command does not support doing multiple conversions in the same command, but we can use it as a filter:

```
shell-prompt: tr '[:lower:]' '[:upper:]' < fox.txt | tr -s ' '
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG.
```

There is some overlap between the capabilities of **tr**, **sed**, **awk**, and other tools. Which one you choose for a given task is a matter of convenience.

**Example 3.24** Practice Break

```
shell-prompt: printf "The quick brown fox jumped over the lazy dog." > fox.txt
shell-prompt: cat fox.txt
shell-prompt: tr '[:lower:]' '[:upper:]' < fox.txt | tr -s ' '
```

### 3.14.8 Find

The **find** command is a powerful tool for not only locating path names in a directory tree, but also for taking any desired action when a path name is found.

Unlike popular search utilities in macOS, Windows, and the Unix **locate** command. **find** does not use a previously constructed index of the file system, but searches the file system in its current state. Indexed search utilities very quickly produce results from a recent snapshot of the file system, which is rebuilt periodically by a scheduled job. This is much faster than an exhaustive search, but will miss files that were added since the last index build. The **find** command will take longer to search a large directory tree, but also guarantees accurate results.

The basic format of a find command is as follows:

```
shell-prompt: find top-directory search-criteria [optional-action \;]
```

The search-criteria can be any attribute of a file or other path name. To match by name, we use `-name` followed by a globbing pattern, in quotes to prevent the shell from expanding it before passing it to **find**. To search for files owned by a particular user or group, we can use `-user` or `-group`. We can also search for files with certain permissions, a minimum or maximum age, and many other criteria. The man page provides all of these details.

The default action is to print the relative path name of each match. For example, to list all the configuration files under `/etc`, we could use the following:

```
shell-prompt: find /etc -name '*.conf'
```

We can run any Unix command in response to each match using the `-exec` flag followed the command and a ';' or '+'. The ';' must be escaped or quoted to prevent the shell from using it as a command separator and treating everything after it as a new command, separate from the **find** command. The name of the matched path is represented by '{}'.

```
shell-prompt: find /etc -name '*.conf' -exec ls -l '{}' \;
```

With a ';' terminating the command, the command is executed immediately after each match. This may be necessary in some situations, but it entails a great deal of overhead from running the same command many times. Replacing the ';' with a '+' tells **find** to accumulate as many path names as possible and pass them all to one invocation of the command. This means the command could receive thousands of path names as arguments and will be executed far fewer times.

```
shell-prompt: find /etc -name '*.conf' -exec ls -l '{}' +
```

There are also some predefined actions we can use instead of spelling out a `-exec`, such as `-print`, which is the default action, and `-ls`, which is equivalent to `-exec ls -l '{}' +`. The `-print` action is useful for showing path names being processed by another action:

```
shell-prompt: find Data -name '*.bak' -print -exec rm '{}' +
```

Sometimes we may want to execute more than one command for each path matched. Rather than construct a complex and messy `-exec`, we may prefer to write a shell script containing the commands and run the script using `-exec`. Scripting is covered in Chapter 4.

**Example 3.25** Practice Break

```
shell-prompt: find /etc -name '*.conf' -exec ls -l '{}' +
```

### 3.14.9 Xargs

As stated earlier, most Unix commands that accept a file name as an argument will accept any number of file names. When processing 100 files with the same program, it is usually more efficient to run one process with 100 file name arguments than to run 100 processes with one argument each.

However, there is a limit to how long Unix commands can be. When processing many thousands of files, it may not be possible to run a single command with all of the filenames as arguments. The **xargs** command solves this problem by reading a list of file names from the standard input (which has no limit) and feeding them to another command as arguments, providing as many arguments as possible to each process created.

The arguments processed by **xargs** do not have to be file names, but usually are. The main trick generating the list of files. Suppose we want to change all occurrences of "fox" to "toad" in the files input*.txt in the CWD. Our first thought might be a simple command:

```
shell-prompt: sed -i '' -e 's|fox|toad|g' input*.txt
```

If there are too many files matching "input*.txt", we will get an error such as "Argument list too long". One might think to solve this problem using **xargs** as follows:

```
shell-prompt: ls input*.txt | xargs sed -i '' -e 's|fox|toad|g'
```

However, this won't work either, because the shell hits the same argument list limit for the **ls** command as it does for the **sed** command.

The **find** command can come to the rescue:

```
shell-prompt: find . -name 'input*.txt' | xargs sed -i '' -e 's|fox|toad|g'
```

Since the shell is not trying to expand '*.txt' to an argument list, but instead passing the literal string '*.txt' to **find**, there is no limit on how many file names it can match. The **find** command is sophisticated enough to work around the limits of argument lists.

The **find** command above will send relative path names of every file with a name matching 'input*.txt' in *and under* the CWD. If we don't want to process files in subdirectories of CWD, we can limit the depth of the find command to one directory level:

```
shell-prompt: find . -maxdepth 1 -name '*.txt' \
              | xargs sed -i '' -e 's|fox|toad|g'
```

**Note**
The **xargs** command places the arguments read from the standard input *after* any arguments included with the command. So the commands run by xargs will have the form

```
sed -i '' -e 's|fox|toad|g' input1.txt input2.txt input3.txt ...
```

Some **xargs** implementations have an option for placing the arguments from the standard input *before* the fixed arguments, but this is still limited. There may be cases where we want the arguments intermingled. The most portable and flexible solution to this is writing a simple script that takes all the arguments from **xargs** last, and constructs the appropriate command with the arguments in the correct order. Scripting is covered in Chapter 4.

Most xargs implementations also support running multiple processes at the same time. This provides a convenient way to utilize multiple cores to parallelize processing. If you have a computer with 16 cores and speeding up your analysis by a factor of nearly 16 is good enough, then this can be a very valuable alternative to using an HPC cluster. If you need access to hundreds of cores to get your work done in a reasonable time, then a cluster is a better option.

```
shell-prompt: find . -name '*.txt' \
              | xargs -P 8 sed -i '' -e 's|fox|toad|g'
```

A value of 0 following -P tells xargs to detect the number of available cores and use all of them. Some, but not all **xargs** implementations support `--max-procs` in place of `-P`. While using of long options is more readable, it is not portable in this instance.

There is a more sophisticated open source program called GNU parallel that can run commands in parallel in a similar way, but with more flexibility. It can be installed via most package managers. See Section 3.14.17 for an introduction.

**Example 3.26** Practice Break

```
shell-prompt: find /etc -name '*.conf' | xargs ls -l
```

### 3.14.10 Bc

The **bc** (binary calculator) command is an unlimited range and precision calculator with a scripting language very similar to C. When invoked with `-l` or `--mathlib`, it includes numerous additional functions including l(x) (natural log), e(x) (exponential), s(x) (sine), c(x) (cosine), and a(x) (arctangent). There are numerous standard functions available even without `--mathlib`. See the man page for a full list.

By default, **bc** prints the result of each expression evaluated followed by a newline. There is also a **print** statement that does not print a newline. This allows a line of output to be constructed from multiple expressions, the last of which includes a literal "\n".

```
shell-prompt: bc --mathlib
sqrt(2)
1.41421356237309504880

print sqrt(2), "\n"
1.41421356237309504880
```

```
e(1)
2.71828182845904523536

x=10
5 * x^2 + 2 * x + 1
521

quit
```

**Bc** is especially useful for quick computations where extreme range or precision is required, and for checking the results from more traditional languages that lack such range and precision. For example, consider the computation of factorials. N factorial, denoted N!, is the product of all integers from one to N. The factorial function grows so quickly that 21! exceeds the range of a 64-bit unsigned integer, the largest integer value supported by most CPUS and most common languages. The C program and output below demonstrate the limitations of 64-bit integers.

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    unsigned long   c, fact = 1;

    for (c = 1; c <= 22; ++c)
    {
        fact *= c;
        printf("%lu! = %lu\n", c, fact);
    }
    return EX_OK;
}
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768      This does not equal 20! * 21
22! = 17196083355034583040
23! = 8128291617894825984
24! = 10611558092380307456
25! = 7034535277573963776
```

At 21!, an integer overflow occurs. In the limited integer systems used by computers, adding 1 to the largest possible value produces a result of 0. The integer number sets used by computers are called *modular* number systems and are actually circular. The limitations of computer number systems are covered in Chapter 14.

In contrast, **bc** can compute factorials of any size, limited only by the amount of memory needed to store all the digits. It is, of course, much slower than C, both because it is an interpreted language and because it performs multiple precision arithmetic, which requires multiple machine instructions for every math operation. However, it is more than fast enough for many purposes and the easiest way to do math that is beyond the capabilities of common languages.

The **bc** script below demonstrates the superior range of **bc**. The first line (#!/usr/bin/bc -l) tells the Unix shell how to run the script, so we can run it by simply typing its name, such as **./fact.bc**. This will be covered in Chapter 4. For now, create the script using **nano fact.bc** and run it with **bc < fact.bc**.

```
#!/usr/bin/bc -l

fact = 1;
for (c = 1; c <= 100; ++c)
{
    fact *= c;
    print c, "!= ", fact, "\n";
}
quit
```

```
1!= 1
2!= 2
3!= 6
4!= 24
5!= 120
6!= 720
7!= 5040
8!= 40320
9!= 362880
10!= 3628800
11!= 39916800
12!= 479001600
13!= 6227020800
14!= 87178291200
15!= 1307674368000
16!= 20922789888000
17!= 355687428096000
18!= 6402373705728000
19!= 121645100408832000
20!= 2432902008176640000
21!= 51090942171709440000
22!= 1124000727777607680000
23!= 25852016738884976640000
24!= 620448401733239439360000
25!= 15511210043330985984000000

[ Output removed for brevity ]

100!= 93326215443944152681699238856266700490715968264381621468592963\
8952175999932299156089414639761565182862536979208272237582511852109\
6864000000000000000000000000
```

Someone with a little knowledge of computer number systems might think that we can get around the range problem in general purpose languages like C by using floating point rather than integers. This will not work, however. While a 64-bit floating point number has a much greater range than a 64-bit integer (up to $10^{308}$ vs $10^{19}$ for integers), floating point actually has less precision. It sacrifices some precision in order to achieve the greater range. The modified C code and output below show that the double (64-bit floating point) type in C only gets us to 22!, and round-off error corrupts 23! and beyond.

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])
```

```
{
    double  c, fact = 1;

    for (c = 1; c <= 25; ++c)
    {
        fact *= c;
        printf("%0.0f! = %0.0f\n", c, fact);
    }
    return EX_OK;
}
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 51090942171709440000
22! = 1124000727777607680000
23! = 25852016738884978212864
24! = 620448401733239409999872
25! = 15511210043330986055303168
```

**Example 3.27** Practice Break

```
shell-prompt: printf "sqrt(31.67)\nquit\n" | bc -l
```

### 3.14.11 Tar

The **tar** command, short for TApe Archive, is a tool for combining multiple files into one. Recall that Unix incorporates the idea of device independence, where an input/output device is treated exactly like an ordinary file. Originally, **tar** was meant to write the archive to a tape device, such as `/dev/tape`. This was a way to create backups for important files on removable tapes in case of a disk failure or other mishap.

Thanks to device independence, we can substitute any other device or ordinary file for `/dev/tape`. In modern times, backups are more often done over high-speed networks to sophisticated backup systems and **tar** is more often used to create *tarballs*, ordinary files containing archives for sharing whole directories. Most open source software is downloaded as a single tarball and unpacked on the local system.

The basic command template for creating a tarball is as follows:

```
shell-prompt: tar -cvf archive.tar path [path ...]
```

Archiving files this way has many potential advantages. It saves disk space, since each file has on average 1/2 of a disk block unused. Files can only allocate whole blocks and almost never have a size that is an exact multiple of the block size. Replacing many files with one archive reduces the size of the directory containing the files. Processing many small files (moving, transferring to another computer over a network, etc.) takes longer than processing one large file, since there is overhead for opening each file.

The `-c` flag means "Create". The `-v` means "Verbose" (echo each file name as it is added). The `-f` means "File name". If not provided, the default is the first tape device in `/dev`. The "path" arguments name files or directories to archive.

We can specify any number of files and directories, but the file name of the archive must come immediately after the `-f` flag.

**Note**
The **tar** command is one of the commands that predates the convention of using a '-' to indicate flags. Hence, you may see examples on the web such as:

```
tar cvf file.tar directory
```

To unpack a tarball, we use the `-x` flag, which means "eXtract".

```
shell-prompt: tar -xvf archive.tar
```

We can list the contents of a tarball using `-t`.

```
shell-prompt: tar -tf archive.tar
```

**Example 3.28** Practice Break

```
shell-prompt: cd
shell-prompt: mkdir Tempdir
shell-prompt: touch Tempdir/temp1
shell-prompt: touch Tempdir/temp2
shell-prompt: tar -cvf tempdir.tar Tempdir
shell-prompt: tar -tf tempdir.tar
shell-prompt: rm -rf Tempdir
shell-prompt: tar -xvf tempdir.tar
shell-prompt: ls Tempdir
```

### 3.14.12 Gzip, bzip2, xz

The **gzip** (GNU zip), **bzip2** (Burrows-Wheeler zip), and **xz** (LZMA zip) commands compress files in order to save disk space. In the most basic use, we run the command with a single file argument:

```
shell-prompt: gzip file
shell-prompt: bzip2 file
shell-prompt: xz file
```

This will produce a compressed output file with a ".gz", ".bz2", or ".xz" extension. The original file is automatically removed after the compressed file is successfully created.

The compressed files can be decompressed using companion commands to restore the original file. Compression is *lossless* (unlike JPEG), so the restored file will be identical to the original.

```
shell-prompt: gunzip file.gz
shell-prompt: bzip2 file.bz2
shell-prompt: xz file.xz
```

All three commands can be used as filters to directly compress output from another program:

```
shell-prompt: myanalysis | gzip > output.gz
shell-prompt: myanalysis | bzip2 > output.bz2
shell-prompt: myanalysis | xz > output.xz
```

Likewise, the decompression tools can send decompressed output to another program via a pipe. They also include analogs to the **cat** command for better readability:

```
shell-prompt: gunzip -c output.gz | more

shell-prompt: bunzip2 -c output.bz2 | more
shell-prompt: bzcat output.bz | more

shell-prompt: unxz -c output.xz | more
shell-prompt: xzcat output.xz | more
```

**Note**

For historical reasons, the portable command for viewing gzipped files is **zcat**, not **gzcat**. However, as of this writing, **zcat** on macOS looks for a ".Z" extension (from the outdated **compress** command), and only **gzcat** works with ".gz" files. Hence, **gunzip -c** is the most portable approach.

The choice between them is a matter of speed vs compression ratio. **Gzip** is generally the fastest, but achieves the least compression. **Xz** produces the best compression, but at a high cost in CPU time. **Bzip2** produces intermediate compression and is also CPU-intensive. All three compression tools allow the user to control the compression ratio in order to trade speed for compression. Lower values use less CPU time but to not compress as well.

```
shell-prompt: myanalysis | xz -3 > output.xz
```

If a program produces high-volume output (more than a few megabytes per second), some compression tools may not be able to keep up. You may want to use **gzip** and/or lower the compression level in these cases.

When archiving data for long-term storage, on the other hand, you will generally want the best possible compression and should not be too concerned about how long it takes. There are numerous websites containing benchmark data comparing the run time and compression of these tools with various compression levels. Such data will not be included in this guide as it is dated: it will change as the tools are continually improved.

Decompression is generally much faster than compression. While **xz** with medium to high compression levels requires a great deal of CPU time, **unxz** can decompress the data very quickly. Hence, if files need only be compressed once, but read many times, **xz** may be a good choice.

All three tools are integrated with **tar** in order to produce compressed tarballs. This can be done with a pipe by specifying "-" as the filename following `-f`, or using `-z, --gzip, --gunzip`, `-j, --bzip2, --bunzip2`, or `-J, --xz` with the tar command. The conventional file name extensions are ".tar.gz" or ".tgz" for **gzip**, ".tar.bz2" or ".tbz" for **bzip2**, and ".tar.xz" or ".txz" for **xz**.

```
shell-prompt: tar -cvf - Tempdir | gzip > tempdir.tgz
shell-prompt: tar -zcvf tempdir.tgz Tempdir

shell-prompt: tar -cvf - Tempdir | bzip2 > tempdir.tbz
shell-prompt: tar -jcvf tempdir.tbz Tempdir

shell-prompt: tar -cvf - Tempdir | xz > tempdir.txz
shell-prompt: tar -Jcvf tempdir.txz Tempdir
```

**Example 3.29** Practice Break

```
shell-prompt: cat | xz > test.xz
Type in some text, then press Ctrl+d.
shell-prompt: xzcat test.xz

shell-prompt: tar -Jcvf tempdir.txz Tempdir
```

### 3.14.13 Zip, unzip

**Zip** is both an archiver and compression tool in one. It was originally developed by Phil Katz, co-founder of PKZIP, Inc. in Milwaukee, WI in 1989, for MS-DOS. The zip format has become the standard for many other Windows-based archive tools. The compression algorithms have evolved significantly since the original PKZIP.

The **zip** and **unzip** commands are open source tools for creating and extracting .zip files. They are primarily for interoperability with Windows file archives and far less popular than tarballs compressed with **gzip**, **bzip2**, and **xz**.

### 3.14.14 Time

The **time** command runs another command under its supervision and measures *wall time*, *user time*, and *system time*. Wall time, also known as *real time*, is the elapsed in the world while a program is running. The term was coined at a time when most people had clocks on their walls, rather than relying on a smart phone. User time is the time spent using a core. If a program uses only one core (logical CPU), user time is less than wall time. If it uses more than one core, user time can exceed wall time. System time is the time spent by the operating system performing tasks on behalf of the process. Hence total CPU time is user time + system time.

The **time** command is used by simply prefixing any other Unix command with "time ". Some shells have an internal time command, which presents output in a different format than the external **time** command normally found in `/usr/bin`. The T shell internal **time** command also reports percent of CPU time used. Low CPU utilization generally indicates that the process was I/O-bound, i.e. it spent a lot of time waiting for disk or other input/output transactions and therefore was not utilizing the CPU. Also reported are memory use in kibibytes, a count of I/O operations, and page faults (where memory blocks are swapped to or from disk due to memory being full).

```
shell-prompt: time find /usr/local/lib > /dev/null
0.055u 0.094s 0:00.15 93.3%     43+179k 0+0io 0pf+0w

shell-prompt: /usr/bin/time find /usr/local/lib > /dev/null
        0.14 real         0.04 user         0.09 sys
```

Reported times will vary, usually by a fraction of a second, due to limited precision of measurement and other factors. It is usually fairly consistent for programs that use at least a few seconds of CPU time.

**Example 3.30** Practice Break

```
shell-prompt: time find /usr/local/lib > /dev/null
```

### 3.14.15 Top

The **top** command displays real-time information about currently running processes, sorted in order of resource use. It does not show information about *all* processes, but only the top resource users. Snapshots are reported every two seconds by default.

At the top of the screen is a summary of the system state, including *load average* (% of available cores in use), total processes running and sleeping (waiting for input/output), and a summary of memory (RAM and swap) use. *Swap* is an area of disk used to extend the amount of memory apparent to processes. Processes see the *virtual memory* size, which is RAM (electronic memory) + swap.

| **Tag** | **Meaning** |
|---|---|
| PID | The process ID |
| USERNAME | User owning the process |
| THR | Number of threads (cores used) |
| PRI | CPU scheduling priority |
| NICE | Nice value: Limits scheduling priority |
| SIZE | Virtual memory allocated |
| RES | Resident memory: Actual RAM (not swap) used |
| STATE | State of process at the moment of the last snapshot, such as running (using a core), waiting for I/O, select (waiting on any of multiple devices) pipdwt (writing to a pipe), nanslp (sleeping for nanoseconds), etc. |
| C | Last core on which it ran |
| TIME | CPU time accumulated so far |
| WCPU | Weighted CPU % currently using |
| COMMAND | Command executed, usually truncated |

Table 3.13: Column headers of top command

Below the system summary is information about the most active and resource-intensive processes currently running. Columns in the example below are summarized in Table 3.13.

Different operating systems will display slightly different information. There are many command-line flags to alter behavior, and behavior can be adjusted while running. Press 'h' for a help menu to see the options for altering output.

```
last pid: 70340;  load averages:  0.67,  0.34,  0.35; b up 3+03:11:57  08:57:32
61 processes:  3 running, 58 sleeping
CPU: 40.6% user,  0.0% nice,  2.2% system,  0.0% interrupt, 57.2% idle
Mem: 145M Active, 1871M Inact, 166M Laundry, 1210M Wired, 648M Buf, 4247M Free
Swap: 3852M Total, 3852M Free

  PID USERNAME    THR PRI NICE   SIZE    RES STATE    C   TIME    WCPU COMMAND
70338 bacon         1  79    0    13M  2160K CPU2     2   0:03  72.65% fastq-tr
70340 bacon         1  79    0    13M  3056K CPU1     1   0:03  72.15% gzip
70339 bacon         1  44    0    13M  2856K pipdwt   3   0:01  28.68% gunzip
69958 bacon         3  20    0   237M    92M select   2   0:02   0.54% coreterm
 9690 root          5  20    0   144M    80M select   0   5:08   0.23% Xorg
 9719 bacon         4  20    0   340M   132M select   1   3:42   0.12% lumina-d
70332 bacon         1  20    0    14M  3668K CPU0     0   0:00   0.05% top
 1644 root          1  20    0    13M  1656K select   0   0:58   0.01% powerd
27489 root         14 -44   r8    20M  7576K cuse-s   1   0:01   0.00% webcamd
 9756 bacon         1  20    0    51M    24M select   0   0:03   0.00% python3.
 1666 root          1  20    0    13M  1748K select   3   4:29   0.00% moused
 9716 bacon         1  20    0    27M    11M select   1   0:13   0.00% fluxbox
 1756 root          1  20    0    18M  3400K select   0   0:04   0.00% sendmail
 1315 root          1  20    0    11M  1020K select   2   0:02   0.00% devd
 9744 bacon         3  20    0   153M    48M select   2   0:02   0.00% python3.
 1495 root          1  20    0    13M  2100K select   1   0:02   0.00% syslogd
 1641 root          1  20    0    13M  1984K wait     1   0:01   0.00% sh
24775 bacon         4  20    0    34M  7220K select   3   0:01   0.00% at-spi2-
 9711 bacon         3  20    0    94M    21M select   2   0:01   0.00% start-lu
 1725 root          1  20    0    13M  1992K nanslp   3   0:01   0.00% cron
 1615 messagebus    1  20    0    14M  2860K select   0   0:01   0.00% dbus-dae
 1639 ntpd          1  20    0    21M  3308K select   3   0:01   0.00% ntpd
```

**Example 3.31** Practice Break

Run **top**, press 'h' to see the help screen, and press 'n' followed by '5' to make the screen less noisy.

### 3.14.16 Iostat

The **iostat** command displays information about disk activity and possibly other status information, depending on the flags used. Unfortunately, **iostat** is one of the rare commands that is not well-standardized across Unix systems. Check the man page on your system for details on all the flags. Here we show basic use for monitoring disk activity similarly to how we monitor CPU and memory use with **top**.

Low CPU utilization in **top** often indicates that a process is I/O-bound (e.g. spending a great deal of time waiting for disk operations). Processes go to sleep and do not use the CPU while waiting for disk and other input/output. To help verify this, we can check the STATE column in **top** as well. If it shows a state such as "wait", "select", or "pipe", then the process is waiting for I/O. Lastly, we can use **iostat** to see exactly how busy the disks are. This tells us nothing about a specific process, but we can generally deduce which processes are causing high disk activity.

The FreeBSD **iostat** offers concise output on a single line including the rates of tty (terminal) and disk throughput, and some CPU stats similar to **top**. We can request an update every N seconds by specifying `-w N` or simply `N`. The header is kindly reprinted when it is scrolled off the terminal.

```
FreeBSD shell-prompt: iostat 1
       tty            ada0              cd0            pass0             cpu
 tin  tout KB/t  tps  MB/s  KB/t  tps  MB/s  KB/t  tps  MB/s  us ni sy in id
   4   583 47.0    5   0.2   0.0    0   0.0   0.0    0   0.0   5  0  1  0 95
   1   537 1024   18  18.0   0.0    0   0.0   0.0    0   0.0  45  0  1  0 54

   [snip]

   0   733  988   18  17.4   0.0    0   0.0   0.0    0   0.0  42  0  2  0 56
   0   295 1024   18  18.0   0.0    0   0.0   0.0    0   0.0  42  0  2  0 55
       tty            ada0              cd0            pass0             cpu
 tin  tout KB/t  tps  MB/s  KB/t  tps  MB/s  KB/t  tps  MB/s  us ni sy in id
   0   300  927   21  19.0   0.0    0   0.0   0.0    0   0.0  45  0  1  0 54
   0   457  536   35  18.3   0.0    0   0.0   0.0    0   0.0  44  0  2  0 54
```

Apple's **iostat** is derived from FreeBSD's and has a similar output format and behavior.

```
macOS shell-prompt: iostat 1
              disk0       cpu    load average
    KB/t  tps  MB/s   us sy id   1m   5m   15m
   13.46    3  0.03    7  5 88  1.15 1.03 1.01
   11.97  289  3.38   40 13 47  1.15 1.03 1.01
    4.00    1  0.00    6  3 91  1.15 1.03 1.01
    0.00    0  0.00    0  2 98  1.15 1.03 1.01
    0.00    0  0.00    1  2 97  1.14 1.03 1.01
    4.25  145  0.60    9  6 85  1.14 1.03 1.01
```

The Linux **iostat** has significantly different options and output format. In addition, it may not be present on all Linux systems by default. On RHEL (Redhat Enterprise Linux), for example, we must install the sysstat package using the **yum** package manager. The output format contains multiple lines for each snapshot, but presents similar information.

```
RHEL shell-prompt: yum install -y sysstat
RHEL shell-prompt: iostat 1
Linux alma8.localdomain  bacon ~ 1001: (pkgsrc): iostat 1
Linux 4.18.0-372.26.1.el8_6.x86_64 (alma8.localdomain) 10/16/2022  _x86_64_(4 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.14    0.00    0.30    0.10    0.00   99.46

Device             tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda              11.57       250.38        34.24     443223      60607
scd0              0.01         0.00         0.00          1          0
dm-0             11.53       221.50        33.04     392110      58492
dm-1              0.06         1.25         0.00       2220          0
```

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.00    0.00    0.25    0.00    0.00   99.75

Device             tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda               0.00         0.00         0.00          0          0
scd0              0.00         0.00         0.00          0          0
dm-0              0.00         0.00         0.00          0          0
dm-1              0.00         0.00         0.00          0          0
```

### 3.14.17 GNU Parallel

GNU Parallel is a sophisticated open source tool for running multiple processes simultaneously. Users who do not have access to an HPC cluster for running large parallel jobs can at least utilize all the cores on their laptop or workstation using GNU parallel. GNU Parallel can be installed in seconds using most package managers.

In its simplest form, GNU parallel can be used as a drop-in replacement for xargs:

```
shell-prompt: find . -name 'input-*.txt' | xargs analyze
shell-prompt: find . -name 'input-*.txt' | parallel analyze
```

However, GNU parallel has many options for more sophisticated execution. The numerous use cases and syntax of GNU parallel are beyond the scope of this guide. There are many web tutorials and even books about GNU Parallel for details. If GNU parallel is properly installed on your system (i.e. via a package manager), you can begin by running **man parallel_tutorial**.

**Note**
The GNU parallel tutorial, at the time of this writing, contains some examples of overcomplicating simple tasks, such as the following:

```
# The tutorial recommends the following to generate sample input files:
shell-prompt: perl -e 'printf "A_B_C_"' > abc_-file
shell-prompt: perl -e 'for(1..1000000){print "$_\n"}' > num1000000

# In reality, perl serves no purpose in either case.
# We can just use the POSIX printf command:
shell-prompt: printf "A_B_C_" > abc_-file
shell-prompt: printf "%s\n" `seq 1 1000` > nums1000000
```

### 3.14.18 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a regular expression? Is it the same as a globbing pattern?
2. Show a Unix command that shows lines in analysis.c containing hard-coded floating point constants.
3. How can we speed up grep searches when searching for a fixed string rather than an RE pattern?
4. How can we use extended REs with grep?
5. How can we make the matched pattern visible in the grep output?
6. Describe two major differences between grep and awk.
7. How does awk compare to spreadsheet programs like LibreOffice Calc and MS Excel?

8. The /etc/group file contains colon-separated lines in the form groupname:password:groupid:members. Show an awk command that will print the groupid and members of the group "root".

9. A GFF3 file contains tab-separated lines in the form "seqid source feature-type start end score strand phase attributes". The first attribute for an exon feature is the parent sequence ID. Write an awk script that reports the seqid, start, end, strand, and parent for each feature of type "exon". It should also report the number of exons and the number of genes. To test your script, download Mus_musculus.GRCm39.107.chromosome.1.gff3.gz from ensembl.org and then do the following:

```
gunzip Mus_musculus.GRCm39.107.chromosome.1.gff3.gz
awk -f your-script.awk Mus_musculus.GRCm39.107.chromosome.1.gff3
```

10. Show a cut command roughly equivalent to the following awk command, which processes a tab-separated GFF3 file.

```
awk '{ print $1, $3, $4, $5 }' file.gff3
```

11. Show a sed command that replaces all occurrences of "wolf" with "werewolf" in the file halloween-list.txt.

12. Show a command to sort the following data by height. Show a separate command to sort by weight. The data are in params.txt.

```
ID  Height  Weight
1   34      10
2   40      14
3   29      9
4   28      11
```

13. Show a Unix command that reads the file fox.txt, replaces the word "fox" with "toad" and converts all lower case letters to upper case, and stores the output in big-toad.txt.

14. Show a Unix command that lists and removes all the files whose names end in '.o' in and under ~/Programs.

15. Why is the xargs command necessary?

16. Show a Unix command that removes all the files with names ending in ".tmp" only in the CWD, assuming that there are too many of them to provide as arguments to one command. The user should not be prompted for each delete. ( Check the **rm** man page if needed. )

17. Show a Unix command that processes all the files named 'input*' in the CWD, using as many cores as possible, through a command such as the following:

```
analyze --limit 5 input1 input2
```

18. What is the most portable and flexible way to use xargs when the arguments it provides to the command must precede some of the fixed arguments?

19. What is the major advantage of the **bc** calculator over common programming languages?

20. Show a bc expression that prints the value of the natural number, e.

21. Write a **bc** script that prints the following. Create the script with **nano sqrt.bc** and run it with **bc -l < sqrt.bc**.

```
sqrt(1)  = 1.00000000000000000000
sqrt(2)  = 1.41421356237309504880
sqrt(3)  = 1.73205080756887729352
sqrt(4)  = 2.00000000000000000000
sqrt(5)  = 2.23606797749978969640
sqrt(6)  = 2.44948974278317809819
sqrt(7)  = 2.64575131106459059050
sqrt(8)  = 2.82842712474619009760
sqrt(9)  = 3.00000000000000000000
sqrt(10) = 3.16227766016837933199
```

22. What are some advantages of archiving files in a tarball?
23. Show a Unix command that creates a tarball called research.tar containing all the files in the directory ./Research.
24. Show a Unix command that saves the output of **find /etc** to a compressed text file called `find-output.txt.bz2`.
25. Show a Unix command for viewing the contents of the compressed text file output.txt.gz, one page at a time.
26. Show a Unix command that creates a tarball called research.txz containing all the files in the directory ./Research.
27. What are **zip** and **unzip** primarily used for on Unix systems?
28. Show a Unix command that reports the CPU time used by the command **awk -f script.awk input.tsv**.
29. Show a Unix command that will help us determine which processes are using the most CPU time or memory.
30. How can we find out how to adjust the behavior of **top** while it is running?
31. What kind of output from **top** might suggest that a process is I/O-bound? Why?
32. Show a Unix command that continuously monitors total disk activity on a Unix system.
33. How can users who do not have access to an HPC cluster run things in parallel, in a more sophisticated way than possible with standard Unix tools such as xargs?

## 3.15 File Transfer

Many users will need to transfer data to or from remote servers. For example, we often want to analyze publicly available data hosted on a web server. Users of a shared research computer or HPC cluster running Unix may also need to transfer files from their computer to the Unix machine, run research programs, and finally transfer results back to their computer. There are many software tools available to accomplish this. Some of the convenient standard tools are described below.

### 3.15.1 Downloading Files with Curl, Fetch, and Wget

The **curl**, **fetch**, and **wget** commands are open source command-line tools for downloading files from a remote server. Most often, they serve the same purpose as a web browser. However, they allow us to automate the downloading of files when we know the *URL* (Uniform Resource Locator), also known as the web address. This is especially useful when we script an analysis that requires many files retrieved from one or more websites. Scripting is covered in Chapter 4.

The URL begins with a protocol indicator, such as "https:" or "ftp:". This is followed by the server name (such as those reported by the **hostname** command), and finally a path name on the remote server.

**Curl** is included in the default installation of some GNU/Linux operating systems and is easily installed via package managers on most other systems. Unlike other tools, it sends output to the standard output by default. To save the downloaded file using the same name as on the remote system, we need to add the `-O` (capital O) flag:

```
shell-prompt: curl -O http://ftp.ensembl.org/pub/release-107/gff3/homo_sapiens/Homo_sapiens ↩
    .GRCh38.107.chromosome.1.gff3.gz
```

**Fetch** is a somewhat simpler FreeBSD-specific tool included in the base system. The FreeBSD ports system is heavily dependent on fetch for automated downloading of files from various developer websites. Relying on the more complex and independently developed **curl** or **wget** would be riskier. When writing scripts that download files, it is generally better to use **curl** or **wget** so that the script will be portable to other systems that may not offer FreeBSD's fetch as a package. Fetch is mentioned here mainly as a fall-back option for researchers using FreeBSD.

```
shell-prompt: fetch http://ftp.ensembl.org/pub/release-107/gff3/homo_sapiens/Homo_sapiens. ↩
    GRCh38.107.chromosome.1.gff3.gz
```

**Wget** is fairly comparable to **curl** in its interface and capabilities. It is also included in the base install of some GNU/Linux operating systems and easily installed via most package managers on other systems.

```
shell-prompt: wget http://ftp.ensembl.org/pub/release-107/gff3/homo_sapiens/Homo_sapiens. ↩
    GRCh38.107.chromosome.1.gff3.gz
```

**Example 3.32** Practice Break

Run any or all of the sample commands shown above.

### 3.15.2 Pushing and Pulling Files with SFTP and Rsync

**SFTP** (Secure File Transfer Protocol) is often used to remotely log into another machine over a network for the purpose of transferring files to or from it. It is based on **ftp**, which should no longer be used, since it does not use encryption. Not all remote Unix systems have SFTP enabled.

**SFTP** provides a shell-like environment that allows us to list files and directories, cd into subdirectories, push (send, upload) files using **put** and pull (receive, download) files using **get**. It does not allow us to run programs on the remote system.

```
shell-prompt: sftp joe@unixdev1.ceas.uwm.edu
password: (Nothing is echoed when the password is typed)
Connected to unixdev1.ceas.uwm.edu.
sftp> ls
Data                    My Programs             Pictures
Qemu                    R                       STRESS
sftp> cd Data
sftp> ls
CNC-EMDiff   IRC
sftp> cd CNC-EMDiff/
sftp> ls
ATAC-Seq        Combined        Common          Misc            README.md
RNA-Seq         Raw             adapter-stats   backup.sh       todo
sftp> get backup.sh
Fetching /usr/home/bacon/Data/CNC-EMDiff/backup.sh to backup.sh
backup.sh                                    100%  166     3.4KB/s   00:00
sftp> exit
```

There are also graphical programs that use SFTP protocol, such as FileZilla. The vanilla **sftp** command and tools like FileZilla are convenient for small, simple transfers.

The **scp** command can be used to transfer files to any host that accepts **ssh** connections. This is a simple command with limited capabilities.

For more sophisticated and larger transfers from Unix to Unix (including Mac and Cygwin) users, the recommended transfer tool is **rsync**. The **rsync** command is a simple but intelligent tool that makes it easy to synchronize two directories on the same machine or on different machines across a network. **Rsync** is free software and part of the base installation of many Unix systems including macOS. On Cygwin, you can easily add the rsync package using the Cygwin Setup utility. **Rsync** has some major advantages over other file transfer programs:

- Unlike GUI tools, it can be scripted to automate file transfers as part of an analysis. Scripting is covered in Chapter 4.
- If you have transferred a directory before, and only want to synchronize the destination with the latest changes, **rsync** will automatically determine the differences between the two copies and only transfer what is necessary. When conducting research that generates large amounts of data, this can save an enormous amount of time.
- If a transfer fails for any reason (which is fairly common for large transfers due to network hiccups, etc), the inherent ability to determine the differences between two copies allows **rsync** to resume from where it left off. Simply run the exact same rsync command again, and the transfer will resume.

**Rsync** can push (send, upload) files from the local machine to a remote machine, or pull (retrieve, download) files from a remote machine to the local machine. The command syntax is basically the same in both cases. It's just a matter of how you specify the source and destination for the transfer.

The rsync command has many options, but the most typical usage is to create an exact copy of a directory on a remote system. The general rsync command to push a new directory or just changes to another host would be:

```
shell-prompt: rsync -av --delete source-path [username@]hostname:[destination-path]
```

**Example 3.33** Pushing data with rsync

The following command synchronizes the directory `Project` from the local machine to `~joeuser/Data/Project` on Peregrine:

```
shell-prompt: rsync -av --delete Project joeuser@unixdev1.ceas.uwm.edu:Data
```

The general syntax for pulling files from another host is:

```
shell-prompt: rsync -av --delete [username@]hostname:[source-path] destination-path
```

**Example 3.34** Pulling data with rsync

The following command synchronizes the directory ~joeuser/Data/Project on Peregrine to ./Project on the local machine:

```
shell-prompt: rsync -av --delete joeuser@unixdev1.ceas.uwm.edu:Data/project .
```

Note that the only difference between a push and a pull is which argument contains "[user@]hostname:".

This syntax, using a single colon (:) following the host name, tells rsync to use an *ssh tunnel*. This means that **ssh** is used to establish a secure connection, and **rsync** uses that connection to transfer files. Hence, all traffic, including username and password, is encrypted. **Rsync** can use other connection protocols, but **ssh** is the most common.

If you omit "username@" from the source or destination, **rsync** will try to log into the remote system with your username on the local system.

If you omit "destination-path" in an **rsync** push command or "source-path" in a pull command, **rsync** will place the source directory under your home directory on the remote host.

The command-line flags used above have the following meanings:

**-a, --archive** Use *archive* mode, equivalent to `-rlptgoD`. Archive mode copies all subdirectories recursively and preserves as many file attributes as possible, such as ownership, permissions, etc.

**-v, --verbose** Verbose copy: Display names of files and directories as they are copied.

**--delete** Delete files and directories from the destination that do not exist in the source. Without `--delete`, **rsync** will add and replace files in the destination, but never remove anything. This is a good strategy when using **rsync** to create backups of important files.

**Caution**

Note that a trailing "/" on source-path affects where rsync stores the files on the destination system. Without a trailing "/", rsync will create a directory called "source-path" under "destination-path" on the destination host.
With a trailing "/" on source-path, destination-path is assumed to be the directory that will replace source-path on the destination host. This feature is a somewhat cryptic method of allowing you to change the name of the directory during the transfer. It is compatible with the behavior of the Unix **cp** command.
Note also that the trailing "/" only affects the command when applied to source-path. A trailing "/" on destination-path has no effect.

The command below creates an identical copy of the directory `Model` in `~/Data/Model` on unixdev1.ceas.uwm.edu. The resulting directory is the same regardless of whether the destination directory existed before the command or not.

```
shell-prompt: rsync -av --delete Model joeuser@unixdev1.ceas.uwm.edu:Data
```

The command below dumps the *contents* of the local `Model` directly into `~/Data` on unixdev1, and deletes everything else in the Data directory! In other words, it makes the destination directory `~Data` identical to the local directory `Model`.

**Caution**
Carelessness with **rsync** can be very dangerous!

```
shell-prompt: rsync -av --delete Model/ joeuser@unixdev1.ceas.uwm.edu:Data
```

Note that if using globbing to specify files to pull from the remote system, any globbing patterns must be protected from expansion by the local shell by escaping them or enclosing them in quotes. We want the pattern expanded on the remote system, not the local system:

```
shell-prompt: rsync -av --delete joeuser@unixdev1.ceas.uwm.edu:Data/Study\* .
shell-prompt: rsync -av --delete 'joeuser@unixdev1.ceas.uwm.edu:Data/Study*' .
```

**Example 3.35** Practice Break

If you have access to a remote Unix system, run the following commands, replacing "unixdev1.ceas.uwm.edu" with "your-username@your-remote-hostname".

```
shell-prompt: mkdir -p Temp
shell-prompt: touch Temp/temp1.txt Temp/temp2.txt
shell-prompt: rsync -av Temp unixdev1.ceas.uwm.edu:
shell-prompt: ssh unixdev1.ceas.uwm.edu ls Temp
shell-prompt: rm Temp/temp2.txt
shell-prompt: rsync -av Temp unixdev1.ceas.uwm.edu:
shell-prompt: ssh unixdev1.ceas.uwm.edu ls Temp
```

**Rsync** can also be used to copy files locally, though creating multiple copies of a file on the same computer is generally senseless. If you don't have access to a remote Unix system, you can use the commands below to practice **rsync**.

```
shell-prompt: mkdir -p Temp
shell-prompt: touch Temp/temp1.txt Temp/temp2.txt
shell-prompt: rsync -av Temp Temp2
shell-prompt: ls Temp2
shell-prompt: rm Temp/temp2.txt
shell-prompt: rsync -av Temp Temp2
shell-prompt: ls Temp2
shell-prompt: rm -r Temp2
```

### 3.15.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are three commands we can use in place of a web browser to download files? Which should we generally use in scripts that need to be portable? Why?
2. Name three Unix commands that can be used to transfer files between two systems.
3. Describe three advantages of **rsync** over other file transfer tools.
4. What is the meaning of a trailing '/' on the source directory?
5. Show an **rsync** command that makes the directory `~/Data/Study1` on unixdev1.ceas.uwm.edu identical to `MyStudy` on the local machine.
6. Show an **rsync** command that makes the local directory `MyStudy` identical to `~/Data/Study1` on unixdev1.ceas.uwm.edu.

## 3.16 Environment Variables

Every Unix process maintains a list of variables called the *environment*. When a new process is created, it inherits the environment from the process that created it (its parent process). For typical Unix commands, the parent is usually the shell process.

All environment variables are character strings, i.e. sequences of characters. There are no other data types such as integer, float, Boolean, etc. There are some shell features for treating variables as numbers, but their values are always stored as character strings. For this reason, numeric operations in the shell are very inefficient.

Since the shell creates a new process whenever you run an external command, the shell's environment can be used to pass information to any command that you run. For example, text editors, **top**, **ls** with colorized output, and other programs that manipulate the terminal screen, need to know what type of terminal you are using. Different types of terminals use different *magic sequences* to move the cursor, change the foreground or background color, clear the screen, scroll, etc. To provide this information, we set the shell's environment variable `TERM` to the terminal type (usually "xterm"). When you run a command from the shell, the new process inherits the shell's `TERM` variable, and uses it to look up the correct magic sequences for your terminal type.

`PATH` is another important environment variable which specifies a list of directories containing external Unix commands. When you type a command at the shell prompt, the shell checks the directories listed in `PATH` in order to find the command you typed. For example, when you type the **ls** command, the shell utilizes `PATH` to locate the program in `/bin/ls`.

The directory names within in PATH are separated by colons. A simple value for `PATH` might be `/bin:/usr/bin:/usr/local/bin`. When you type **ls**, the shell first checks for the existence of `/bin/ls`. If it does not exist, the shell then checks for `/usr/bin/ls`, and so on, until it either finds the program or has checked all directories in `PATH`. If the program is not found, the shell issues an error message such as "ls: Command not found".

The **printenv** shows all of the environment variables currently set in your shell process.

```
shell-prompt: printenv
BLOCKSIZE=K
COLORTERM=xterm-256color
DISPLAY=:0
HOME=/home/bacon
LANG=C.UTF-8
LOGNAME=bacon
PWD=/home/bacon
SHELL=/bin/tcsh
TERM=xterm-256color
USER=bacon
...
```

Setting environment variables requires a different syntax depending on which shell you are using. Most modern Unix shells are extensions of either Bourne shell (sh) or C shell (csh), so there are only two variations of most shell commands that we need to know for most purposes.

For Bourne shell derivatives (sh, bash, dash, ksh, zsh), we use the **export** command:

```
shell-prompt: TERM=xterm
shell-prompt: PATH='/bin:/usr/bin:/usr/local/bin'
shell-prompt: export TERM PATH
```

**Note** There cannot be any space before or after the '=', for reasons that will be clarified later.

For C shell derivatives (csh, tcsh), we use **setenv**:

```
shell-prompt: setenv TERM xterm
shell-prompt: setenv PATH '/bin:/usr/bin:/usr/local/bin'
```

**Note** **Setenv** requires a space, not an '=', between the variable and the value.

The **env** can be used to alter the environment just for the invocation of one child process, rather than setting it for the current shell process. Suppose Bob has a program called **rna-trans** that uses *OpenMP threads* (multiple subprocesses running on separate cores to speed up the program) and he would like to run it using two cores. OpenMP normally uses all available cores, but we can limit it by setting the environment variable `OMP_NUM_THREADS`.

```
shell-prompt: env OMP_NUM_THREADS=2 rna-trans
```

Here, the **env** command sets `OMP_NUM_THREADS`, and then runs **rna-trans**. Since the **rna-trans** process is a child of the **env** process, it inherits the entire environment, including `OMP_NUM_THREADS`.

You can create environment variables with any name and value you like. However, there are many environment variable names that are reserved for specific purposes. A few of the most common ones are listed in Table 3.14.

| **Name** | **Purpose** |
|---|---|
| TERM | Terminal type for an interactive shell session |
| USER | User's login name |
| HOME | Absolute path of the user's home directory (~) |
| PATH | List of directories searched for commands |
| LANG | Character set for the local language |
| EDITOR | User's preferred interactive text editor |

Table 3.14: Reserved Environment Variables

**Example 3.36** Practice Break

```
shell-prompt: printenv | fgrep TERM
```

### 3.16.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the environment in Unix?
2. What data types are available for environment variables?
3. Does a Unix process have any environment variables when it starts? If so, where do they come from?
4. What is the purpose of the TERM environment variable? What kinds of programs make use of it?
5. What is the purpose of the PATH environment variable? What kinds of programs make use of it?
6. Show how to set the environment variable TERM to the value "xterm" in
   (a) Bourne shell (sh)
   (b) Korn shell (ksh)
   (c) Bourne again shell (bash)
   (d) C shell (csh)
   (e) T-shell (tcsh)
7. Show a Unix command that runs **ls** with the LSCOLORS environment variable set to "CxFxCxDxBxegedaBaGaCaD". You may not change the LSCOLORS variable for the current shell process.

## 3.17 Shell Variables

In addition to the environment, shells maintain a similar set of variables for their own use. These variables are not passed down to child processes, and are only used by the shell.

Like environment variables, you can create shell variables with any name and value you like. However, there are many shell variable names that are reserved for specific purposes. For example, each shell uses a special shell variable to define the shell prompt.

In Bourne-shell derivatives, this variable is called PS1. To set a shell variable in Bourne-shell derivatives, we use a simple assignment. The export command in the previous section actually sets a shell variable called TERM and then exports (copies) it to the environment.

```
shell-prompt: PS1="peregrine: "
```

---

**Note** Again, there cannot be any white space before or after the '='. This is how Bourne shell and its derivatives distinguish a variable assignment from a command. If we wrote **PS1 = "unixdev1: "**, the shell would think that PS1 is a command, '=' is the first argument, and "unixdev1: " is the second argument. PS1 is a variable, not a command.

---

In C shell derivatives, the variable is called `prompt`, and is set using the **set** command:

```
shell-prompt: set prompt="unixdev1: "
```

---

**Note** The syntax for **set** is slightly different than for **setenv**. **Set** uses an '=' while **setenv** uses a space. Unlike a Bourne shell variable assignment, the **set** command can tolerate white space around the '='.

---

Shell prompt variables may contain certain special symbols to represent dynamic information that you might want to include in your shell prompt, such as the host name, command counter, current working directory, etc. Consult the documentation on your shell for details.

In all shells, you can view the current shell variables by typing **set** with no arguments:

```
shell-prompt: set
```

Besides special variables like `PS1` and `prompt`, shell variables are primarily used in shell scripts, much like variables in a C or Java program. Shell scripts are covered in Chapter 4.

### 3.17.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

---

1. What is the difference between shell variables and environment variables?
2. Show how to set the shell prompt to "Unixdev1: " in:
   (a) Bourne shell
   (b) C shell
3. How can you view a list of all current shell variables and their values?

## 3.18 Process Control

Unix systems provide many tools for managing and monitoring processes that are already running. It is possible to have multiple processes running under the same shell session. Such processes are considered either *foreground processes* or *background processes*. The foreground process is simply the process that receives the keyboard input. There can be no more than one foreground process under a given shell session, since keyboard input cannot be sent to multiple processes at once.

However, all processes, both foreground and background, are allowed to send output to the terminal at the same time. It is up to the user to ensure that output is managed properly and not intermixed. Generally it does not make sense to have multiple processes sending output to the same terminal, but it is not forbidden. This is another example of Unix "staying out of the way" rather than enforcing rules that may not always be reasonable.

There are three types of tools for process management, described in the following subsections.

### 3.18.1 External Commands

Unix systems provide a variety of external commands that monitor or manipulate processes based on their process ID (PID). A few of the most common commands are described below.

**ps** lists the currently running processes.

```
shell-prompt: ps [-a]     # BSD
shell-prompt: ps [-e]     # SYSV
```

**ps** is one of the rare commands whose options vary across different Unix systems. There are only two standards to which it may conform, however. The BSD version uses `-a` to indicate that all processes (not just your own) should be shown. System 5 (SYSV) **ps** uses `-e` for the same purpose. Most modern systems accept the BSD standard. Some, such as most GNU/Linux systems, accept either. Run **man ps** on your system to determine which flags should be used.

**kill** sends a signal to a process, which may kill the process, but could serve other purposes. We may want to kill a process after noticing that it is producing incorrect output or has been running too long, indicating that something is wrong. The basic form of the command is:

```
shell-prompt: kill [-#] PID
```

The PID (process ID) is often determined from the output of **ps**.

```
shell-prompt: ps
  PID  TT  STAT      TIME COMMAND
 41167   0  Is     0:00.25 tcsh
 78555   0  S+     0:01.98 fdtd
shell-prompt: kill 78555
```

The signal number is an integer value or signal name (minus the SIG prefix) following a -, such as `-9` or `-SIGKILL`. If not provided, the default signal sent is SIGTERM (terminate), which is signal 15. Run **man signal** to learn about all the signals that can be issued with **kill**.

Processes can choose to ignore the SIGTERM signal, or respond in a different way than just terminating. Such processes can be force killed using the SIGKILL (9) signal.

```
shell-prompt: kill -9 78555
shell-prompt: kill -KILL 78555
```

The **pkill** command will kill all processes running the program named as the argument. This eliminates the need to find the PID first, and is more convenient for killing multiple processes running the same program.

```
shell-prompt: pkill fdtd
```

### 3.18.2 Special Key Combinations

Ctrl+c sends a SIGINT signal to the current foreground process. It is equivalent to **kill -INT PID**. This usually terminates the process immediately, although it is possible that some processes will ignore the signal, as stated earlier.

Ctrl+z sends a stop (SIGSTSTP) signal to the current foreground process. The process remains in memory, but does not execute further until it receives a continue (SIGCONT) signal (usually sent by running **fg**).

Ctrl+s suspends output to the terminal. This does not signal the process directly, but has the effect of blocking any processes that are sending output, since they cannot complete the output operation.

Ctrl+q allows output to the terminal to resume, if it has been suspended by **Ctrl+s**.

**Example 3.37** Practice Break

Run **find /**, then press Ctrl+s to suspend output, press Ctrl+q to resume it, press Ctrl+z to stop the process, run **fg** to continue the process, and press Ctrl+c to terminate the process.

### 3.18.3 Shell Features for Job Control

An & at the end of any command causes the command to be immediately placed in the background. It can be brought to the foreground using **fg** at any time. Normally, we do not want background processes sending any output to the terminal, so redirection is usually used at the same time.

**jobs** lists the processes running under the current shell, but using the shell's job IDs instead of the system's process IDs.

```
shell-prompt: find / >& output.txt &
[1] 89227
shell-prompt: jobs
[1]  + Running                       find / >& output.txt
```

**fg** brings a background job into the foreground. It optionally takes a shell job number as an argument. This only matters if there are multiple background jobs running. It can now be terminated using Ctrl+c. There cannot be another job already running in the foreground. If no job ID is provided, and multiple background jobs are running, the shell will choose which background job to bring to the foreground. A job ID should always be provided if more than one background job is running.

```
shell-prompt: fg 1
find / >& output.txt
^C
shell-prompt:
```

**bg** resumes a stopped job, such as a foreground job stopped by Ctrl+z, but in the background.

```
shell-prompt: find / >& output.txt
/
/.snap
/dev
/dev/log
/dev/dumpdev
/dev/cuse
Ctrl+z
shell-prompt: bg
shell-prompt:
```

**nice** runs a process with limited priority. We use this when we are not concerned about how fast the job runs and want other processes to have more CPU time, so they can complete sooner or respond to user input more quickly.

```
shell-prompt: nice command
```

If (and only if) other processes in the system are competing for CPU time, they will get a bigger share than processes run under **nice**.

**nohup** allows you to run a command that will continue after you log out. Naturally, all input and output must be redirected away from the terminal in order for this to work.

Bourne shell and compatible:

```
shell-prompt: nohup ./myprogram < inputfile > outputfile 2>&1
```

C shell and compatible:

```
shell-prompt: nohup ./myprogram < inputfile >& outputfile
```

This is often useful for long-running commands and where network connections are not reliable, or you simply don't want to remain logged in until it's finished.

There are also free add-on programs such as GNU screen that allow a session to be resumed if it's disrupted for any reason.

### 3.18.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the difference between a foreground process and a background process? How many of each can be running under a given shell process?
2. How do we keep the output of many background processes from mixing together?
3. Show a Unix command that terminates process 8210 if **kill 8210** has no effect.
4. What can we do if we have many processes running a program called **fastqc** and we want to terminate all of them?
5. What is the easiest way to terminate the foreground process?
6. How can we temporarily stop the foreground process, run **ls** under the same shell, and then continue the previous process.
7. How can we pause output from processes in a terminal, inspect it, and then allow it to continue?
8. How can we stop the current foreground process and resume it as a background process?
9. Show a Unix command that runs ./analysis so that its process uses less CPU time than other processes.
10. Show a Unix command that runs ./analysis so that it will continue running even after we log out.

## 3.19 Remote Graphics

### 3.19.1 Background

Most users will not need to run graphical applications on a remote Unix system.. If you know that you will need to use a graphical user interface with your research software, or if you want to use a graphical editor such as eclipse or emacs on over the network, read on. Otherwise, you can skip this section for now.

Unix uses a networked graphics interface called the X Window system. It is also sometimes called simply X11 for short. ( X11 is the latest major version of the system. ) X11 allows programs running on a remote Unix system to display graphics on your screen. The programs running on the remote system are called *clients*, and they display graphical output by sending commands (magic sequences) such as "draw a line from (x1,y1) to (x2,y2)" to the *X11 server* on the machine where the output is to be displayed. The computer in front of you must be running an X server process in order to display Unix graphics, regardless of whether the client programs are running on your machine or a remote machine.

Apple's macOS, while Unix compatible at the API and command-line, does not include X11 by default. It instead uses Apple's proprietary GUI. X11 for macOS is provided by the XQuartz project: https://www.xquartz.org/. XQuartz is free open source software (FOSS) that can be downloaded and installed in a few minutes. It should start automatically when either local or remote X11 clients attempt to access your display, but you can also start it manually.

### 3.19.2 Configuration Steps Common to all Operating Systems

Modern Unix systems such as BSD, Linux, and macOS have most of the necessary tools and configuration in place for running remote graphical applications. Some Unix servers may be configured without a GUI. If you want to remotely log into such a server and run X11 programs from your desktop or laptop system, you will need to at least install the **xauth** package on the remote system. This allows your system to configure X11 permissions for the remote system when you log in using **ssh -X** or **ssh -Y**.

```
# Debian
shell-prompt: apt install xauth

# FreeBSD
shell-prompt: pkg install xauth

# RHEL
shell-prompt: yum install xorg-x11-xauth
```

Some additional steps may be necessary on your computer to allow remote systems to access your display. This applies to *all* computers running an X11 server, regardless of operating system. Additional steps that may be necessary for Cygwin systems are discussed in Section 3.19.3.

If you want to run graphical applications on a remote computer over an ssh connection, you will need to forward your local display to the remote system. This can be done for a single ssh session by providing the `-X` flag:

```
shell-prompt: ssh -X joe@unixdev1.ceas.uwm.edu
```

This causes the **ssh** command to inform the remote system that X11 graphical output should be sent to your local display through the **ssh** connection. ( This is called SSH tunneling. )

**Caution** Allowing remote systems to display graphics on your computer can pose a security risk. For example, a remote user may be able to display a false login window on your computer in order to trick you into giving them your login and password information.

If you want to forward X11 connections to all remote hosts for all users on the local system, you can enable X11 forwarding in your `ssh_config` file (usually found in `/etc` or `/etc/ssh`) by adding the following line:

```
ForwardX11 yes
```

**Caution** Do this only if you are prepared to trust all users of your local system as well as all remote systems to which they might connect.

Some X11 programs require additional protocol features that can pose more security risks to the client system. If you get an error message containing "Invalid MIT-MAGIC-COOKIE" when trying to run a graphical application over an **ssh** connection, try using the `-Y` flag instead of `-X` to open a *trusted* connection.

```
shell-prompt: ssh -Y joe@unixdev1.ceas.uwm.edu
```

You can establish trusted connections to *all* hosts by adding the following to your `ssh_config` file:

```
ForwardX11Trusted yes
```

**Caution** This is generally considered a bad idea, since it states that every host we connect to from this computer to should be trusted completely. Since you don't know in advance what hosts people will connect to in the future, this is a huge leap of faith.

If you are using ssh over a slow connection, such as home DSL/cable, and plan to use X11 programs, it can be very helpful to enable compression, which is enabled by the `-C` flag. Packets are then compressed before being sent over the wire and decompressed on the receiving end. This adds more CPU load on both ends, but reduces the amount of data flowing over the network and may significantly improve the responsiveness of a graphical user interface.

```
shell-prompt: ssh -C -X joe@unixdev1.ceas.uwm.edu
```

**Caution** Using `-C` over a fast connection, such as a gigabit network, may actually slow down the connection, since the CPU may not not be able to compress data fast enough to use all of the network bandwidth.

### 3.19.3 Graphical Programs on Windows with Cygwin

It is possible for Unix graphical applications on the remote Unix machine to display on a Windows machine with Cygwin, but this will require installing additional Cygwin packages and performing a few configuration steps on your computer in addition to those discussed in Section 3.19.2.

#### Installation

You will need to install the x11/xinit and x11/xhost packages using the Cygwin setup utility. This will install a basic X11 server on your Windows machine.

#### Configuration

After installing the Cygwin X packages, there are additional configuration steps:

1. Create a working `ssh_config` file by running the following command from a Cygwin shell window:

   ```
   shell-prompt: cp /etc/defaults/etc/ssh_config /etc
   ```

2. Then, using your favorite text editor, update the new `/etc/ssh_config` as described in Section 3.19.2.
3. Add the following line to .bashrc or .bash_profile (in your home directory):

   ```
   export DISPLAY=":0.0"
   ```

   Cygwin uses bash for all users by default. If you are using a different shell, then edit the appropriate start up script instead of .bashrc or .bash_profile. For **tcsh**, add the following to your `.cshrc` or `.tcshrc`:

   ```
   setenv DISPLAY ":0.0"
   ```

   This is not necessary when running commands from an xterm window (which is launched from Cygwin-X), but *is* necessary if you want to launch X11 applications from the Cygwin terminal which is part of the base Cygwin installation, and not X11-aware.

#### Start-up

To enable X11 applications to display on your Windows machine, you need to start the X11 server on Windows by clicking Start → All Programs → Cygwin-X → XWin Server. The X server icon will appear in your Windows system tray to indicate that X11 is running. You can launch an xterm terminal emulator from the system tray icon, or use the Cygwin bash terminal, assuming that you have set your DISPLAY variable as described above.

### 3.19.4 Remote 3D Graphics

It is possible to run Open 3D graphical applications on remote systems as well, but performance may or may not be acceptable. There are also numerous problems that can arise that depend on the operating system and video drivers used on each end. In any case, OpenGL applications will require additional X11 components to be installed on both the remote machine running the application and the display machine running the X11 server. Determining the minimal set of packages required for every platform would be excessively difficult, so we recommend simply install the entire Xorg system on the remote system. This will likely enable at least some OpenGL applications to run remotely. This proved sufficient to run the 3D mesh visualizer MeshLab comfortably on a remote FreeBSD machine from a FreeBSD display. A similar application called VMD proved too sluggish to use remotely.

```
# Debian
shell-prompt: apt install xorg

# FreeBSD
shell-prompt: pkg install xorg
```

You may find it easier to simply download the data files to your local machine and run the 3D graphics applications there. Performance will be much better when using *direct rendering*, where the display is on the same machine running the 3D application. When using *indirect rendering*, where the 3D application is running on a different machine than the display, sending graphics commands over a network can be a bottleneck.

### 3.19.5 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is X11?
2. Does Apple's macOS use X11? Explain.
3. What must be installed at minimum on a remote computer to allow X11 client programs to run there and display graphics on the X11 display in front of you?
4. Show a Unix command that logs into unixdev1.ceas.uwm.edu and allows us to run remote graphical programs.
5. Show a Unix command that logs into unixdev1.ceas.uwm.edu and allows us to run remote graphical programs over a slow network.
6. Why is setting ForwardX11Trusted not generally a good idea?
7. What packages are needed on a Cygwin setup to enable X11?
8. What alternative do you have if running a 3D graphics program remotely proves to be too complicated or slow? How will this help?

# Chapter 4

# Unix Shell Scripting

**Before You Begin**
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 3) and the Unix shell (Section 3.3.3).

## 4.1 What is a Shell Script?

A shell script is essentially a file containing a sequence of Unix commands. A script is a type of program, but is distinguished from other programs in that it represents programming at a higher level. C programs are made up of C statements and calls to subprograms. Shell scripts are made up of shell commands and calls to C programs and other programs. In other words, entire programs serve as the subprograms in a shell script. A script is a way of automating the execution of multiple separate programs in sequence.

All Unix shells share a feature that can help us avoid this repetitive work: They don't care where their input comes from. It is often said that the Unix shell reads commands from the keyboard and executes them. This is not true. The shell reads commands from *any input source* and executes them. The keyboard is just one of many sources of commands that can be used by the shell. Ordinary files are also very commonly used as shell input.

**Note** About the only difference between a shell process reading commands from the keyboard and one reading commands from a file is that the process reading from a file does not print a shell prompt. Otherwise, they do not behave any differently. The commands we put in a script are exactly the same as the commands we would run interactively.

Recall from Chapter 3 that Unix systems employ device independence, which means that a keyboard is the same thing as a file from the perspective of a Unix program. Any program that reads from a keyboard can also read the same input from a file or any other input device.

The Unix command-line structure was designed to be convenient for both interactive use and for programming in scripts. In fact, a Unix command looks a lot like a subprogram call. The difference is just minor syntax. A subprogram call in C encloses the arguments in parenthesis and separates them with commas:

```
function_name(arg1,arg2,arg3);
```

A Unix command is basically the same, except that it uses spaces instead of parenthesis and commas and does not use parenthesis:

```
command_name arg1 arg2 arg3
```

It is important to understand the difference between a "script" and a "real program", and which languages are appropriate for each. Scripts tend to be small, usually a few lines to a few hundred lines, and do not do any significant computation of their own. Instead, scripts run other programs to do most of the computational work. The job of the script is simply to automate and document the process of running programs.

As a result, scripting languages do not need to be fast and are generally interpreted rather than compiled. Recall that interpreted language programs run orders of magnitude slower than equivalent compiled programs. Programs written in more general-purpose languages such as C, C++, or Java, may be quite large and may implement complex computational algorithms. Hence, they need to be fast and as a result are usually written in compiled languages.

If you plan to use exclusively pre-existing programs such as Unix commands and/or add-on application software, and need only automate the execution of these programs, then you need to write a script and should choose a good scripting language. If you plan to implement your own algorithm(s) that may require a lot of computation, then you need to write a program and should select an appropriate compiled programming language.

### 4.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a shell script?
2. Compare and contrast Unix commands with subprogram calls in a C or Java program.
3. What is the difference between a script and a "real" program?
4. Is a scripting language a good choice for performing matrix multiplication? Why or why not?

## 4.2 Why Write Shell Scripts?

### 4.2.1 Efficiency and Accuracy

Any experienced computer user knows that we often end up running basically the same sequence of commands many times over. Typing the same sequence of commands over and over is a waste of time and highly prone to errors.

Hence, if we're going to run the same sequence of commands more than once, we don't need to retype the sequence each time. The shell can read the commands from anywhere, and the keyboard is about the worst possible choice in this situation. We can put the same sequence of commands into a text file *once* and tell the shell to read the commands from the file as many times as we want, which is much easier than typing them all repeatedly, and eliminates the need to remember the details of the commands.

**Rule of Thumb**
If you're not completely certain that you will never need to do it again, script it.

In theory, Unix commands could also be piped in from another program or read from any other device attached to a Unix system, although in practice, they usually come from the keyboard or a script file.

### 4.2.2 Documentation

There is another very good reason for writing shell scripts in addition to saving us a lot of redundant typing. A shell script is the ultimate documentation of the work we have done on a computer, ensuring repeatability of the analysis, which is one of the cornerstones of science. By writing a shell script, we record the exact sequence of commands needed to reproduce results, in perfect detail. Hence, the script serves a dual purpose of automating and documenting our processes.

Developing a script has a ratchet effect on your knowledge. Once you add a command to a script, you will never again have to figure out how to do the same thing. Clear documentation of our work flow is important in order to justify research funding and to be able to reproduce results months or years later.

**Rule of Thumb**
Scientists and Unix users should never find themselves trying to remember how they did something. Script it the first time and you will never be in this situation. Unix makes it easy to automate our analyses, so nobody will waste time struggling to reproduce results.

Imagine that we instead decided to run our sequence of commands manually and document what we did in a word processor. First, we'd be typing everything twice: Once at the shell prompt and again into the document. We would want to add the exact command with all flags and data arguments in the document to ensure that we can reproduce the results. It is also very inconvenient for people to read a document and type in the commands contained in it. Why not just give them a script that they can easily run?

The process of typing the same commands each time would be painful enough, but to document it in detail while we do it would be distracting. We'd also have to remember to update the document every time we type a command differently. This is hard to do when we're trying to focus on getting results.

Writing a shell script allows us to stay focused on perfecting the process. Once the script is finished and working perfectly, we have already documented the process perfectly. We can and should add comments to the script to make it more readable, but even without comments, the script itself preserves the process in detail.

Many experienced users will *never* run a data processing command from the keyboard. Instead, they *only* put commands into a script. They run, tweak, and re-run the script until it's working perfectly.

An important part of documenting code is making the code *self-documenting*. When writing shell scripts, using long options in commands such as **zip --preserve-case** instead of **zip -C** makes the script much easier to read. While -C is less typing and may be preferable when running zip interactively many times, we only have to type `--preserve-case` once when writing the script, so the laziness of using `-C` doesn't pay here. It just makes us waste time later looking up their meaning, whereas the meaning of the long option may be obvious.

If you use an integrated development environment, such as **APE**, testing the script is a simple matter of pressing **F5**. We do not have to exit the editor (as we would when using **nano**) and we don't lose our place in the script.

### 4.2.3 Why Unix Shell Scripts?

There are many scripting languages to choose from, including those used on Unix systems, like Bourne shell, C shell, Perl, Python, etc., as well as some languages confined to other platforms like Visual Basic (Microsoft Windows only) and AppleScript (Apple only).

Note that the Unix-based scripting languages can be used on *any* platform, *including* Microsoft Windows (with Cygwin, for example) and Apple's Mac OS X, which is Unix-compatible by design. Once you learn to write Unix shell scripts, you're prepared to do scripting on any computer, without having to learn another language. There is little reason *not* to use Unix shell scripts in place of proprietary scripting languages.

### 4.2.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Describe three reasons for writing shell scripts instead of running commands from the keyboard.
2. What feature of Unix makes scripting so easy to implement and use? Explain.
3. When should we run commands at the shell prompt and when should we put them in a script?
4. What are three advantages of a script over a document explaining the commands to run?
5. What type of flag arguments should we use in scripts? Why?
6. What is the advantage of Unix scripting languages over others such as Visual Basic or AppleScript? What is the disadvantage?

## 4.3 Which Shell?

### 4.3.1 Common Shells

When writing a Unix shell script, there are two families of scripting languages to choose from: Bourne shell and C shell. These were the first two mainstream shells for Unix, and all mainstream shells that have come since are compatible with one or the other.

Some of most popular new shells derived from Bourne shell are Bourne again shell (bash), Debian Almquist Shell (dash), KornShell (ksh), and Z shell (zsh). T shell (TENEX C shell, tcsh) is the only mainstream extension of C shell.

- Bourne shell family
  - Bourne shell (sh)
  - Bourne again shell (bash)
  - Debian Almquist shell (dash)
  - Korn shell (ksh)
  - Z-shell (zsh)
- C shell family
  - C shell (csh)
  - T shell (tcsh)

Both Bourne shell and C shell have their own pros and cons. C shell syntax is cleaner, more intuitive, and more similar to the C programming language (hence the name C shell). However, C shell lacks some features such as subprograms (although C shell scripts can run other C shell scripts, which is arguably a better approach in many situations).

Bourne shell is used almost universally for Unix system scripts, while C shell is fairly popular in some areas of scientific research. Every Unix system has a Bourne shell in `/bin/sh`. All other shells may need to be installed via a package manager on some systems. Hence, using POSIX Bourne shell syntax (not bash, ksh, or zsh) for scripts maximizes their portability by ensuring that they will run on any Unix system.

If your script contains only external commands, then it actually won't matter which shell runs it. However, most scripts utilize the shell's internal commands, control structures, and features like redirection and pipes, which differ among shells.

More modern shells such as bash, ksh, and tcsh, are backward-compatible with Bourne shell or C shell. This means that shells such as **bash** and **dash** can run POSIX Bourne shell scripts. In fact, on some systems, **sh** is just a link to **bash** or **dash**. The extended shells add some additional scripting constructs and convenient interactive features. Most of the advantages of shells such as **bash** and **tcsh** are in the interactive features, like auto-completion and command editing. The added constructs for scripting are nice, but do not make scripting significantly easier.

### 4.3.2 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the two families of shell programs?
2. State one advantage of Bourne shell for scripting and one advantage of C shell.
3. What is the advantage of using POSIX Bourne shell for scripting rather than an extended shell such as **bash** or **dash**?
4. What is the disadvantage of using POSIX Bourne shell for scripting rather than an extended shell such as **bash** or **dash**?

## 4.4 Writing and Running Shell Scripts

A shell script is a simple text file and can be written using any Unix text editor. Some discussion of Unix text editors can be found in Section 3.10.4.

**Caution** Recall from Section 3.9.1 that Windows uses a slightly different text file format than Unix. Hence, editing Unix shell scripts in a Windows editor can be problematic. Users are advised to do all of their editing on a Unix machine rather than write programs and scripts on Windows and transfer them to Unix.

Shell scripts often contain very complex commands that are wider than a typical terminal window. A command can be continued on the next line by typing a backslash (\) immediately before pressing **Enter**. A backslash as the very last character on a line (not even white space may follow it) is known as a *continuation character*. This feature is included in all Unix shells and other languages such as Python.

```
printf "%s %s\n" "This command is too long to fit in a single 80-column" \
       "terminal window, so we break it up with a backslash.\n"
```

It's a good idea to name the script with a file name extension that matches the shell it uses. This just makes it easier to see which shell each of your script files use. Table 4.1 shows conventional file name extensions for the most common shells. However, if a script is to be installed into the PATH so that it can be used as a regular command, it is usually given a name with no extension. Most users would rather type "cleanup" than "cleanup.bash".

**Caution** A common mistake is to use the wrong file name extension on a shell script, such as naming the file `script.sh` when it uses **bash** features. On some systems such as Redhat Enterprise Linux, **sh** is actually a link to **bash**, so this will work fine. However, this causes problems on systems where **sh** is not **bash**, such as BSD and Debian Linux. If your script uses **bash** features, it should have a ".bash" file name extension, not ".sh".

| Shell | Extension |
|---|---|
| Bourne Shell | .sh |
| C shell | .csh |
| Bourne Again Shell | .bash |
| T shell | .tcsh |
| Korn Shell | .ksh |
| Z-shell | .zsh |

Table 4.1: Conventional script file name extensions

Like all programs, shell scripts should contain comments to explain what the commands in it are doing. In all Unix shells, anything from a '#' character to the end of a line is considered a comment and ignored by the shell.

```
# Print the name of the host running this script
hostname
```

**Practice Break**
Using your favorite text editor, enter the following text into a file called `hello.sh`.

1. The first step is to create the file containing your script, using any text editor, such as **nano**:

   ```
   shell-prompt: nano hello.sh
   ```

   Once in the text editor, add the following text to the file:

   ```
   printf "Hello!\n"
   printf "I am a script running on a computer called 'hostname'\n"
   ```

   After typing the above text into the script, save the file and exit the editor. If you are using **nano**, the menu at the bottom of the screen tells you how to save (write out, Ctrl+o) and exit (Ctrl+x).

2. Once we've written a script, we need a way to run it. A shell script is simply input to a shell program. Like many Unix programs, shells take their input from the standard input by default. We could, therefore, use redirection to make it read the file via standard input:

   ```
   shell-prompt: sh < hello.sh
   ```

   Shells can also take an input file as a data argument:

   ```
   shell-prompt: sh hello.sh
   ```

However, Unix shells and other scripting languages provide a more convenient method of indicating what program should interpret them. If we add a special comment, called a *shebang line* to the top of the script file and make the file executable using **chmod**, the script can be executed like a Unix command. We can then simply type its name at the shell prompt, and another shell process will start up and run the commands in the script. If the directory containing such a script is included in $PATH, then the script can be run from any CWD just like **ls**, **cp**, etc.

The shebang line consists of the string "#!" followed by the full path name of the command that should be used to execute the script, or the path **/usr/bin/env** followed by the name of the command. For example, both of the following are valid ways to indicate a Bourne shell (sh) script, since `/bin/sh` is the Bourne shell command.

```
#!/bin/sh
```

```
#!/usr/bin/env sh
```

When you run a script simply by typing its file name at the Unix command-line, a new shell process is created to interpret the commands in the script. The shebang line specifies which program is invoked for the new shell process that runs the script.

**Note** The shebang line must begin *at the very first character of the script file.* There cannot even be blank lines above it or white space to the left of it. The "#!" is an example of a *magic number*. Many files begin with a 16-bit (2-character) code to indicate the type of the file. The "#!" indicates that the file contains some sort of interpreted language program, and the characters that follow will indicate where to find the interpreter.

The **/usr/bin/env** method is used for add-on shells and other interpreters, such as Bourne again shell (bash), Korn shell (ksh), and Perl (perl). These interpreters may be installed in different directories on different Unix systems. For example, bash is typically found in **/bin/bash** on Linux systems, **/usr/local/bin/bash** on FreeBSD systems, **/usr/pkg/bin/bash** on NetBSD, and **/usr/bin/bash** on SunOS. The T shell is found in **/bin/tcsh** on FreeBSD and CentOS Linux and in **/usr/bin/tcsh** on Ubuntu Linux.

In addition, users of Redhat Enterprise Linux (RHEL) and derivatives may want to install a newer version of bash under a different prefix, using pkgsrc or another add-on package manager. RHEL is a special kind of Linux distribution built on an older snapshot of Fedora Linux for the sake of long-term binary compatibility and stability. As such, it comes with older versions of bash and other common tools.

The **env** command is found in `/usr/bin/env` on virtually all Unix systems. Hence, this provides a method for writing shell scripts that are portable across Unix systems (i.e. they don't need to be modified to run on different Unix systems).

**Note**
Every script or program should be tested on more than one platform (e.g. BSD, Cygwin, Linux, Mac OS X, etc.) immediately, in order to shake out bugs before they cause problems.
The fact that a program works fine on one operating system and CPU does not mean that it's free of bugs.
By testing it on other operating systems, other hardware types, and with other compilers or interpreters, you will usually expose bugs that will seem obvious in hindsight.
As a result, the software will be more likely to work properly when time is critical, such as when there is an imminent deadline approaching and no time to start over from the beginning after fixing bugs. Encountering software bugs at times like these is very stressful and usually easily avoided by testing the code on multiple platforms in advance.

Bourne shell (sh) is present and installed in `/bin` on all Unix-compatible systems, so it's safe to hard-code #!/bin/sh is the shebang line.

C shell (csh) is not included with all systems, but is virtually always in /bin if present, so it is generally safe to use #!/bin/csh as well.

For all other interpreters it's best to use #!/usr/bin/env.

```
#!/bin/sh           (OK and preferred)
```

```
#!/bin/csh          (Usually OK)
```

```
#!/bin/bash         (Bad idea: Not portable)
```

```
#!/usr/bin/perl     (Bad idea: Not portable)
```

```
#!/usr/bin/python   (Bad idea: Not portable)
```

```
#!/bin/tcsh         (Bad idea: Not portable)
```

```
#!/usr/bin/env bash     (This is portable)
```

```
#!/usr/bin/env tcsh     (This is portable)
```

```
#!/usr/bin/env perl     (This is portable)
```

```
#!/usr/bin/env python   (This is portable)
```

By default, a shell script will continue to run after one of the commands in the script fails. We usually do not want this behavior. We can tell the script to exit on errors by adding a `-e` to the command line:

```
#!/bin/sh -e
#!/bin/csh -e
```

Unfortunately, we cannot pass arguments to the interpreter following **env** in the shebang line.

```
#!/usr/bin/env bash -e      # This will fail
```

With Bourne family shells, we can add command line options after the fact using the internal **set** command. We can also disable any command line option by changing the '-' to a '+'. It may seem counterintuitive to enable something with '-' and disable it with '+', but it is what it is.

```
#!/usr/bin/env bash

# Set exit-on-error as if bash had been run with -e
set -e

# Disable exit-on-error
set +e
```

**Example 4.1** A Simple Bourne Shell Script

Suppose we want to write a script that is always executed by bash, the Bourne Shell. We simply need to add a shebang line indicating the path name of the **bash** executable file.

```
shell-prompt: nano hello.sh
```

Enter the following text in the editor. Then save the file and exit back to the shell prompt.

```
#!/bin/sh -e

# A simple command in a shell script
printf "Hello, world!\n"
```

Now, make the file executable and run it:

```
shell-prompt: chmod a+rx hello.sh    # Make the script executable
shell-prompt: ./hello.sh             # Run the script as a command
```

**Example 4.2** A Simple C-shell Script

Similarly, we might want to write a script that is always executed by csh, C Shell. We simply need to add a shebang line indicating the path name of the **csh** executable file.

```
shell-prompt: nano hello.csh
```

```
#!/bin/csh -e

# A simple command in a shell script
printf "Hello, world!\n"
```

```
shell-prompt: chmod a+rx hello.csh   # Make the script executable
shell-prompt: ./hello.csh            # Run the script as a command
```

**Note**

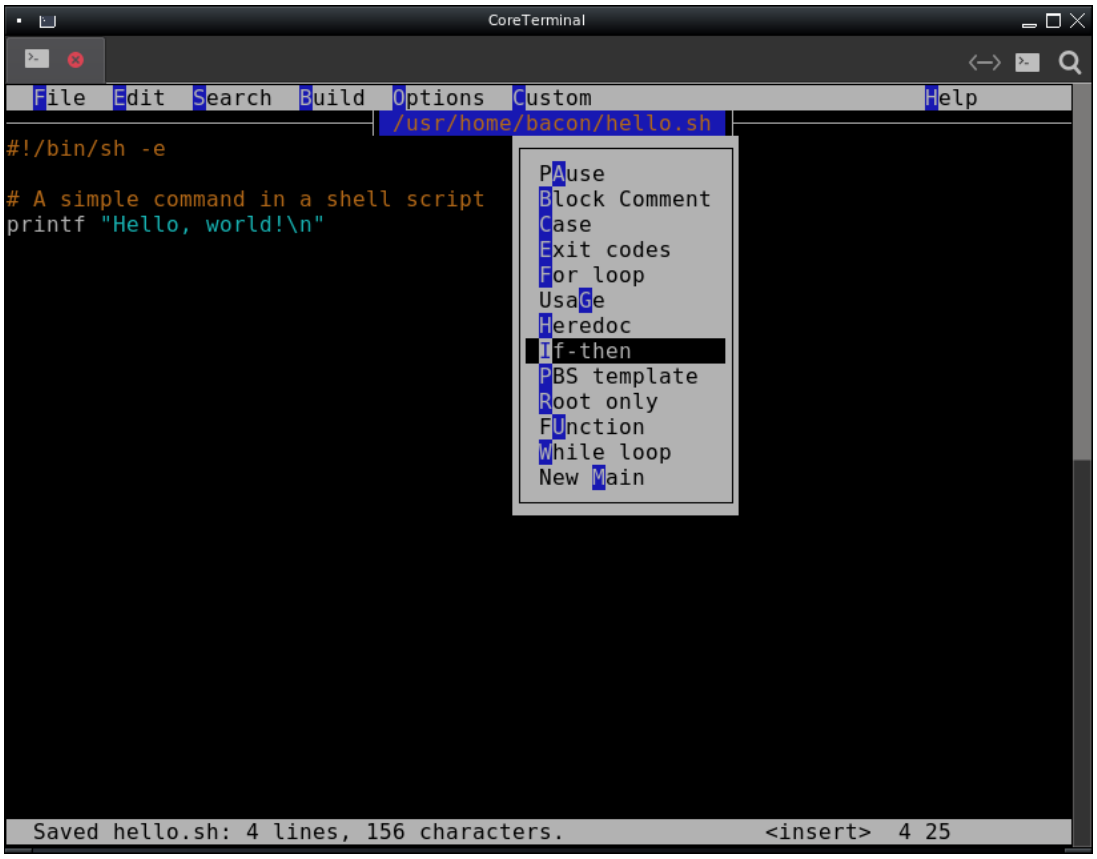
The shebang line in a script is ignored when you explicitly run a shell and provide the script name as an argument or via redirection. The content of the script will be interpreted by the shell that you have manually invoked, regardless of what the shebang line says.

```
# This might not work, since sh will not recognize
# C shell syntax.
shell-prompt: sh hello.csh
```

Scripts that you create and intend to use regularly can be placed in your PATH, so that you can run them from anywhere. A common practice among Unix users is to create a directory called `~/bin`, and configure the login environment so that this directory is always in the PATH. Programs and scripts placed in this directory can then be used like any other Unix command, without typing the full path name.

You can greatly speed up the script development process by using an *Integrated Development Environment*, or *IDE*, instead of a simple text editor. Using an IDE, such as APE, eliminates the need to exit the editor (or use another shell window) to run the script. In APE, we can simply press F5 or type Esc followed by 'r' to run the script. When it finishes, we are still in the editor at the same spot in the script. An IDE is also specialized for writing programs, and hence provides features such as syntax colorization, edit macros, etc.

```
shell-prompt: ape hello.sh
```



```
 File  Edit  Search  Build  Options  Custom                         Help
                            /usr/home/bacon/hello.sh
#!/bin/sh -e

# A simple command in a shell script
printf "Hello, world!\n"

 Saved hello.sh: 4 lines, 156 characters.              <insert>  4 25
```

Figure 4.1: Editing a script in APE

### 4.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Is it wise to write Unix shell scripts in a Windows editor and then upload them to a Unix system? Why or why not?
2. How can we keep very long commands tidy in a script?
3. What rules does Unix enforce regarding file name extensions on shell scripts?
4. What problems might arise if we write a **bash** script and give it a ".sh" file name extension?
5. Why are comments important in shell scripts?
6. Show three ways to run a Bourne shell script called `analysis.sh` and any requirements they entail.
7. What shebang line should be used for Bourne shell scripts? For Bourne again shell scripts? For Python scripts? Explain.

8. How to we make a shell script exit immediately when any command fails?
9. What happens if we run a script as an argument to a shell command that does not match the intended shell, such as **csh analysis.sh**?
10. What are the advantages of using an IDE instead of a simple text editor like nano?

## 4.5 Sourcing Scripts

Normally, running a shell script starts a new shell process, often running a different shell than our interactive shell. For example, our login shell may be **tcsh** or **bash**, and a script may be run by **sh** because it starts with '#!/bin/sh -e".

In some circumstances, we might not want a script to be executed by a separate shell process. For example, suppose we just made some changes to our .cshrc or .bashrc file that would affect `PATH` or some other important environment variable, or the `prompt` shell variable that describes our shell prompt.

If we run the start up script by typing **~/.cshrc** or **~/.bashrc**, a new shell process will be started which will execute the commands in the script and then terminate. Commands in the script may alter the shell variables and environment variables of the child process, but the parent process (the process from which you invoked the script), will be unaffected.

In order to make the "current" shell process run the commands in a script, we must *source* it. This is done using the internal shell command **source** in all shells except Bourne shell, which uses **"."**. Bourne shell derivatives support both "." and "source".

Hence, to source `.cshrc`, we would run

```
shell-prompt: source ~/.cshrc
```

To source `.bashrc`, we would run

```
shell-prompt: source ~/.bashrc
```

or

```
. ~/.bashrc
```

To source `.shrc` from a basic Bourne shell, we would have to run

```
. ~/.shrc
```

### 4.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What happens when we run a script by simply typing its name at the shell prompt?
2. How does sourcing a script differ from running it the usual way?
3. Under what circumstances would we want to source a script rather than run it under a child shell process?

## 4.6 Shell Start-up Scripts

When you start a new shell process (e.g. log in via ssh or open a new terminal window), the shell process may source one or more special scripts called *start up scripts*.

Which scripts are sourced depends on how the shell is started. Non-interactive Bourne shell family processes, such as those used to execute shell scripts, do not source any start up scripts by default.

In contrast, C shell scripts by default source ~/.cshrc if it exists. You can override this in C-shell scripts by invoking the shell with `-f` as follows:

```
#!/bin/csh -ef
```

Shell processes considered *login shells* will source additional scripts that non-login shells do not. For example, a shell process started remotely via ssh is a login shell. A shell process started when opening a new terminal window in a Unix GUI is not a login shell.

The man page for your shell has all the details about which start up scripts are sourced and when. Table 4.2 provides a brief summary.

| **Script** | **Shells that use it** | **Executed by** |
|---|---|---|
| /etc/profile, ~/.profile | Bourne shell family | Login shells only |
| File named by $ENV (typically .shrc or .shinit) | Bourne shell family | All interactive shells (login and non-login) |
| ~/.bashrc | Bourne again shell only | All interactive shells (login and non-login) |
| ~/.bash_profile | Bourne again shell only | Login shells only |
| ~/.kshrc | Korn shell | All interactive shells (login and non-login) |
| /etc/csh.login, ~/.login | C shell family | Login shells only |
| /etc/csh.cshrc, ~/.cshrc | C shell family | All shell processes |
| ~/.tcshrc | T shell | All shell processes |

Table 4.2: Shell Start Up Scripts

Start up scripts are used to configure your PATH and other environment variables, set your shell prompt and other shell features, create aliases for your favorite commands, and anything else you want done when you start a new shell.

One of the most common alterations users make to their start up script is editing their PATH to include a directory containing their own programs and scripts. Typically, this directory is named `~/bin`, but you can name it anything you want. To set up your own `~/bin` to store your own scripts and programs, do the following:

1. shell-prompt: mkdir ~/bin
2. Edit your start up script and add `~/bin` to the PATH.

   If you're using Bourne again shell, you can add `~/bin` to your PATH by adding the following to your `.bashrc`:

   ```
   export PATH=${PATH}:~/bin
   ```

   If you're using T shell, add the following to your `.cshrc` or `.tcshrc`:

   ```
   setenv PATH ${PATH}:~/bin
   ```

   If you're using a different shell, see the documentation for your shell to determine the correct start up script and command syntax.

   **Caution**
   Adding `~/bin` before (left of) ${PATH} will cause your shell to look in `~/bin` before looking in the standard directories such as `/bin` and `/usr/bin`. Hence, if a binary or script in `~/bin` has the same name as another command, the one in `~/bin` will be executed. This is considered a security risk, since users could be tricked into running a Trojan-horse **ls** or other common command if care is not taken to protect `~/bin` from modification. Hence, adding to the tail (right side) of PATH is usually recommended, especially for inexperienced users.

3. Update the PATH in your current shell process by sourcing the start up script, or by logging out and logging back in.

There is no limit to what your start up scripts can do, so you can use your imagination freely and find ways to make your Unix shell environment easier and more powerful.

### 4.6.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

---

1. What are startup scripts?
2. Are startup scripts sourced by shell processes running other scripts?
3. What are startup scripts used for?

## 4.7 String Constants and Terminal Output

Although Unix shells make no distinction between commands entered from the keyboard and those input from a script, there are certain shell features that are meant for scripting and not convenient or useful to use interactively. Many of these features will be familiar to anyone who has done computer programming. They include constructs such as comments, conditionals (e.g. if commands) and loops. The following sections provide a very brief introduction to shell constructs that are used in scripting, but generally not used on the command line.

A string constant in a shell script is anything enclosed in single quotes (’this is a string’) or double quotes ("this is also a string").

Unlike most programming languages, text in a shell scripts that is not enclosed in quotes and does not begin with a ’$’ or other special character is also interpreted as a string constant. Hence, all of the following are the same:

```
shell-prompt: ls /etc
shell-prompt: ls "/etc"
shell-prompt: ls '/etc'
```

Most programming languages would interpret the ’/’ as a division operator and `etc` as a variable name. As you can see, Unix shell languages differ from general purpose languages significantly.

If a string contains white space (spaces or tabs), then it will be seen as multiple separate strings, unless we explicitly state otherwise by enclosing the string in quotes or escaping every white space character. The last example below will not work properly, since ’Program’ and ’Files’ are seen as separate arguments:

```
shell-prompt: cd 'Program Files'     # One directory name
shell-prompt: cd "Program Files"     # One directory name
shell-prompt: cd Program\ Files      # One directory name
shell-prompt: cd Program Files       # Two directory names
```

---

**Note**
Special sequences such as ’\n’, which represents the newline character, must be enclosed in quotes or escaped. Otherwise, the ’\’ is seen as escaping the ’n’, which has no effect. Remember that ’\’ tells the shell to not to interpret the next character as special. This is useful behind a special character such as ’*’ or ’[’, but ’n’ has no special meaning in the first place, so a ’\’ before it will not alter its interpretation.

```
#!/bin/sh -e

printf Hello\n
printf "Hello\n"
printf Hello\\n

HellonHello
Hello
```

---

Output commands are only occasionally useful at the interactive command line. We may sometimes use them to take a quick look at a variable such as $PATH. Output commands are far more useful in scripts, and are used in the same ways as output statements in any programming language.

The **echo** command is commonly used to output text to the terminal:

```
shell-prompt: echo 'Hello!'
Hello!
shell-prompt: echo $PATH
/usr/local/bin:/home/bacon/scripts:/home/bacon/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local ↩
    /sbin
```

However, **echo** should be avoided, since it is not portable across different shells and even the same shell on different Unix systems. There are many different implementations of **echo** commands, some internal to the shell and some external programs. Different implementations of **echo** use different command-line flags and special characters to control output formatting. In addition, the output formatting capabilities of **echo** commands are extremely limited.

The **printf** command supersedes **echo**. It has a rich set of capabilities similar to the `printf()` function in the standard C library and in Java. The **printf** command is specified in the POSIX.2 standard, so its behavior is largely the same on all Unix systems. **Printf** is an external command, so it is independent of which shell you are using. Unlike **echo**, it does not add a newline character by default.

```
shell-prompt: printf 'Hello!\n'
Hello!
```

The general syntax of a printf command is as follows:

```
printf format-string [arguments]
```

The format-string contains literal text and a *format specifier* to match each of the arguments that follows. Each format specifier begins with a '%' and is followed by symbols indicating the format in which to print the argument.

The format string can be the sole argument to **printf**, as long as it does not contain any specifiers. Otherwise, there must be exactly one argument following the format string to match each specifier.

| **Specifier** | **Output** |
|---|---|
| %s | String |
| %20s | String right-justified in a 20-character space |
| %-20s | String left-justified in a 20-character space |

Table 4.3: Printf Format Specifiers

The printf command also recognizes most of the same special character sequences as the C printf() function:

| **Sequence** | **Meaning** |
|---|---|
| \n | Newline (move down to next line) |
| \r | Carriage Return (go to beginning of current line) |
| \t | Tab (go to next tab stop) |

Table 4.4: Special Character Sequences

```
#!/bin/sh -e

printf '%s.\n' 'This is default format'
printf '%50s.\n' 'This is right-justified'
printf '%-50s.\n' 'This is left-justified'
```

```
This is default format.
                           This is right-justified.
This is left-justified                            .
```

There are many other format specifiers and special character sequences. For complete information, run **man printf**.

Any program that prints error messages should send them to the standard error, not the standard output. The **printf** command always outputs to the standard output, but we can work around it using redirection to `/dev/stderr`.

```
        printf 'Error: Input must be a number.\n' >> /dev/stderr
```

---

**Practice Break**
Write a shell script containing the printf command above and run it. Write the same script using two different shells, such as Bourne shell and C shell. What is the difference between the two scripts?

---

### 4.7.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

---

1. How does the Unix shell interpret the character sequence `firstname`? How would it be interpreted by most general purpose programming languages?
2. Show three ways to make the shell see the character sequence `Alfred E.  Neumann` as a single string rather than three strings. Note that there are two spaces after the period.
3. What is the output of the following statement? How do we make it interpret the \n as a newline?

```
#!/bin/sh -e

printf \$15*3=\$45\n
```

4. What are some of the problems with the **echo** command? What is the solution?
5. Show a **printf** statement that prints the number 32,767, right-justified in a field of 10 columns.
6. Show a **printf** statement that prints the error message "Error: Invalid data found in column 2 of input." to the standard error.

## 4.8 Shell and Environment Variables

Variables are essential to any programming language, and scripting languages are no exception. Variables are useful for user input, control structures, and for giving descriptive names to commonly used constants, such as numbers and long path names.

Recall from Section 3.16 that every Unix process has a set of string variables called the *environment*, which are handed down from the parent process in order to communicate important information. For example, the TERM variable, which identifies the type of terminal a user is using, is used by programs such as top, **vi**, **nano**, and **more**, that need to manipulate the terminal screen (move the cursor, highlight characters, etc.) The TERM environment variable is usually set by the shell process so that all of the shell's child processes (those running **vi**, **nano**, etc.) will inherit the variable.

Unix shells also keep another set of variables called *shell variables* that are not part of the environment. These variables are used only for the shell's purposes and are not inherited child processes. The shell variables are structured exactly the same way as the environment variables, each having a name and a value, which is a character string.

There are some special shell variables such as "prompt" and "PS1" (which control the appearance of the shell prompt in C shell and Bourne shell, respectively). Most shell variables, however, are defined by the user for use in scripts, just like variables in any other programming language.

Environment and shell variable names must begin with a letter or an underscore (_), which is optionally followed by more letters, underscores, or digits. The regular expression defining the naming rules for environment variables '[A-Za-z_][A-Za-z0-9_]*'. Environment variable names traditionally use upper case for all letters.

### 4.8.1 Assignment Statements

In all Bourne Shell derivatives, a shell variable is created or modified using the same simple syntax:

```
varname=value
```

**Caution** There can be no space around the '='. If there were, the shell would think that 'varname' is a command, and '=' and 'value' are arguments. A variable assignment is distinct from a command in the Bourne shell family.

```
bash-4.2$ name = Fred
bash: name: command not found
bash-4.2$ name=Fred
bash-4.2$ printf "$name\n"
Fred
```

When assigning a string that contains white space, it must be enclosed in quotes or all white space characters must be escaped:

```
#!/bin/sh -e

name=Joe Sixpack     # Error
name="Joe Sixpack"   # OK
name=Joe\ Sixpack    # OK
```

C shell and T shell use the internal **set** command for assigning variables. Since a variable assignment *is* a command in C shell, we can have white space around the '=' if we wish:

```
#!/bin/csh -ef

set name = "Joe Sixpack"
```

**Caution** Note that Bourne family shells also have a **set** command, but it has a completely different meaning, so take care not to get confused. The Bourne **set** command is used to enable or disable shell command-line options, not variables.

In many languages such as C, Fortran, or Java, we must define variables before we can use (reference) them:

```
    int     c;
    double  x;

    c = 5;
    x = 1.4;
```

Unix shell variables need not be defined before they are assigned a value. Defining variables is unnecessary, since there is only one data type in shell scripts. All shell variables are character strings. There are no integers, Booleans, enumerated types, or floating point variables, although there are some facilities for interpreting shell variables as integers, assuming they contain only digits.

In Bourne shell, we can perform basic integer arithmetic by enclosing an expression in `$(( ))`:

```
c=$(($c + 1))
```

In C shell, we use the @ command, which is a special form of **set** that enables basic arithmetic:

```
@ c = $c + 1
```

Most shells are not capable of handling real numbers. Only integers are supported, mainly for the sake of loop counters and a few other purposes. If you *must* manipulate real numbers in a shell script, you could accomplish it by piping an expression through **bc**, the Unix arbitrary-precision calculator:

```
printf "243.9 * $variable\n" | bc -l
```

Such facilities are very inefficient compared to other languages, however, partly because shell languages are interpreted, not compiled, and partly because they must convert each string to a number, perform arithmetic, and convert the results back to a string. Shell scripts are meant to automate sequences of Unix commands and other programs, not perform numerical computations.

In Bourne shell family shells, environment variables are set by first setting a shell variable of the same name and then *exporting* it to the environment:

```
TERM=xterm
export TERM
```

Modern Bourne shell derivatives such as bash (Bourne Again Shell) can do this in one command:

```
export TERM=xterm
```

**Note** Exporting a shell variable permanently tags it as exported. Any future changes to the variable's value will automatically be copied to the environment. This type of linkage between two objects is very rare in programming languages: Usually, modifying one object has no effect on any other.

C shell derivatives use the setenv command to set environment variables:

```
setenv TERM xterm
```

**Caution** Note that unlike the 'set' command, setenv requires white space, not an '=', between the variable name and the value.

**Note** Since the C shell allows us to create environment variables separate from shell variables, C shell variables traditionally use all lower-case letters, while environment variables use all upper-case. This makes it easier to read scripts that access both shell and environment variables.

C shell variables are not linked to environment variables, except for some special variables, like `path` which is automatically exported to the environment variable `PATH` each time is it updated.

### 4.8.2 Variable References

To reference a shell variable or an environment variable in a shell script, we must precede its name with a '$'. The '$' tells the shell that the following text is to be interpreted as a variable name rather than a string constant. The variable reference is then *expanded*, i.e. replaced by the value of the variable. This occurs anywhere in a command except inside a string bounded by single quotes or following an escape character (\), as explained in Section 4.7. These rules are basically the same for all Unix shells.

```
#!/bin/sh -e

name="Joe Sixpack"
printf "Hello, name!\n"      # Not a variable reference
printf "Hello, $name!\n"     # References variable "name"
printf 'Hello, $name!\n'     # Not a variable reference
printf "Hello, \$name!\n"    # Not a variable reference
```

Output:

```
Hello, name!
Hello, Joe Sixpack!
Hello, $name!
Hello, $name!
```

**Practice Break**
Type in and run the following scripts:

```
#!/bin/sh -e

first_name="Bob"
last_name="Newhart"
printf "%s %s is a superhero.\n" $first_name $last_name
```

CSH version:

```
#!/bin/csh -ef

set first_name = "Bob"
set last_name = "Newhart"
printf "%s %s is a superhero.\n" $first_name $last_name
```

**Note**
If both a shell variable and an environment variable with the same name exist, a normal variable reference will expand the shell variable.
In Bourne shell derivatives, a shell variable and environment variable of the same name always have the same value, since exporting is the only way to set an environment variable. Hence, it doesn't really matter which one we reference.
In C shell derivatives, a shell variable and environment variable of the same name can have different values. If you want to reference the environment variable rather than the shell variable, you can use the printenv command:

```
Darwin heron bacon ~ 319: set name=Sue
Darwin heron bacon ~ 320: setenv name Bob
Darwin heron bacon ~ 321: echo $name
Sue
Darwin heron bacon ~ 322: printenv name
Bob
```

There are some special C shell variables that are automatically linked to environment counterparts. For example, the shell variable `path` is always the same as the environment variable `PATH`. The C shell man page is the ultimate source for a list of these variables.

If a variable reference is immediately followed by a character that could be part of a variable name, we could have a problem:

```
#!/bin/sh -e

name="Joe Sixpack"
printf "Hello to all the $names of the world!\n"
```

Instead of printing "Hello to all the Joe Sixpacks of the world", the printf will fail because there is no variable called "names". In Bourne Shell derivatives, non-existent variables are treated as empty strings, so this script will print "Hello to all the of the world!". C shell will print an error message stating that the variable "names" does not exist.

We can correct this by delimiting the variable name in curly braces:

```
#!/bin/sh -e

name="Joe Sixpack"
printf "Hello to all the ${name}s of the world!\n"
```

This syntax works for all shells. Some shell programmers might insist that all variable references should use {}. My philosophy is that if something is not necessary or at least helpful, then typing it is a waste of time and added clutter.

### 4.8.3 Using Variables for Code Quality

Another very good use for shell variables is in eliminating redundant string constants from a script. Suppose we have a path name referenced multiple times in a script:

```
#!/bin/sh -e

output_value=`myprog`

printf "$output_value\n" >> Run2/Output/results.txt
more Run2/Output/results.txt
cp Run2/Output/results.txt latest-results.txt
```

If for any reason the relative path `Run2/Output/results.txt` should change, then you'll have to search through the script and make sure that all instances are updated. This is a tedious and error-prone process, which can be avoided by using a variable:

```
#!/bin/sh -e

output_file="Run2/Output/results.txt"

output_value=`myprog`
printf "$output_value\n" >> $output_file
more $output_file
cp $output_file latest-results.txt
```

In the second version of the script, if the path name of `results.txt` changes, then only one change must be made to the script. Avoiding redundancy is one of the primary goals of any good programmer.

In a more general programming language such as C or Fortran, this role would be served by a constant, not a variable. However, shells do not support constants, so we use a variable for this.

In most shells, a variable can be marked read-only in an assignment to prevent accidental subsequent changes. Bourne family shells use the **readonly** command for this, while C shell family shells use **set -r**.

```
#!/bin/sh -e

readonly output_file="Run2/Output/results.txt"

output_value=`myprog`
printf "$output_value\n" >> $output_file
more $output_file
cp $output_file latest-results.txt
```

```
#!/bin/csh -ef

set -r output_file = "Run2/Output/results.txt"
set output_value=`myprog`
printf "$output_value\n" >> $output_file
more $output_file
cp $output_file latest-results.txt
```

### 4.8.4 Output Capture

Output from a command can be captured and used as a string in the shell environment by enclosing the command in back-quotes (``). In Bourne-compatible shells, we can also use $() in place of back-quotes.

```
            #!/bin/sh -e

            # Using output capture in a command
            printf "Today is %s.\n" `date`
            printf "Today is %s.\n" $(date)

            # Using a variable.  If using the output more than once, this will
            # avoid running the command multiple times.
            today=`date`
            printf "Today is %s\n" $today
```

### 4.8.5 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the convention for naming C shell variables to help distinguish them from environment variables?
2. Can we use any name we want for a shell variable in our script?
3. What rules do shell and environment variable names need to follow?
4. Show how to assign the value "Alfred E. Neumann" to the shell variable full_name in Bourne shell and in C shell.
5. How do we declare a shell variable? Explain.
6. What is the relationship between a given shell variable and an environment variable with the same name? Explain.
7. Are all C shell variable independent of environment variables? Use an example to clarify.
8. Show the output of the following script: (Try to figure it out first, and then check by typing in and running the script).

```
#!/bin/sh -e

name="Wile E. Coyote"

printf "$name\n"
printf "name\n"
printf '$name\n'
```

9. What is the output of the following script? What do you think is the intended output and how can we make it happen?

```
#!/bin/sh -e

file_size=200

printf "File size is $file_sizeMB.\n"
```

10. What is the danger in the following script? Alter the script to eliminate the risk.

    ```
    #!/bin/sh -e

    printf "The first 20 lines of file.txt are:\n"
    head -n 20 file.txt
    printf "The last 20 lines of file.txt are:\n"
    tail -n 20 file.txt
    ```

11. Write a shell script that prints the following, using a single **printf** command and using **wc -l** to find the number of lines in the file. Exact white space is not important.

    ```
    input1.txt contains 3258 lines.
    ```

## 4.9 Hard and Soft Quotes

Double quotes are known as *soft quotes*, since shell variable references, history events (!), and command output capture ($() or ``) are all expanded when used inside double quotes.

```
shell-prompt: history
  1003  18:11   ps
  1004  18:11   history

shell-prompt: echo "!hi"
echo "history"
history

shell-prompt: echo "Today is `date`"
Today is Tue Jun 12 18:12:33 CDT 2018

shell-prompt: echo "$TERM"
xterm
```

Single quotes are known as *hard quotes*, since every character inside single quotes is taken literally as part of the string, except for history events. Nothing else inside hard quotes is processed by the shell. If you need a literal ! in a string, it must be escaped.

```
shell-prompt: history
  1003  18:11   ps
  1004  18:11   history
shell-prompt: echo '!hi'
echo 'history'
history
shell-prompt: echo '\!hi'
!hi
shell-prompt: echo 'Today is `date`'
Today is `date`
shell-prompt: echo '$TERM'
$TERM
```

What will each of the following print? ( If you're not sure, try it! )

```
#!/bin/sh -e

name='Joe Sixpack'
printf "Hi, my name is $name.\n"
```

```
#!/bin/sh -e

name='Joe Sixpack'
printf 'Hi, my name is $name.\n'
```

```
#!/bin/sh -e

first_name="Joe"
last_name='Sixpack'
name='$first_name $last_name'
printf "Hi, my name is $name.\n"
```

If you need to include a quote character as part of a string, you have two choices:

1. Escape it (precede it with a backslash character):

   ```
   printf 'Hi, I\'m Joe Sixpack.\n'
   ```

2. Use the other kind of quotes to enclose the string. A string terminated by double quotes can contain a single quote and vice-versa:

   ```
   printf "Hi, I'm Joe Sixpack.\n"
   printf 'I'm a Unix scripting "newbie".\n'
   ```

No operator is needed to concatenate strings in a shell command. We can simply place multiple strings in any form (variable references, literal text, etc.) next to each other.

```
var=Joe
printf 'Hello, ''Joe.'
printf "Hello, "'Joe.'
printf 'Hello ,'$var'.'      # Variable reference between hard-quoted strings
printf "Hello, $var."        # Variable between text in a soft-quoted string
```

### 4.9.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What shell constructs are expanded when found inside hard quotes?
2. What shell constructs are expanded when found inside soft quotes?
3. What is the output of the following script if it is run in /home/joe?

   ```
   #!/bin/sh -e

   cwd=$(pwd)
   printf "The CWD is $cwd.\n"
   string='The CWD is $cwd.\n'
   printf "$string"
   ```

4. How do we print a single quote character in a shell command?
5. How do we concatenate two strings in a shell command?

## 4.10 User Input

Shell scripts are often interactive, requesting input in the form of data or parameters directly from the user via the keyboard. All shells have the ability to read input and assign it to shell variables.

In Bourne Shell derivatives, data can be input from the standard input using the **read** command:

```
#!/bin/sh -e

printf "Please enter the name of an animal: "
read animal
printf "Please enter an adjective: "
read adjective1
printf "Please enter an adjective: "
read adjective2
printf "The $adjective1 $adjective2 $animal jumped over the lazy dog.\n"
```

C shell and T shell use the special symbol $< rather than a command to read input:

```
#!/bin/csh -ef

printf "Please enter the name of an animal: "
set animal = "$<"
printf "Please enter an adjective: "
set adjective1 = "$<"
printf "Please enter an adjective: "
set adjective2 = "$<"
printf "The $adjective1 $adjective2 $animal jumped over the lazy dog.\n"
```

The $< symbol behaves like a variable, which makes it more flexible than the **read** command used by Bourne family shells. It can be used anywhere a regular variable can appear.

```
#!/bin/csh -ef

printf "Enter your name: "
printf "Hi, $<!\n"
```

**Caution** The $< symbol should usually be enclosed in soft quotes in case the user enters text containing white space. Otherwise, only the first "word" of input (text before the first white space character) will be captured by a **set** command like those above.

**Practice Break**
Write a shell script that asks the user to enter their first name and last name, stores each in a separate shell variable, and outputs "Hello, first-name last-name". For example:

```
Please enter your first name: Barney
Please enter your last name: Miller
Hello, Barney Miller!
```

### 4.10.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a shell script that lists the files in the CWD, asks the user for the name of a file, a source string, and a replacement string, and then shows the file content with all occurrences of the source replaced by the replacement.

```
shell-prompt: cat fox.txt
The quick brown fox jumped over the lazy dog.

shell-prompt: ./replace.sh
Documents           R                   fox.txt             stringy
Downloads           igv                 stringy.c
File name? fox.txt
Source string? fox
Replacement string? tortoise
The quick brown tortoise jumped over the lazy dog.
```

## 4.11 Conditional Execution

As a Unix scripter, you have been living in a cocoon until now, growing and developing, but confined and unable to do much. In the next few sections, you will become ready to emerge and spread your wings, so you can see the vast possibilities of automated research computing for the first time. To use another metaphor, this is where you will reach the critical mass of knowledge needed to step aside and let Unix do much of the work for you. Give this material the attention it deserves, so that your future as a computational scientist will be as easy and rewarding as it can be.

Sometimes we need to run a particular command or sequence of commands only if a certain condition is true. For example, if program B processes the output of program A, we probably won't want to run B at all unless A finished successfully.

### 4.11.1 Command Exit Status

Conditional execution in Unix shell scripts often utilizes the *exit status* of the most recent command. All Unix programs return an exit status. By convention, programs return an exit status of 0 if they determine that they completed their task successfully and a variety of non-zero error codes if they failed. There are some standard error codes defined in the C header file sysexits.h. You can learn about them by running **man sysexits**. Or, for a quick listing of their names and values, run **grep '#define.*EX_' /usr/include/sysexits.h**.

A shell script can assign an exit status by providing an argument to the **exit** command:

```
#!/bin/sh -e

exit 0      # Report success (EX_OK)

exit 65     # Report input error (EX_DATAERR)
```

We can check the exit status of the most recent command by examining the shell variable `$?` in Bourne shell family shells or `$status` in C shell family shells.

```
bash> ls
myprog.c
bash> echo $?
0
bash> ls -z
ls: illegal option -- z
usage: ls [-ABCFGHILPRSTUWZabcdfghiklmnopqrstuwx1] [-D format] [file ...]
bash> echo $?
1
bash>
```

```
tcsh> ls
myprog.c
tcsh> echo $status
```

```
0
tcsh> ls -z
ls: illegal option -- z
usage: ls [-ABCFGHILPRSTUWZabcdfghiklmnopqrstuwx1] [-D format] [file ...]
tcsh> echo $status
1
tcsh>
```

**Practice Break**
Run several commands correctly and incorrectly and check the $? or $status variable after each one.

### 4.11.2 If-then-else Commands

All Unix shells have an `if-then-else` construct implemented as internal commands. The Bourne shell family of shells all use the same basic syntax. The C shell family of shells also use a common syntax, which is somewhat different from the Bourne shell family, more closely resembling the C language.

#### Bourne Shell Family

Unlike general-purpose languages such as C and Java, a Bourne shell conditional command does not take a Boolean (true/false) expression. Rather, it takes a Unix command, and the decision is based on the exit status of that command. The general syntax of a Bourne shell family conditional is shown below. Note that there can be an unlimited number of `elifs`, but we will use only one for this example.

```
#!/bin/sh -e

if command1; then   # Command1 succeeded (exit status was 0)
    command
    command
    ...
elif command2; then # Command2 succeeded (exit status was 0)
    command
    command
    ...
else                # All commands above failed
    command
    command
    ...
fi
```

**Note**
The 'if' and the 'then' are actually two separate commands, so they must either be on separate lines, or separated by a ';', which can be used instead of a newline to separate Unix commands.

**Note**
Code controlled by an `if` should be consistently indented as shown above. How much indentation is used is a matter of personal taste, but four spaces is typical.

In the example above, the **if** command executes **command1** and checks the exit status when it completes. If the exit status is 0 (indicating success), then all the commands before the **elif** are executed, and everything after the **elif** is skipped.

If the exit status is non-zero, then nothing above the **elif** is executed. The **elif** command then executes **command2** and checks its exit status.

If the exit status of **command2** is 0, then the commands between the **elif** and the **else** are executed and everything after the **else** is skipped.

If the exit status of **command2** is non-zero, everything above the **else** is skipped and everything between the **else** and the **fi** is executed.

**Note** In Bourne shell if commands, an exit status of zero effectively means 'true' and non-zero means 'false', which is the opposite of C and similar languages.

In most programming languages, we use some sort of Boolean expression (usually a comparison, also known as a relation), not a command, as the condition for an if statement. This is generally true in Bourne shell scripts as well, but the capability is provided in an interesting way. We'll illustrate by showing an example and then explaining how it works.

Suppose we have a shell variable and we want to check whether it contains the string "blue". We could use the following if command to test:

```
#!/bin/sh -e

printf "Enter the name of a color: "
read color

if [ "$color" = "blue" ]; then
    printf "You entered blue.\n"
elif [ "$color" = "red" ]; then
    printf "You entered red.\n"
else
    printf "You did not enter blue or red.\n"
fi
```

This may look like it violates what we just stated; that Bourne shell conditionals take a command, not a Boolean expression. The interesting thing about this code is that the square brackets are *not* Bourne shell syntax.

The '[' in the conditional above is actually an external command. In fact, it is simply another name for the **test** command. The files `/bin/test` and `/bin/[` are actually links the same executable file:

```
shell-prompt: ls -l /bin/test /bin/[
-r-xr-xr-x  2 root  wheel  8516 Apr  9  2012 /bin/[*
-r-xr-xr-x  2 root  wheel  8516 Apr  9  2012 /bin/test*
```

We could have also written the following, to make it more obvious that we are actually running another command in the **if** command:

```
if test "$color" = "blue"; then
```

Hence, '$color', '=', 'blue', and ']' are arguments to the '[' command, and must be separated by white space. If the command is invoked as '[', it requires the last argument to be ']'. If invoked as 'test', the ']' is not allowed.

The test command can be used to perform comparisons (relational operations) on variables and constants, as well as a wide variety of tests on files. For comparisons, test takes three arguments: the first and third are string values and the second is a relational operator.

```
# Compare a variable to a string constant
test "$name" = 'Bob'
[ "$name" = 'Bob' ]
```

```
# Compare the output of a program directly to a string constant
test `myprog` = 42
[ `myprog` = 42 ]
```

For file tests, test takes two arguments: The first is a flag indicating which test to perform and the second is the path name of the file or directory.

```
# See if output file exists and is readable to the user
# running test
test -r output.txt
[ -r output.txt ]
```

The exit status of test is 0 (success) if the test is deemed to be true and a non-zero value if it is false.

```
shell-prompt: test 1 = 1
shell-prompt: echo $?   # Or $status for C shell
0
shell-prompt: test 1 = 2
shell-prompt: echo $?   # Or $status for C shell
1
```

The relational operators supported by **test** are shown in Table 4.5.

| **Operator** | **Relation** |
|---|---|
| = | Lexical equality (string comparison) |
| -eq | Integer equality |
| != | Lexical inequality (string comparison) |
| -ne | Integer inequality |
| < | Lexical less-than (10 < 9) |
| -lt | Integer less-than (9 -lt 10) |
| -le | Integer less-than or equal |
| > | Lexical greater-than |
| -gt | Integer greater-than |
| -ge | Integer greater-than or equal |

Table 4.5: Test Command Relational Operators

**Caution**
Note that some operators, such as < and >, have special meaning to the shell, so they must be escaped or quoted.

```
[ 10 > 9 ]
test 10 > 9     # Redirects output to a file called '9'.
                # The only argument sent to the test command is '10'.
                # The test command issues an error message since it
                # did not receive enough arguments.

[ 10 \> 9 ]
test 10 \> 9    # Compares 10 to 9

[ 10 '>' 9 ]
test 10 '>' 9   # Compares 10 to 9
```

**Caution** It is a common error to use '==' with the test command, but the correct equality operator is '=', unlike C and similar languages.

Common file tests are shown in Table 4.6. To learn about additional file tests, run **man test**.

| **Flag** | **Test** |
|---|---|
| -e | Exists |
| -r | Is readable |
| -w | Is writable |
| -x | Is executable |
| -d | Is a directory |
| -f | Is a regular file |
| -L | Is a symbolic link |
| -s | Exists and is not empty |
| -z | Exists and is empty |

Table 4.6: Test command file operations

**Caution**

Variable references in a [ or test command should usually be enclosed in soft quotes. If the value of the variable contains white space, such as "navy blue", and the variable is not enclosed in quotes, then "navy" and "blue" will be considered two separate arguments to the **[** command, and it will fail.

Furthermore, if there is a chance that a variable used in a comparison is empty, then we must attach a common string to the arguments on both sides of the operator. It can be almost any character, but '0' is popular and easy to read.

```
name=""
if [ "$name" = "Bob" ]; then    # Error, expands to: if [ = Bob; then
if [ 0"$name" = 0"Bob" ]; then  # OK, expands to: if [ 0 = 0Bob ]; then
```

Relational operators are provided by the test command, not by the shell. Hence, to find out the details, we would run "man test" or "man [", not "man sh" or "man bash".

**Practice Break**

Run the following commands in sequence and run **echo $?** after every test under Bourne shell or **echo $status** after every test under C shell.

```
which [ test
test 1 = 1
test 1=2
test 1 = 2
[ 1 = 1
[ 1 = 1 ]
[ 2 < 10 ]
[ 2 \< 10 ]
[ 2 -lt 10 ]
name=''                 # Bourne shell only
set name=''             # C shell only
[ $name = Bill ]
[ 0$name = 0Bill ]
name=Bob                # Bourne shell only
set name=Bob            # C shell only
[ $name = Bill ]
[ $name = Bob ]
```

### C shell Family

Unlike the Bourne shell family of shells, the C shell family implements its own conditional expressions and operators, so there is generally no need for the **test** or **[** command, though you can use it in C shell scripts if you really want to.

The C shell **if** command requires () around the condition, and the condition is a Boolean expression, just like in C and similar languages. As in C, and unlike Bourne shell, a value of zero is considered false and non-zero is true.

```
#!/bin/csh -ef

printf "Enter the name of a color: "
set color = "$<"

if ( { test "$color" \< blue } ) then
    printf "Yup.\n"
endif

if ( "$color" == "blue" ) then
    printf "You entered blue.\n"
else if ( "$color" == "red" ) then
    printf "You entered red.\n"
else
    printf "You did not enter blue or red.\n"
endif
```

The C shell relational operators are shown in Table 4.7.

| **Operator** | **Relation** |
|---|---|
| < | Integer less-than |
| > | Integer greater-than |
| <= | Integer less-then or equal |
| >= | Integer greater-than or equal |
| == | String equality |
| != | String inequality |
| =~ | String matches glob pattern |
| !~ | String does not match glob pattern |

Table 4.7: C Shell Relational Operators

C shell if commands also need soft quotes around strings that contain white space. However, unlike the test command, it can handle empty strings, so we don't need to add an arbitrary prefix like '0' if the string may be empty.

```
if [ 0"$name" = 0"Bob" ]; then
```

```
if ( "$name" == "Bob" ) then
```

The most readable way to check the status of a command in C shell is using the status variable. Note that we need to avoid invoking **csh** with `-e` so that the shell process will not terminate when a command fails.

```
#!/bin/csh -f

command1
if ( $status == 0 ) then
    # Stuff to do only if command1 succeeded
else
    exit 1  # Exit on error since we did not use #!/bin/csh -e
endif
```

### 4.11.3 Shell Conditional Operators

Unix shells provide conditional operators that allow us to invert the exit status of a command or combine exit status from multiple commands. They use the same Boolean operators as C for AND (&&), OR (||), and NOT (!).

| Operator | Meaning | Exit status |
|---|---|---|
| test ! command | NOT | 0 if command failed, 1 if it succeeded |
| command1 && command2 | AND | 0 if both commands succeeded |
| command1 \|\| command2 | OR | 0 if either command succeeded |

Table 4.8: Shell Conditional Operators

```
# Invert exit status (0 to non-zero, non-zero to 0)
shell-prompt: ! command

# See if both command1 and command2 succeeded
shell-prompt: command1 && command2

# See if either command1 or command2 succeeded
shell-prompt: command1 || command2
```

These operators can be used in Bourne shell conditionals much the same way as in C:

```
if [ 0"$first_name" = 0"Bob" ] && [ 0"$last_name" = 0"Newhart" ]; then
```

We can also the **test** command's own operators:

```
if [ 0"$first_name" = 0"Bob" -a 0"$last_name" = 0"Newhart" ]; then
```

Note that in the case of the && operator, command2 will not be executed if command 1 fails (exits with non-zero status). There is no point running the second command, since both commands must succeed to produce an overall status of 0. Once any command in an && sequence fails, the exit status of the whole sequence will be 1 no matter what happens after that.

Likewise in the case of a || operator, once any command succeeds (exits with zero status), the remaining commands will not be executed.

This fact is often used as a clever trick to conditionally execute a command only if another command succeeds or fails.

```
# Execute main-processing only if pre-processing succeeds and
# post-processing only if main-processing succeeds
pre-processing && main-processing && post-processing

# Equivalent using an if-then-fi
if pre-processing; then
    if main-processing; then
        post-processing
    fi
fi
```

Conditional operators can also be used in a C shell if command. Parenthesis are recommended around each relation for readability.

```
if ( ("$first_name" == "Bob") && ("$last_name" == "Newhart") ) then
```

**Practice Break**
Run the following commands in sequence and run **echo $?** after every command under Bourne shell or **echo $status** after every command under C shell.

```
ls -z
ls -z && echo Done
ls -a && echo Done
ls -z || echo Done
ls -a || echo Done
```

**Practice Break**
Instructor: Lead the class through development of a script that does the following. The solution is at the end of the chapter. No peeking...

1. Lists the CWD
2. Prompts the user for a filename and reads it
3. Prints an error message and exits with status 65 if the file does not exist or is not a regular file
4. Displays the first 5 lines of the file
5. Prompts the user for a simple search string and reads it
6. Displays lines in the file that contain the search string

```
shell-prompt: ./search.sh
CNC-EMDiff              Reference               search-me.txt
Computer                SVN                     search.sh
Enter the name of the file to search: searchme.txt
searchme.txt is not a regular file.

shell-prompt: ./search.sh
CNC-EMDiff              Reference               search-me.txt
Computer                SVN                     search.sh
Enter the name of the file to search: GFF
GFF is not a regular file.

shell-prompt: ./search.sh
CNC-EMDiff              Reference               search-me.txt
Computer                SVN                     search.sh
Enter the name of the file to search: search-me.txt
This
is
a
test
file
Enter a string to search for in search-me.txt: test
test
```

### 4.11.4 Case and Switch Commands

If you need to compare a single variable to many different values, you could use a long string of elifs or else ifs:

```
#!/bin/sh -e

printf "Enter a color name: "
```

```
read color

if [ "$color" = "red" ] || \
   [ "$color" = "orange" ]; then
    printf "Long wavelength\n"
elif [ "$color" = "yellow" ] || \
     [ "$color" = "green" ] || \
     [ "$color" = "blue" ]; then
    printf "Medium wavelength\n"
elif [ "$color" = "indigo" ] || \
     [ "$color" = "violet" ]; then
    printf "Short wavelength\n"
else
    printf "Invalid color name: $color\n"
fi
```

Like most languages, however, Unix shells offer a cleaner solution.

Bourne shell has the **case** command:

```
#!/bin/sh -e

printf "Enter a color name: "
read color

case $color in
    red|orange)
        printf "Long wavelength\n"
        ;;
    yellow|green|blue)
        printf "Medium wavelength\n"
        ;;
    indigo|violet)
        printf "Short wavelength\n"
        ;;
    *)
        printf "Invalid color name: $color\n"
        ;;
esac
```

C shell has a **switch** command that looks almost exactly like the switch statement in C, C++, and Java:

```
#!/bin/csh -ef

printf "Enter a color name: "
set color = "$<"

switch($color)
case    red:
case    orange:
    printf "Long wavelength\n"
    breaksw
case    yellow:
case    green:
case    blue:
    printf "Medium wavelength\n"
    breaksw
case    indigo:
case    violet:
    printf "Short wavelength\n"
    breaksw
```

```
default:
    printf "Invalid color name: $color\n"
endsw
```

**Note** The **;;** and **breaksw** commands cause a jump to the first command after the entire **case** or **switch**. The **;;** is required after every value in the Bourne shell **case** command. The **breaksw** is optional in the **switch** command. If omitted, the script will simply "fall through" to the next case (continue on and execute the commands for the next case value).

**Note**
Code controlled by a `case` or `switch` should be consistently indented as shown above. How much indentation is used is a matter of personal taste, but four spaces is typical.

### 4.11.5 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the exit status of a command that succeeds? One that fails?
2. What variables contain the exit status of the most recent command in Bourne shell and C shell?
3. What is the meaning of **[** in a shell script and how is it used?
4. Write a shell script that lists the files in the CWD, asks the user for the name of a file, tests for the existence of the file, and issues an error message if it does not exist. If the file does exist, the script then asks for a source string and a replacement string, verifying that the source string is not empty, and then shows the file content with all occurrences of the source replaced by the replacement. The script should exit with status 65 (EX_DATAERR) if any bad input is received.

```
shell-prompt: cat fox.txt
The quick brown fox jumped over the lazy dog.

shell-prompt: ./replace.sh
Documents           R                     fox.txt             stringy
Downloads           igv                   stringy.c
File name? fxo.txt
File fxo.txt does not exist.
shell-prompt: echo $?
65

shell-prompt: ./replace.sh
Documents           R                     fox.txt             stringy
Downloads           igv                   stringy.c
File name? fox.txt
Source string?
Source string cannot be empty.
shell-prompt: echo $?
65

shell-prompt: ./replace.sh
Documents           R                     fox.txt             stringy
Downloads           igv                   stringy.c
File name? fox.txt
Source string? fox
Replacement string? tortoise
The quick brown tortoise jumped over the lazy dog.
shell-prompt: echo $?
0
```

5. Modify the previous script so that it reports an error if the replacement string is empty or the same as the source. Use a conditional operator to check both conditions in one **if-then-else** command.

```
shell-prompt: ./replace.sh
Documents           R                   fox.txt             stringy
Downloads           igv                 stringy.c
File name? fox.txt
Source string? fox
Replacement string?
Replacement must not be empty or the same as source.
shell-prompt: echo $?
65

shell-prompt: ./replace.sh
Documents           R                   fox.txt             stringy
Downloads           igv                 stringy.c
File name? fox.txt
Source string? fox
Replacement string? fox
Replacement must not be empty or the same as source.
shell-prompt: echo $?
65
```

6. Write a shell script that asks the user for a directory name and the name of an archive to create from it, checks the file name extension on the archive name using a **switch/case**, and creates a tarball with the appropriate compression. The script should report an error and exit with status 65 if an invalid file name extension is used for the archive name, or if the directory name entered does not exist or is not a directory.

```
shell-prompt: ./case.sh
Coral               Qemu                case.sh             scripts
Directory to archive? Qem
Qem is not an existing directory.

shell-prompt: ./case.sh
Coral               Qemu                case.sh             scripts
Directory to archive? case.sh
case.sh is not an existing directory.

shell-prompt: ./case.sh
Coral               Qemu                case.sh             scripts
Directory to archive? Qemu
Archive name? qemu.ta
Invalid archive name: qemu.ta

shell-prompt: ./case.sh
Coral               Qemu                case.sh             scripts
Directory to archive? Qemu
Archive name? qemu.txz
a Qemu
a Qemu/FreeBSD-13.0-RELEASE-riscv-riscv64.raw
```

| Extension | Tool |
|---|---|
| tar | No compression |
| tar.gz or tgz | gzip |
| tar.bz2 or tbz | bzip2 |
| tar.xz or txz | xz |

Table 4.9: Compression tool for each filename extension

In Bourne shell, the file name extension can be extracted from a shell variable as follows:

```
extension=${filename##*.}    # Strip off everything to the last '.'
```

In C shell:

```
set extension=${archive:e}  # Extract filename extension
```

7. Write a shell script that does the following in sequence:

   (a) Check for the existence of Homo_sapiens.GRCh38.107.chromosome.1.gff3 in the CWD. If it is not present, download the gzipped file using **curl** from http://ftp.ensembl.org/pub/release-107/gff3/homo_sapiens/ and decompress it.

   (b) Display the first 5 lines of the file that do not begin with '#', so the user can see the format of an entry. Hint: See the **grep** man page for an option to select lines that *do not* match the given pattern.

   (c) Ask the user which column to search, and then display all unique values in that column. Hint: Use **grep** again to filter out lines beginning with '#', run them through **cut** or **awk** to select just the desired column, and run the output through **sort** and **uniq** or just **sort** with the appropriate flags.

   (d) Ask the user for a search key, and display all the lines in the file not beginning with '#' that *contain* the given key in the given column. Hint: The awk '~' operator means "contains", e.g. '$1 ~ "text"' means the first field contains the string "text". You can use the `-v` to set `column` and `key` variables to use in the awk pattern.

   ```
   awk -v column=$column -v key=$key 'your awk script'
   ```

```
shell-prompt: ./gff.sh
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 4111k  100 4111k    0     0   550k      0  0:00:07  0:00:07 --:--:--  589k

1       GRCh38  chromosome      1       248956422       .       .       .      ID= ↩
    chromosome:1;Alias=CM000663.2,chr1,NC_000001.11
1       .       biological_region       10469   11240   1.3e+03 .       .       ↩
    external_name=oe %3D 0.79;logic_name=cpg
1       .       biological_region       10650   10657   0.999   +       .       ↩
    logic_name=eponine
1       .       biological_region       10655   10657   0.999   -       .       ↩
    logic_name=eponine
1       .       biological_region       10678   10687   0.999   +       .       ↩
    logic_name=eponine

Column to search? 3
CDS
biological_region
chromosome
exon
five_prime_UTR
gene
lnc_RNA
mRNA
miRNA
ncRNA
ncRNA_gene
pseudogene
pseudogenic_transcript
rRNA
scRNA
snRNA
snoRNA
three_prime_UTR
unconfirmed_transcript
```

```
Search key? UTR
1       havana  five_prime_UTR  65419   65433   .       +       .       Parent= ↩
    transcript:ENST00000641515
1       havana  five_prime_UTR  65520   65564   .       +       .       Parent= ↩
    transcript:ENST00000641515
1       havana  three_prime_UTR 70009   71585   .       +       .       Parent= ↩
    transcript:ENST00000641515
1       havana  five_prime_UTR  923923  924431  .       +       .       Parent= ↩
    transcript:ENST00000616016
1       havana  three_prime_UTR 944154  944574  .       +       .       Parent= ↩
    transcript:ENST00000616016
...
```

## 4.12 Loops

As in other programming languages, our scripts often need to run the same command or commands repeatedly. Very often, we need to run the same sequence of commands on a group of files. In simple cases, we can simply provide all of the files as arguments to a single invocation of a command, or use **xargs** to provide them:

```
shell-prompt: analyze input*.txt
find . -name 'input*.txt' | xargs analyze
```

We can achieve the same effect, as well as handle more complex situations involving multiple commands, using a loop in a shell script.

### 4.12.1 For and Foreach

Unix shells offer a type of loop that takes an enumerated list of string values, rather than counting through a sequence of numbers or looping while some input condition is true. This makes shell scripts very convenient for working with sets of files or arbitrary sets of values. This type of loop is well suited for use with globbing (file name patterns using wild cards, as discussed in Section 3.7.5):

```
#!/bin/sh -e

# Process input-1.txt, input-2.txt, etc.
for file in input-*.txt
do
    ./myprog $file
done
```

```
#!/bin/csh -ef

# Process input-1.txt, input-2.txt, etc.
foreach file (input-*.txt)
    ./myprog $file
end
```

**Note**
Code controlled by a loop should be consistently indented as shown above. How much indentation is used is a matter of personal taste, but four spaces is typical.

These loops are not limited to using file names, however. We can use them to iterate through any list of string values:

```
#!/bin/sh -e

for fish in flounder gobie hammerhead manta moray sculpin
do
    printf "%s\n" $fish
done
```

```
#!/bin/sh -e

for c in 1 2 3 4 5 6 7 8 9 10
do
    printf "%d\n" $c
done
```

To iterate through a list of integers too long to type out, we can utilize the **seq** command, which takes a starting value, optionally an increment value, and an ending value, and prints the sequence to the standard output. We can use shell output capture (Section 4.8.4) to represent the output of the **seq** command as a string in the script:

```
#!/bin/sh -e

# Count from 0 to 1000 in increments of 5
for c in $(seq 0 5 1000); do
    printf "%d\n" $c
done
```

```
#!/bin/csh -ef

foreach c (`seq 0 5 1000`)
    printf "%s\n" $c
end
```

The **seq** can even be used to embed integer values in a non-integer list:

```
#!/bin/sh -e

# Process all human chromosomes
for chromosome in $(seq 1 22) X Y; do
    printf "chr%s\n" $chromosome
done
```

**Practice Break**
Type in and run the fish example above.

**Example 4.3** Multiple File Downloads

Often we need to download many large files from another site. This process would be tedious to do manually: Start a download, wait for it to finish, start another... There may be special tools provided by the website, but often they are poorly maintained or difficult to use. In many cases, we may be able to automate the download using a simple script and a mainstream transfer tool such as **curl**, **rsync**, or **wget**.

The model scripts below demonstrate how to download a set of files using **curl**. The local file names will be the same as those on the remote site, and if the transfer is interrupted for any reason, we can simply run the script again to resume the download where it left off.

Depending on the tools available on your local machine and the remote server, you may need to substitute another file transfer program for curl.

```
#!/bin/sh -e
```

```
# Download genome data from the ACME genome project
site=http://server.with.my.files/directory/with/my/files
for file in frog1 frog2 frog3 toad1 toad2 toad3; do
    printf "Fetching $site/$file.fasta.gz...\n"

    # Use filename from remote site and try to resume interrupted
    # transfers if a partial download already exists
    curl --continue-at - --remote-name $site/$file.fasta.gz
fi
```

```
#!/bin/csh -ef

# Download genome data from the ACME genome project
set site=http://server.with.my.files/directory/with/my/files
foreach file (frog1 frog2 frog3 toad1 toad2 toad3)
    printf "Fetching $site/$file.fasta.gz...\n"

    # Use filename from remote site and try to resume interrupted
    # transfers if a partial download already exists
    curl --continue-at - --remote-name $site/$file.fasta.gz
end
```

### 4.12.2 While Loops

A for or foreach loop is only convenient for iterating through a fixed set of values or a sequence generated by a program such as **seq**. Sometimes we may need to terminate a loop based on inputs that are unknown when the loop begins, or values computed over the course of the loop.

The **while** loop is a more general loop that iterates as long as some condition is true. It uses the same types of expressions as an **if** command. The while loop can be used to iterate through long integer sequences, as we might do with **seq** and a for/foreach loop:

```
#!/bin/sh -e

c=1
while [ $c -le 100 ]
do
    printf "%d\n" $c
    c=$(($c + 1))          # (( )) encloses an integer expression
done
```

Note again that the **[** above is an external command, as discussed in Section 4.11.1, so we must use white space to separate the arguments.

```
#!/bin/csh -ef

set c = 1
while ( $c <= 100 )
    printf "%d\n" $c
    @ c = $c + 1          # @ is like set, but indicates an integer expression
end
```

**Note**
Code controlled by a loop should be consistently indented as shown above. How much indentation is used is a matter of personal taste, but four spaces is typical.

**Practice Break**
Type in and run the script above.

While loops can also be used to iterate until an input condition is met:

```
#!/bin/sh -e

continue=''
while [ 0"$continue" != 0'y' ] && [ 0"$continue" != 0'n' ]; do
    printf "Would you like to continue? (y/n) "
    read continue
done
```

```
#!/bin/csh -ef

set continue=''
while ( ("$continue" != 'y') && ("$continue" != 'n') )
    printf "Continue? (y/n) "
    set continue="$<"
end
```

**Practice Break**
Type in and run the script above.

We may even want a loop to iterate forever. This is often useful when using a computer to collect data at regular intervals. It is up to the user to terminate the process using Ctrl+c or **kill**.

```
#!/bin/sh -e

# 'true' is an external command that always returns an exit status of 0
while true; do
    sample-data     # Read instrument
    sleep 10        # Pause for 10 seconds without using any CPU time
done
```

```
#!/bin/csh -ef

while ( 1 )
    sample-data     # Read instrument
    sleep 10        # Pause for 10 seconds without using any CPU time
end
```

### 4.12.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Describe three ways to run the same program using multiple input files.
2. Write a shell script that prints the square of every number from 1 to 10 using **for/foreach** and **seq**.

3. Write a shell script that prints the square of every number from 1 to 100 using **while**.
4. Write a shell script that sorts each file in the CWD whose name begins with "input" and ending in ".txt", and saves the output to `original-filename.sorted`. You may assume that there are not too many input files for a simple globbing pattern. The script then merges all the sorted text into a single file called `combined.txt.sorted`, with duplicate lines removed. Hint: The **sort** can also merge presorted files. Check the man page for the necessary flag.

```
shell-prompt: ./sort.sh

input1.txt:
Starbuck
Adama

input1.txt.sorted:
Adama
Starbuck

input2.txt:
Tigh
Apollo

input2.txt.sorted:
Apollo
Tigh

Combined:
Adama
Apollo
Starbuck
Tigh
```

## 4.13 Generalizing Your Code

All programs and scripts require input in order to be useful. Inputs commonly include things like scalar parameters to use in equations and the names of files containing more extensive data such as a matrix or a database.

### 4.13.1 Hard-coding: Failure to Generalize

All too often, inexperienced programmers provide what should be input to a program by hard-coding values and file names into their programs and scripts:

```
#!/bin/sh -e

# Hard-coded values 1000 and output.txt
calcpi 1000 > output.txt
```

Some programmers will even make another copy of the program or script with different constants or file names in order to do a different run. The problem with this approach should be obvious. It creates a mess of many nearly identical programs or scripts, all of which have to be maintained together. If a bug is found in one of them, then all of them have to be checked and corrected for the same error.

### 4.13.2 Generalizing with User Input

A more rational approach is to take these values as input:

```
#!/bin/sh -e

printf "How many iterations? "
read iterations
printf "Output file? "
read output_file

calcpi $iterations > $output_file
```

If you don't want to type in the values every time you run the script, you can put them in a separate input file, such as "input-1000.txt" and use redirection:

```
shell-prompt: cat input-1000.txt
1000
output-1000.txt
shell-prompt: calcpi-script < input-1000.txt
```

This way, if you have 10 different inputs to try, you have 10 input files and only one script to maintain instead of 10 scripts.

### 4.13.3 Generalizing with Command-line Arguments

Another approach is to design the script so that it can take command-line arguments, like most Unix commands. Using command-line arguments is quite simple in most scripting and programming languages. In all Unix shells, the first argument is denoted by the special variable `$1`, the second by `$2`, and so on.

`$0` refers to the name of the command as it was invoked. A script file can be renamed, so hard-coding the current name in error messages is a bad idea. Using `$0` eliminates future maintenance.

#### Bourne Shell Family

In Bourne Shell family shells, we can find out how many command-line arguments were given by examining the special shell variable `$#`. This is most often used to verify that the script was invoked with the correct number of arguments.

```
#!/bin/sh -e

# If invoked incorrectly, tell the user the correct way
if [ $# != 2 ]; then
    printf "Usage: $0 iterations output-file\n" >> /dev/stderr
    exit 1
fi

# Assign to named variables for readability
iterations=$1
output_file="$2"    # File name may contain white space!

calcpi $iterations > "$output_file"
```

```
shell-prompt: ./calcpi-script
Usage: calcpi-script iterations output-file
shell-prompt: ./calcpi-script 1000 output-1000.txt
shell-prompt: cat output-1000.txt
3.141723494
```

#### C shell Family

In C shell family shells, we can find out how many command-line arguments were given by examining the special shell variable `$#argv`.

```
#!/bin/csh -ef

# If invoked incorrectly, tell the user the correct way
if ( $#argv != 2 ) then
    printf "Usage: $0 iterations output-file\n" >> /dev/stderr
    exit 1
endif

# Assign to named variables for readability
set iterations=$1
set output_file="$2"     # File name may contain white space!

calcpi $iterations > "$output_file"
```

### 4.13.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Modify the following shell script so that it takes the starting and ending values of the loop as user input rather than hard-coding them.

```
#!/bin/sh -e

for c in $(seq 1 10); do
    c_squared=$(($c * $c))
    printf "%s^2 = %s\n" $c $c_squared
done
```

```
shell-prompt: ./squares.sh
First value to square? 3
Last value to square? 9
3^2 = 9
4^2 = 16
5^2 = 25
6^2 = 36
7^2 = 49
8^2 = 64
9^2 = 81
```

2. Repeat the above exercise, but use command-line arguments instead of user input.

```
shell-prompt: ./squares.sh
Usage: ./squares.sh first-value last-value

shell-prompt: ./squares.sh 4 10
4^2 = 16
5^2 = 25
6^2 = 36
7^2 = 49
8^2 = 64
9^2 = 81
10^2 = 100
```

## 4.14 Pitfalls and Best Practices

A very common and very bad practice in shell scripting is inferring things from the wrong information in order to make decisions. One of the most common ways this bad approach is used involves making assumptions based on the operating system in use. Take the following code segment, for example:

```
if [ `uname` == 'Linux' ]; then
    compiler='gcc'       # Compiler to use
    endian='little'      # Byte order for this CPU
fi
```

Both of the assumptions made about Linux in the code above were taken from real examples!

Setting the compiler to **gcc** because we're running on Linux is simply wrong, because Linux can run other compilers such as clang or icc. Compiler selection should be based on the user's wishes or the needs of the program being compiled, not the operating system. Also, **gcc** is the same as **cc** on most Linux systems (they are generally hard links to the same executable file). Hence, there is no reason to explicitly use **gcc** on Linux. The **cc** command is available on all Unix systems, being **gcc** on Linux and **clang** on FreeBSD, for example. So just using **cc** to compile programs is the safest default.

Assuming the machine is little-endian is wrong because Linux runs on a variety of CPU types, some of which are big-endian. The user who wrote this code assumed that if the computer is running Linux, it must be a PC with an x86 processor, which is not a valid assumption. The alternative for that user was an SGI IRIX workstation using a big-endian MIPS processor. Even if an operating system only runs on little-endian processors today does not mean the same will be true tomorrow. Hence, a check like this is a time-bomb, even if it's valid at the moment you write it.

There are simple ways to find out the actual endianness of a system, so why would we instead try to infer it from an unrelated fact? We should instead use something like the open source **endian** program, which runs on any Unix compatible system.

```
if [ `endian` == 'little' ]; then

fi
```

Moreover, users at this level should never have to worry about the endianness of a system. The fact that the user needed to check for this at the shell level indicates a serious design flaw in one of the programs he was using.

We can find out the exact CPU type using **uname -m** or **uname -p**. They usually report the same string, but on some platforms may produce different but equivalent strings such as "amd64" and "x86_64" or "arm64" and "aarch64".

### 4.14.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What can we infer about the hardware on which our script is running based on the name of the operating system?
2. How should a script decide what compiler to use to build programs?
3. How should a script go about determining what type of CPU your system uses?

## 4.15 Script Debugging

Sometimes a command in a script will fail for reasons that are not obvious. The -x flag is another flag common to both Bourne Shell and C shell that enables *execute tracing*, causing the shell to echo commands to the standard error before executing them. Using this, we can see the exact commands that are being executed *after* variable and globbing expansions. This might reveal a misconception we had about how to specify something.

We can tell a script to echo *every* command by adding a `-x` flag to the shell command:

```
#!/bin/sh -ex
```

```
#!/bin/csh -efx
```

Echoing every command is often overkill, however. In both Bourne shell and C shell, we can turn execute tracing on and off within the script:

```
#!/bin/sh -e

set -x      # Enable command echo
command
command
set +x      # Disable command echo
```

```
#!/bin/csh -ef

set echo    # Enable command echo
command
command
unset echo  # Disable command echo
```

### 4.15.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is execute tracing?
2. How do we enable execute tracing for an entire script?
3. How do we enable execute tracing for just a few commands in a script?

## 4.16 Functions, Child Scripts, and Aliases

Most shell scripts tend to be short, but even a program of 100 lines long can benefit from being broken down and organized into modules.

The Bourne family shells support simple functions for this purpose. C shell family shells do not support separate functions within a script, but this does not mean that they cannot be modularized. A C shell script can, of course, run other scripts and these separate scripts can serve the purpose of subprograms.

Some would argue that separate scripts are more modular than functions, since a separate script is inherently available to any script that could use it, whereas a function is confined to the script that contains it and is prone to get reinvented.

Another advantage of using separate scripts is that they run as a separate process, so they have their own set of shell and environment variables. Hence, they do not have side-effects on the calling script. Bourne shell functions, on the other hand, can modify variables in the main script and other functions, impacting the subsequent behavior in ways that will be difficult to debug.

### 4.16.1 Bourne Shell Functions

A Bourne shell function is defined by simply writing a name followed by parenthesis, and a body between curly braces on the lines below:

```
name()
{
    commands
}
```

We call a function the same way we run any other command.

```
#!/bin/sh -e

line()
{
    printf '--------------------------------------------------------------\n'
}

line
```

If we pass arguments to a function, then the variables $1, $2, etc. in the function will be set to the arguments passed to the function. Otherwise, $1, $2, etc. will be the command-line arguments passed to the script.

```
#!/bin/sh -e

print_square()
{
    printf $(($1 * $1))
}

c=1
while [ $c -le 10 ]; do
    printf "%d squared = %d\n" $c `print_square c`
    c=$((c + 1))
done
```

The **return** command can be used to return a value to the caller, much like **exit** returns a value from the main script. The return value is received by the caller in `$?`, just like the exit status of any other command.

```
#!/bin/sh -e

myfunc()
{
    if ! command1; then
        return 1

    if ! command2; then
        return 1

    return 0
}

if ! myfunc; then
    exit 1
fi
```

### 4.16.2 C Shell Separate Scripts

Since C shell does not support internal functions, we implement subprograms as separate scripts. Each script is executed by a separate child process, so all variables are local to that process.

We can, of course, use the **source** to run another script using the parent shell process as described in Section 4.5. In this case, it will affect the shell and environment variables of the calling script. This is usually what we intend and the very reason for using the **source** command.

When using separate scripts as subprograms, it is especially helpful to place the scripts in a directory that is in your PATH. Most users use a directory such as `~/bin` or `~/scripts` for this purpose.

### 4.16.3 Aliases

An alternative to functions and separate scripts for very simple things is the **alias** command. This command creates an alias, or alternate name for another command. Aliases are supported by both Bourne and C shell families, albeit with a slightly different syntax. Aliases are most often used to create simple shortcuts for common commands.

In Bourne shell and derivatives, the new alias is followed by an '='. Any command containing white space must be enclosed in quotes, or the white space must be escaped with a '\'.

```
#!/bin/sh -e

alias dir='ls -als'

dir
```

C shell family shells use white space instead of an '=' and do not require quotes around commands containing white space.

```
#!/bin/csh -ef

alias dir ls -als

dir
```

An alias can contain multiple commands separated by semicolons, but in this case it must be enclosed in quotes, even in C shell, since the semicolon would otherwise indicate the end of the **alias** command.

```
#!/bin/csh -ef

# This will not work:
# alias pause printf "Press return to continue..."; $<
#
# It is the same as:
#
# alias pause printf "Press return to continue..."
# $<

# This works
alias pause 'printf "Press return to continue..."; $<'

pause
```

### 4.16.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the advantage of using a Bourne shell function as opposed to running a separate script?
2. What are the advantages of using a separate script, as opposed to a Bourne shell function?
3. Show how to create an alias called **rls** which runs **find . -type f** in Bourne shell and C shell.

## 4.17 Here Documents

We often want to output multiple lines of text from a script, for instance to provide detailed instructions to the user. The output below is a real example from a script that generates random passphrases.

```
===========================================================================
If no one can see your computer screen right now, you may use one of the
suggested passphrases about to be displayed.  Otherwise, make up one of
your own consisting of three words separated by random characters and
modified with a random capital letters or other characters inserted.
===========================================================================
```

We could output this text using six **printf** commands or one **printf** and six string constants. This would be messy, though, and would require quotes around each line of text. We could also store it in a separate file and display it with the **cat** command, but this would mean more files to maintain.

A *here document*, or *heredoc* for short, is another form of redirection that is typically only used in scripts. It essentially redirects the standard input from a portion of the script itself. The general form is as follows:

```
command << end-of-document-marker

Content of the heredoc

end-of-document-marker
```

The content can contain anything except the marker. `End-of-document-marker` can be any arbitrary text of your choosing. You simply must choose a marker that is not in the text you want to display. Common markers are EOM (end of message) or EOF (end of file).

Heredocs can be used with any Unix command that reads from standard input, but are most often used with the **cat** or **more** command:

```
#!/bin/sh -e

cat << EOM
===========================================================================
If no one can see your computer screen right now, you may use one of the
suggested passphrases about to be displayed.  Otherwise, make up one of
your own consisting of three words separated by random characters and
modified with a random capital letters or other characters inserted.
===========================================================================
EOM
```

If the heredoc text may be more than one screen long, then use **more** instead of **cat**.

Heredocs can also be used to create files from a template that uses shell or environment variables. Any variable references and command output capture that appear within the text of a heredoc will be expanded. The output of any command reading from a heredoc can, of course, be redirected to a file or other device.

```
#!/bin/csh -ef

# Generate a series of test input files with different ending values
# and tolerances
foreach end_value (10 100 1000 10000)
    foreach tolerance (0.0001 0.0005 0.001)
        cat << EOM > test-input-$end_value-$tolerance.txt
start_value=1
end_value=$end_value
tolerance=$tolerance
EOM
    end
end
```

### 4.17.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a heredoc?
2. Can a heredoc contain anything besides literal text?

## 4.18 Scripting an Analysis Pipeline

### 4.18.1 What's an Analysis Pipeline?

An analysis pipeline is simply a sequence of processing steps. Since the steps are basically the same for a given type of analysis, we can automate the pipeline using a scripting language for the reasons we discussed at the beginning of this chapter: To save time and avoid mistakes.

A large percentage of scientific research analyses require multiple steps, so analysis pipelines are very common in practice.

### 4.18.2 Where do Pipelines Come From?

It has been said that for every PhD thesis, there is a pipeline. There are many preexisting pipelines available for a wide variety of tasks. Many such pipelines were developed by researchers for a specific project, and then generalized in order to be useful for other projects or other researchers.

Unfortunately, most such pipelines are not well designed or rigorously tested, so they don't work well for analyses that differ significantly from the one for which they were originally designed.

Another problem is that most of them are not maintained over the long term. Developers set out with good intentions to help other researchers, but once their project is done and they move onto new things, they find that they don't have time to maintain old pipelines anymore. Also, new tools are constantly evolving and old pipelines therefore quickly become obsolete unless they are aggressively updated.

**Note** The best use of most existing pipelines is as an example to following in developing a similar one.

Finally, many pipelines are integrated into a specific system with a graphical user interface (GUI) or web interface, and therefore cannot be used on a generic computer or HPC cluster (unless the entire system is installed and configured, which is often difficult or impossible).

For these reasons, every researcher should know how to develop their own pipelines. Relying on the charity of your competitors for publishing space and grant money will not lead to long-term success in research.

This is especially true for long-term studies. If you become dependent on a preexisting pipeline early on, and it is not maintained by its developers for the duration of *your* study, then the completion of your study will prove very difficult.

### 4.18.3 Implementing Your Own Pipeline

A pipeline can be implemented in any programming language. Since most pipelines involve simply running a series of programs with the appropriate command-line arguments, a Unix shell script is a very suitable choice in most cases. In some cases, it may be possible to use Unix shell pipes to perform multiple steps at the same time. This will depend on a number of things:

- Do the processing programs use standard input and standard output? If not, then redirecting to and from them with pipes will not be possible.

- What are the resource requirements of each step? Do you have enough memory to run multiple steps at the same time?
- Do you need to save the intermediate files generated by some of the steps? If so, then either don't use a Unix shell pipe, or use the **tee** command to dump output to a file and pipe it to the next command at the same time.

```
shell-prompt: step1 < input1 | tee output1 | step2 > output2
```

### 4.18.4 Example Genomics Pipelines

#### Genomic Differential Analysis

One of the most common types of bioinformatics analysis if *differential gene expression* using *RNA-Seq* (ribonucleic acid sequence) data. In this type of analysis, RNA molecules are extracted from tissue samples under two different conditions, such as two time points during development, or wild-type (normal) and mutant individuals. The amount of RNA transcribed from some genes will differ across the two conditions. From this we can infer that expression of those genes is somehow related to the conditions being compared.

The extracted RNA is sequenced, producing files containing one sequence for each RNA molecule (in theory), such as "ACUGGCAAUCGGAAAUA...". The differential analysis pipeline has the following high-level stages:

1. Clean up the sequences (remove artificial sequences added by the sequencing process)
2. Check the quality of the sequence data
3. Map the RNA sequences to a genome, so we know which gene produced each fragment
4. Count the fragments produced by each gene (not as simple as it sounds)
5. Compare the counts for each gene across the two conditions

If the counts for a given gene differ significantly between one condition and another, then we conclude that this gene is regulated (turned on or off) in response to the condition. A complete pipeline to perform this analysis on an HPC cluster is available at https://github.com/auerlab/cnc-emdiff. A similar pipeline that will run fairly quickly on a single computer is available at https://github.com/auerlab/fasda/tree/main/Test. This latter is a set of test scripts for FASDA, a program that performs the comparison of counts for each gene.

#### Metagenomics Example

Below is a simple shell script implementation of the AmrPlusPlus pipeline, which, according to their website, is used to "characterize the content and relative abundance of sequences of interest from the DNA of a given sample or set of samples".

People can use this pipeline by uploading their data to the developer's website, or by installing the pipeline to their own Docker container or Galaxy server.

**Note** At the time of this writing, November 2022, the AmrPlusPlus pipeline source on Github has not been updated for over five years. This is typical of scientific pipelines for the reasons stated above.

In reality, the analysis is performed by the following command-line tools, which are developed by other parties and freely available:

- Trimmomatic
- BWA
- Samtools

- SNPFinder
- ResistomeAnalyzer
- RarefactionAnalyzer

The role of AmrPlusPlus is to coordinate the operation of these tools. AmrPlusPlus is itself a script.

If you don't want to be dependent on their web service, a Galaxy server, or their Docker containers, or if you would like greater control over and understanding of the analysis pipeline, or if you want to use the newer versions of tools such as samtools, you can easily write your own script to run the above commands.

Also, when developing our own pipeline, we can substitute other tools that perform the same function, such as Cutadapt in place of Trimmomatic, or Bowtie in place of BWA for alignment.

All of these tools are designed for "short-read" DNA sequences (on the order of 100 base pair per fragment). When we take control of the process rather than rely on someone else's pipeline, we open the possibility of developing an analogous pipeline using a different set of tools for "long-read" sequences (on the order of 1000 base pair per fragment).

For our purposes, we install the above commands via FreeBSD ports and/or pkgsrc (on CentOS and Mac OS X). Then we just write a Unix shell script to implement the pipeline for our data.

Note that this is a real pipeline used for research at the UWM School of Freshwater Science.

It is not important whether you understand genomics analysis for this example. Simply look at how the script uses loops and other scripting constructs to see how the material you just learned can be used in actual research. I.e., don't worry about what **cutadapt** and **bwa** are doing with the data. Just see how they are run within the pipeline script, using redirection, command line arguments, etc. Also read the comments within the script for a deeper understanding of what the conditionals and loops are doing.

```
#!/bin/sh -e

# Get gene fraction threshold from user
printf "Resistome threshold? "
read threshold

############################################################################
# 1. Enumerate input files

raw_files="SRR*.fastq"

############################################################################
# 2.  Quality control: Remove adapter sequences from raw data

for file in $raw_files; do
    output_file=trimmed-$file
    # If the output file already exists, assume cutadapt was already run
    # successfully.  Remove trimmed-* before running this script to force
    # cutadapt to run again.
    if [ ! -e $output_file ]; then
        cutadapt $file > $output_file
    else
        printf "$raw already processed by cutadapt.\n"
    fi
done

############################################################################
# 3. If sequences are from a host organism, remove host dna

# Index resistance gene database
if [ ! -e megares_database_v1.01.fasta.ann ]; then
    bwa index megares_database_v1.01.fasta
fi
```

```
############################################################################
# 4. Align to target database with bwa mem.

for file in $raw_files; do
    # Output is an aligned sam file.  Replace trimmed- prefix with aligned-
    # and replace .fastq suffix with .sam
    output_file=aligned-${file%.fastq}.sam
    if [ ! -e $output_file ]; then
        printf "\nRunning bwa-mem on $file...\n"
        bwa mem megares_database_v1.01.fasta trimmed-$file > $output_file
    else
        printf "$file already processed by bwa mem\n"
    fi
done

############################################################################
# 5. Resistome analysis.

aligned_files=aligned-*.sam
for file in $aligned_files; do
    if [ ! -e ${file%.sam}group.tsv ]; then
        printf "\nRunning resistome analysis on $file...\n"
        resistome -ref_fp megares_database_v1.01.fasta -sam_fp $file \
            -annot_fp megares_annotations_v1.01.csv \
            -gene_fp ${file%.sam}gene.tsv \
            -group_fp ${file%.sam}group.tsv \
            -class_fp ${file%.sam}class.tsv \
            -mech_fp ${file%.sam}mech.tsv \
            -t $threshold
    else
        printf "$file already processed by resistome.\n"
    fi
done

############################################################################
# 6. Rarefaction analysis?
```

I generally write a companion to every analysis script to remove output files and allow a fresh start for the next attempt. Many programs will detect when output already exists and either not attempt to rewrite it, or behave differently in other ways.

```
#!/bin/sh -e

rm -f trimmed-* aligned-* aligned-*.tsv megares*.fasta.*
```

### 4.18.5 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is an analysis pipeline?
2. Where do most analysis pipelines come from?
3. How useful are most pipelines to future projects?
4. What is the best way to use most existing pipelines?
5. What languages can be used to implement an analysis pipeline? Which are likely to be convenient?
6. What is a good way to restart a pipeline from the beginning?

## 4.19 Solutions to Practice Breaks

Script to search a file:

```
#!/bin/sh -e

ls

printf "Enter the name of the file to search: "
read file_name

if [ -z "$filename" ]; then
    printf "File name cannot be empty.\n" >> /dev/stderr
    exit 65
fi

if [ ! -f $file_name ]; then
    printf "$file_name is not a regular file.\n" >> /dev/stderr
    exit 65
fi

head -n 5 $file_name

printf "Enter a string to search for in $file_name: "
read key

if [ -z "$key" ]; then
    printf "Search string cannot be empty.\n" >> /dev/stderr
    exit 65
fi

fgrep $key $file_name
```

# Chapter 5

# Data Management

## 5.1 What is Data Management?

Data management, in the context of research computing, refers to how research data is stored, formatted, and disseminated over the long term.

Researchers who generate new data need to plan ahead in order to ensure that any data supporting their research conclusions will be available in the future. The data could be used to reproduce or otherwise verify the research results, or could be analyzed in new ways for completely different purposes.

## 5.2 Why Worry?

Confidence in science is based on transparency, and the ability to reproduce results by repeating experiments. Without access to the raw data generated by, or used in your experiments, your experiments cannot be verified by others.

Given the often high cost of generating data, preserving data for future use can often prove to be far more cost-effective.

The National Science Foundation is now requiring all applicants to submit a detailed plan for preserving and disseminating their research data. I.e., you will not be rewarded a grant from the NSF unless you clearly state your plans for long-term data management.

## 5.3 Storage Logistics

There are many issues you will need to consider in order to plan well for data management. A few of those issues are discussed in the sections that follow.

### 5.3.1 Data Format

When storing data only for yourself, you might not give this issue much thought. However, data management includes not only preservation, but dissemination. If others will have access to your data, it must be in a format that is easy for them to read.

Many areas of science have developed standard data formats to help researchers and software applications interoperate. Research needs are too diverse to cover all of the standard data formats here. The goal of this section is only to raise awareness and encourage researchers to explore available standards before digging themselves into a hole.

**Caution** Changing the format of large amounts of your data at a later time could be a very frustrating and costly process. It is highly advisable to decide on a standard data format before your research progresses too far.

The best way to explore data formats is by talking to others in your specific field and studying options via the Internet. This will help you develop a sense of what the emerging standards are in your niche.

### 5.3.2 Lifespan

Another very important question to ask is how long the data should be preserved. This will impact the cost of data management, although not as much as you might think, assuming that storage costs continue to decline over the long term.

Generally, the harder it is to regenerate the data, the longer it should be preserved. Data that are easy to recreate may actually cost more to store.

### 5.3.3 Security

If the data contain confidential information, such as personal health information (PHI) or financial records, it may be necessary to restrict and track access to it. Regulations on PHI data are strict and somewhat complex, so they should be explored before making any data management plans.

### 5.3.4 Safety

Data safety refers to the risk of data loss. If you're using a service provider to store your data, this will generally be their responsibility. They will maintain backups of data they store and provide a written guarantee about its availability.

If you are storing the data yourself, you'll need to think about how to back it up and where. Backups should always be stored far from the original data in order to protect against fire, theft, and other physical disasters. Backups in the same room are not safe at all. Backups in the same building are somewhat more safe, while backups in a separate building or distant location are best.

### 5.3.5 Funding

Paying for long-term data storage is a complex issue. Depending on the cost involved, it may or may not be possible to pay for it from a one-time grant allocation. Some institutions may provide data storage services, but in most cases, researchers will have to make their own arrangements, such as purchasing hardware or purchasing storage space from a commercial service.

### 5.3.6 Storage Providers

There are a number of organizations that provide long-term data storage, provided by Universities, government organizations, and private companies.

NIH's GenBank is publicly-funded and stores genomic data at no charge to researchers, for the benefit of future medical and other biological research. Commercial services offer very low-cost options, provided you do not need high-speed upload or download. The cost increases along with the desired transfer speed. The best approach to selecting one is investigating their *current* service offerings and talking to colleagues who have been down this road.

### 5.3.7 Managing Your Own Storage

If you must manage your own backup or archival storage for reasons of privacy, funding, etc., there are cost-effective and reliable ways to do it.

The worst option is a USB thumb drive or other external disk that plugs into an interface on your computer such as USB. Such devices are easily damaged, lost, disconnected, turned off, or stolen. Accidental disconnections or power loss can lead to damaged file systems and lost files.

When using such a device, you are also limited to the file systems supported by the computer you plug it into. E.g. if you format it for BSD, Linux, or Mac, you won't be able to plug it into a Windows PC.

A much safer option that’s nearly as cost-effective is a networked file server with a built-in RAID (redundant array of inexpensive disks). With limited skills, you can build a file server using an inexpensive PC with two or more disks, and a specialized storage OS such as TrueNAS or XigmaNAS. These FOSS (Free Open Source Software) products use the advanced ZFS file system to provide redundancy in case of a disk failure, as well as data compression, encryption, snapshots, etc.

They are extremely easy to install and manage through a simple graphical interface. If you’re prepared to spend a little more, you can also purchase a preconfigured TrueNAS box with commercial support.

You can house the file server anywhere, but preferably in a secure location with battery-backed power, such as a data center. If you don’t have access to a data center, then choose the most secure location you can and purchase a small UPS (uninterruptable power supply) along with the PC to protect it from brief power outages.

TrueNAS and XigmaNAS support all common network protocols such as NFS (Unix Network File System), SMB/CIFS (Windows disk sharing), and AFS (Apple’s networked file system), so files on the server can be simultaneously accessed from any computer on the network.

## 5.4 Data Storage on the Cluster

Storage on clusters is generally designed for speed, not long-term capacity or data safety. Data storage on clusters should be considered temporary space. Space is limited, and users are constantly and rapidly generating new data. Hence, it is important for all users to move data off the cluster as soon as possible in order to keep cluster storage available for other jobs.

This does not mean that you cannot leave data on the cluster for further analysis. However, all data generated on the cluster should be *immediately* copied to another location after being generated, so that it will be safe from disasters such as hardware failures or fires. It should be removed from the cluster as soon as it is safely stored in *two* other locations where it is accessible for further analysis.

## 5.5 Data Transfer

Storage is not the only problem associated with big data. It also presents challenges with transferring data, especially over great distances and across different computer platforms.

This can be particularly problematic for small organizations that do not have a very high bandwidth Internet connection. While the Internet backbone may provide plenty of speed to transfer your research data in a reasonable amount of time, the connection from the Internet into your building may be a severe bottleneck. This is known as the *last mile problem*.

One potential solution to this problem is to avoid transferring the data in the first place. Some organizations offer web-based tools to allow users elsewhere to perform common analyses on their data without first downloading it. For example, if you want to search for a DNA sequence in the genomes of many organisms, you can do so on the NCBI BLAST website. This means uploading a short DNA sequence to the NCBI server rather than downloading many gigabytes of genome data.

Another potential solution is to simply perform a more selective transfer. Determining exactly which parts of the data to transfer can involve a lot of manual labor, but it may save many hours or more of transfer time.

Sometimes the problem is not bandwidth, but user interface. The most common type of data transfer utilized ordinary tools like a web browser or FTP client. These methods are collectively known as "data schlepping". Data schlepping requires the user to use a variety of tools to transfer data to and from various sites. It also often suffers from failures due to dropped network connections, power outages, and other issues that are likely to interrupt a long transfer. Some tools, such as **rsync**, allow an interrupted transfer to continue from where it left off. However, not all sites offer **rsync** service.

Globus Transfer is an example of a web-based alternative for data transfer that has built-in capabilities for dealing with connection issues, login credentials, and many other data transfer issues. It also overcomes bottlenecks associated with long-distance file transfers. Downloading data with a web browser or **curl** over thousands of miles typically results in transfer speeds of 1 or 2 megabytes per second. Globus can often transfer over such distances at 50 megabytes per second. The down side is that Globus and comparable high-speed transfer tools are commercial, and require a license and expertise to install and configure.

Data transfer tools are evolving rapidly in response to the growing needs presented by big data. Users should make it a habit to continuously explore and reevaluate new and existing options.

## 5.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is ultimate the goal of scientific data management?
2. When do we need to be concerned about the long-term storage of data?
3. Who must devise a data management plan?
4. Why is storage format an important consideration?
5. How long must data be stored?
6. Should data be made freely available?
7. How do we ensure data safety?
8. How much does it cost to store data?
9. What is the safest and most convenient type of local storage hardware? Why?
10. What is the least safe type of local storage hardware? Why?
11. When using an HPC cluster, why not just leave our data on the cluster?
12. Describe three possible ways to overcome the problem of transferring large amounts of data over a distance.

# Part II

# Parallel Computing

# Chapter 6

# Parallel Computing

## 6.1 Introduction

### 6.1.1 Motivation

In 1976, Los Alamos National Laboratories purchased a Cray-1 supercomputer for $8.8 million. As the world's fastest computer at the time, it had 8 mebibytes (a little over 8 million bytes) of main memory and was capable of 160 million floating point operations per second. (Source: http://www.cray.com/About/History.aspx)

The first draft of this manual was written in September 2010 on a $700 desktop computer with a gibibyte (a little over 1 billion bytes) of main memory, and capable of over 2 billion floating point operations per second.

It may seem that today's computers have made the supercomputer obsolete. However, there are still, and always will be many problems that require far more computing power than any single processor can provide. There are many examples of highly optimized programs that take months to run even with thousands of today's processors. The volume and resolution of raw data awaiting analysis has exploded in recent years due to advances in both research techniques and technology. Enormous amounts of new data are being generated every day, and new ways to analyze old data are constantly being discovered.

### 6.1.2 Computing is not Programming

It is important to understand the difference between *parallel computing* and *parallel programming*. Parallel computing includes any situation where multiple computations are being done at the same time.

This often involves running multiple instances of the same serial (single-processor) program at the same time, each with different inputs. Typically, there is no communication or cooperation between the multiple instances as they run. This scenario is known as *embarrassingly parallel* computing.

Parallel programming, on the other hand, involves writing a special program that will utilize multiple processors. The code running on each processor will exchange information with the others in order to work together toward a common goal. This is much more complex than embarrassingly parallel computing, but is necessary for many problems.

There are many types of parallel architectures, and each is suited for specific types of problems. Writing parallel programs is not a process that can be easily generalized. Understanding of specific algorithms is crucial in determining *if* and *how* they can be decomposed into independent subtasks, and what will be the most suitable parallel architecture on which to run them. Some of the common parallel architectures are outlined in the following sections.

### 6.1.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How much has computing power increased since the 1970s?
2. Is there still a need for parallel computing, given the advances in computing?

## 6.2 Shared Memory and Multi-Core Technology

Around the year 2000, processor manufacturers hit what is known as the *power wall*, a barrier to clocking computer processors faster than about 3 GHz due to the inability to dissipate the heat generated. The laws of thermodynamics indicate that all energy put into any system (machine, biological organism, solar cell, etc) is ultimately converted to heat. Increasing the speed of a CPU causes it to generate heat faster. All heat generated must be conducted through the materials in the chip to the surface and then dissipated by a cooling system. While this hurdle will likely be overcome eventually, the industry has realized that processor clock speeds cannot grow indefinitely, and have therefore turned their focus toward improving efficiency and parallelism.

As a result, most personal computers now come with multiple *cores*. Core is the modern term for what has been traditionally known as the *Central Processing Unit (CPU)* or simply *processor*. The term *core* refers to a *functional* CPU. For a long time before the age of multi-core technology, processor chips traditionally contained a single core, so CPU and core could be used synonymously. Hence, the term *CPU* has been widely used to refer to either the *functional* CPU or the *physical* chip. Now that the assumption of one CPU per chip is no longer valid, the term *CPU* has become somewhat ambiguous. It could refer to either a core, or a chip containing multiple cores. The term *core* is preferred when referring to a functional CPU.

Each core is capable of running it's own *thread*, or sequence of instructions. Hence, a multi-core CPU chip can run multiple processes at the same time, or multiple instructions of the same process.

Since these multiple cores are part of the same computer, they all have access to the same memory bank. This allows the multiple threads to share information very easily. The disadvantage of shared memory is that only one core can access a memory chip at a given time. As we increase the number of cores, we increase the likelihood of contention for accessing each memory chip, just as a bigger family has more contention for the bathroom. Therefore, the number of cores that can truly work in parallel is very limited, i.e. shared-memory architectures don't *scale* well. Adding more than a few dozen cores to a computer typically shows diminishing returns in terms of speedup. E.g. a 128-core computer won't be able to do twice as much computation as a 64-core computer in the same amount of time, since cores spend more time waiting for their turn to access memory.

### 6.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the power wall and what causes it?
2. How has the industry responded to the power wall?
3. Explain the difference between a core and a CPU.
4. What is shared memory parallelism?
5. What is the main advantage of shared memory parallelism from the programmer's perspective?
6. What is the main disadvantage of shared memory parallelism from the programmer's perspective?

## 6.3 Distributed Parallel Computing

### 6.3.1 Overview

In order to achieve higher degrees of parallelism than feasible in shared-memory systems, work must be *distributed* among independent processors that don't contend for shared resources like memory. Instead of sharing memory, processes in a distributed system run on separate computers that each have their own memory. The processes communicate with each other by passing messages over a network. Message passing is not as easy to program or as fast as shared memory, but it does allow for much greater numbers of cores to be used by cooperating processes in many cases. In other words, it scales better than shared-memory parallelism.

In distributed parallel computing, a *job* consists of multiple processes running on the same or different computers in order to solve a single problem.

Creating a distributed parallel job could be as simple as writing a script containing several remote execution commands using a tool like **ssh**. However, multiple users running such jobs without some sort of traffic control could quickly lead to chaos. The solution to this problem, scheduling software, is discussed in Section 7.3.

Distributed parallel computing environments are generally divided into *clusters* and *grids*, which are covered in the following sections.

In both grids and clusters, the various computers are referred to as *nodes*. A user logs into one of the nodes, known as a *head node* or *submit node*, and starts a job that dispatches processes on some of the other nodes, which are known as *compute nodes*.

### 6.3.2 Clusters and HPC

A *cluster* is a group of commodity computers with a dedicated high-speed network and high-speed disk storage that is directly accessible to all the computers in the cluster. The dedicated network and shared disk allow processes on a cluster to communicate heavily with each other without overloading the office or campus network. The computers within a cluster are usually located in the same room, and often mounted together in a refrigerator-sized racks.

*Peregrine, A Small Educational Cluster*

*Mortimer, A 2000-core Research Cluster*

If it's possible to decompose a large computational job into somewhat independent sub-tasks, a cluster can provide a huge speed-up by deploying the work to hundreds or even thousands of cores at the same time. This type of computing is often referred to as *high performance computing (HPC).*

Most clusters also offer shared disk space, so that the same files are directly accessible to all nodes in the cluster. The shared space is a more or less typical file server shared by all the nodes, although on large clusters, it may be implemented using a special parallel file system designed specifically for HPC.

As stated earlier, clusters are suitable for problems that can be decomposed into a large number of relatively independent tasks that can be distributed among multiple computers to run in parallel. If decomposing a problem results in tasks that must share extremely large volumes of data, then a shared memory architecture may provide better performance. Fortunately, there are many large computational problems that adapt well to the distributed computing model offered by clusters.

There are several types of nodes in a typical cluster:

- The *head node* is responsible for running the job scheduler and possibly other system tasks.
- A *login node* is where users log in to run Unix shell commands. Users typically use the login node to edit scripts, submit jobs to the scheduler, and monitor their jobs. On a small to medium sized cluster, the head node typically serves as the login node. Very large clusters may provide one or more login nodes separate from the head node.
- *Compute nodes* run the processes that make up scheduled jobs. Most of the nodes in a cluster are compute nodes. Compute nodes typically have faster processors and more RAM than head and login nodes.
- *I/O nodes* are the file servers in a cluster. I/O nodes typically run NFS (the Unix Network File system Service) on small and medium clusters, or a more sophisticated parallel file system on large clusters. In either case, the login nodes and all compute nodes all have access to the files on the I/O nodes.
- A *visualization node* is another node that users can log into in order to run shell commands. However, a visualization node is meant to run graphical software for viewing the results of jobs run on a cluster.

  Note that using a visualization node usually involves rendering graphics over a network, i.e. the graphical program is running on the visualization node and displaying on a different machine, probably your workstation, through the network. This will be slower than running a graphical program on your workstation and displaying directly to the attached monitor. However, it may save time if it eliminates the need to frequently download large files to be visualized.

### 6.3.3 Grids and HTC

A *grid* made up of loosely coupled computers which may be in multiple distant locations. Grids often utilize spare CPU time on existing office or lab PCs. No additional hardware is generally required to form a grid from existing PCs on a network.

*A Small Grid*

A *grid* is similar to a cluster, but without the high-speed communication network and high-speed shared storage typically seen on clusters. Grids often utilize hardware that was not designed and built primarily for parallel computing. Instead, grids are commonly implemented to take advantage of already existing hardware that may not otherwise be well utilized, such as PCs in college computer labs. Lab PCs tend to be heavily utilized occasionally (during classes) but idle most other times.

Grids are often implemented with no improvements to the existing hardware infrastructure. Since the computers are already installed and connected by a network, a grid can be often implemented simply by installing and configuring grid management software such as the University of Wisconsin's HTCondor system, available at http://research.cs.wisc.edu/htcondor/. It is not even necessary for all the computers to run the same operating system, although standardizing will make the grid easier to use. The standardization of PC configurations can be rendered unnecessary using virtual machines to serve as compute hosts. Virtual Machine software such as VirtualBox can run the same standard guest operating system on a wide variety of platforms.

Since machines on a grid communicate over a standard, non-dedicated campus or office network, extensive message-passing between parallel processes or high-volume shared file servers would be likely to overload the network. Therefore, some of the programs that we typically run on clusters are not suitable for grids.

Grids are only suitable for algorithms that can be decomposed into highly independent processes that require little or no communication with each other while running, i.e. *embarrassingly parallel* computing. Grid users often simply run many instances of serial programs at the same time. In other words, grid users often do parallel *computing* without parallel *programming*.

Grid computing is often referred to as *High Throughput Computing*, or *HTC*, since the lack of communication between processes eliminates potential bottlenecks that might slow the progress of the computations and reduce throughput.

Many campuses are implementing grids to provide an inexpensive parallel computation resource for students and researchers. One of the leaders in this area is Purdue University, where every computer on the entire campus is part of a massive grid.

### 6.3.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is distributed parallel computing, as opposed to shared memory?
2. What is the advantage of distributed parallel computing as opposed to shared memory?

3. What is the disadvantage of distributed parallel computing as opposed to shared memory?
4. What is a high-performance computing cluster?
5. What is a grid, and how is it similar to and different than a cluster?
6. What are clusters good for that grids are not? Why?
7. What are grids good for?

## 6.4 Multiple Parallel Jobs

Very often, users need to run multiple independent (embarrassingly parallel) jobs, each of which is running a parallel program. The question then is how many cores to use for each run of the parallel program. Since doubling the number of cores used by a parallel program generally provides less than a two-fold speedup, it is likely that using fewer cores for each job in this situation will improve the overall run time if the same number of cores are used in total.

E.g., running 100 jobs at a time using two cores each will likely run faster than 25 jobs using 8 cores each. The only way to be sure is by trying it both ways. Running the same job twice is a waste of resources, and ascertaining which method is faster by comparing different jobs may not be feasible. Generally, fewer cores means lower communication overhead and higher throughput, so in the absence of any hard data, keep the core counts fairly low for your best shot at maximizing throughput.

## 6.5 Graphics Processors (Accelerators)

Today's *video processors*, also known as *graphics processing units (GPUs)*, require an enormous amount of computing power in order to quickly render the rapidly changing 3-dimensional graphics of animation software onto a 2-dimensional screen.

Most of this development was initially driven by the video game industry. Ironically, a multi-billion dollar industry was pushed forward largely by people whose primary mode of transportation is a skateboard. Never underestimate the power of a grassroots movement.

Scientists eventually recognized that the processing power of GPUs could be harnessed for scientific computations. In some cases, a GPU can offer much faster processing than a traditional CPU. However, utilizing GPUs requires rewriting software using advanced tools and libraries. Hence, GPUs are not useful for the majority of scientific computing. We cannot just add a GPU to our PC and magically watch all of our programs run 1000 times faster.

We can make a few general statements about GPU computing:

1. It's here to stay.
2. It offers significant performance benefits for a small segment of scientific computing.
3. It's difficult.

Like every new technology, GPU computing quickly became over-hyped. It became a solution looking for problems. The hype has subsided as people became aware of how much effort is involved in utilizing GPUs and where the added programmer hours are actually a worthwhile investment.

GPU boards are now being developed and marketed explicitly for scientific computation rather than graphics rendering. These processors are more aptly called *accelerators*, but the term GPU is still widely used.

CUDA is a proprietary system for utilizing nVidia video processors in scientific computing. Using a system such as CUDA, programmers can boost the performance of programs by running computations in parallel on the CPUs and the GPUs.

One caveat is that while GPUs have a great deal of processing power, they are designed specifically for graphics rendering, and are therefore not well suited to every computational application. The interface to a GPU may lend itself very well to some applications, and not so well to others. The subject is complex, and readers are advised to research whether GPUs would fit their needs before investing in new hardware.

Nevertheless, some sites have invested in clusters with large numbers of GPUs in order to gain the best performance for certain applications that lend themselves well to GPU computing. A fair number of GPU-based programs have already been developed and having access to a pool of GPUs allows such software to be easily utilized. They are especially popular in machine-learning applications, which require enormous amounts of computation.

OpenCL is an alternative to CUDA which is fully open source, supports GPUs from multiple vendors rather than just nVidia, and can also utilize CPUs. With OpenCL, the same parallel code can utilize both CPU and GPU resources, so programming effort is not duplicated to take advantage of different hardware. Work is also underway to bring GPU support to *OpenMP*.

### 6.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What will adding a GPU/accelerator do for most of your current software? Why?
2. Is a GPU a good investment for scientific computing?

## 6.6 Best Practices in Parallel Computing

### 6.6.1 Parallelize as a Last Resort

While there will always be a need for parallel computing, the availability of parallel computing resources may tempt people to use them as a substitute for writing efficient code.

There is virtually no optimal software in existence. Most software at any given moment can be made to run faster, and a significant percentage of it can be made to run orders of magnitude faster. Most performance issues can and should therefore be resolved by optimizing the software first.

Improving software is a more responsible and intelligent way to resolve performance issues wherever it's possible. It will allow effective use of the software on a much wider variety of hardware, possibly including ordinary desktop and laptop machines. It is much better for everyone if they can run a program on a laptop or desktop system rather than use a cluster or grid.

It is also the more ethical way to resolve issues where users are running on shared resources. Using tens of thousands of dollars worth of computer equipment to make inefficient software run in a reasonable amount of time is wasteful and foolish, and may delay the work of others who need those resources for more legitimate uses.

I once had a computer science student who worked as a consultant. His firm was hired to design and install a faster computer for a business whose nightly data processing had grown to the point where it wasn't finishing before the next business day started. He politely asked if he could have a look at the home-grown software that was performing the overnight processing. In about an hour, he found the bottleneck in the software and made some adjustments that reduce the processing time from 14 hours to about 10 minutes.

I've personally experienced numerous cases where researchers were considering buying a faster computer or using a cluster to speed up their work. In many cases, I was able to help them make their programs run orders of magnitude faster and eliminate the need for more hardware. In one case, in about 20 minutes of examining a Matlab script, I identified the location of a bottleneck that the researcher then easily solved, making the script run about 1,000 times as fast. He was extremely grateful that he could not complete his work on his PC, and other cluster users did not suffer from this needless competition for resources.

Before you consider the use of parallel computing, make sure you've done everything you can to optimize your software, by choosing efficient algorithms, using compiled languages for the time-consuming parts of your code, and eliminating wasteful code.

Software performance is discussed in greater detail in Part III.

### 6.6.2 Make it Quick

Parallel computing jobs should be designed to finish within a few hours, if possible. Jobs that run for weeks or months have a lower probability of completing successfully. The longer individual processes run, the higher the risk risk of being interrupted by power failures, hardware issues, security updates, etc. In the case of HTCondor grids, jobs that run for more than a few hours run a high risk of being evicted from desktop machines that are taken over by a local user.

Shorter running jobs also give the scheduler the ability to balance the load fairly among many users.

In the case of HTC (embarrassingly parallel computing), we can usually break up the work into as many pieces as we like. For example, we may have 3,000 core-hours with of computation that we can run as a hundred 30-hour processes, or a thousand 3-hour processes, simply by dividing up the input into smaller chunks. We do not want to divide it to the point where each job takes only a few minutes, however. Doing so increases overhead and reduces throughput.

### 6.6.3 Monitor Your Jobs

Never submit a job and ignore it. Malfunctioning jobs waste expensive resources that could be utilized by others and may actually cause compute nodes to crash in extreme cases. It is every users responsibility to ensure that their jobs are functioning properly and not causing problems for other users.

### 6.6.4 Development and Testing Servers

A well-organized software development operation consists of up to five separate environments (tiers) for developing, testing, and ultimately using new software.

1. Development
2. Testing
3. Quality assurance
4. Staging
5. Production

A detailed description of the five tiers can be found in the Wikipedia article on Deployment Environments. Typically only organizations with a large software development staff will employ all five tiers.

Even in the smallest environments, though, a clear distinction is made between development/testing servers and production servers. Clusters and grids are meant to be production environments. A cluster or grid is not a good place to develop or test new programs, for multiple reasons:

- All jobs run on a cluster or grid must go through a scheduler, which makes the development and testing process more cumbersome.
- We generally don't want test runs of unfinished or unfamiliar code competing with production jobs. Bugs in the code or mistakes in using it will often have unforeseen impact on the system, which may harm important production jobs by consuming too much memory or even crashing compute nodes.
- A cluster or grid is not necessary for testing code correctness, even for parallel programs. Most testing of any program, including parallel programs, can be done on a single computer, even with a single core. All we need is a multitasking operating system that can run multiple processes at the same time. We do not need parallel hardware resources. The only testing that requires parallel hardware resources is measuring speedup.

These principles apply whether you are developing your own code or learning to use a program written by others. Code should be developed and tested on servers completely separate from the scheduled production environment of a cluster or grid. This is easily done using a separate server with the same operating system and software installed as a compute node on the production cluster or grid.

Think of development server as a compute node where you are free from the need to schedule jobs, or worries about impacting the important work of other users. Here you can quickly and easily make changes to your program and run small test cases. Once you are confident that the code is working properly, you can move up to the next available tier for additional testing or production runs.

### 6.6.5 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. When should we make the decision to use parallel computing? Why?
2. How long should parallel jobs run ideally? Why?
3. Why is it important to actively monitor jobs running on a cluster or grid?
4. What are some problems associated with doing code development on a cluster or grid?
5. What do we need to test the correctness of a parallel program?

# Chapter 7

# Job Scheduling

**Before You Begin**
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 3), the Unix shell (Section 3.3.3, redirection (Section 3.13), and shell scripting (Chapter 4).

## 7.1 Fair Share Guidelines

**Caution**
BEFORE you start submitting jobs to a cluster or grid, you MUST know how to monitor and control them.
Jobs that go rogue can cause problems for other users, so you need to know how to watch over them to ensure they're not using more resources than expected.
If a job does go astray, you need to know how to terminate it quickly, before is impacts other users.

If a job is submitted and the resources required to run it are not available, the job waits in a queue until the resources become available. These *pending* jobs are, for the most part, started in the order they were submitted. Hence, if one user submits too many jobs at once, other users' jobs may end up waiting in the queue for a very long time.

As a general rule, if you are already using a significant share of of the total cluster resources *and* there are jobs pending, you should not submit new jobs until other users' jobs have a chance to run.

If there are a lot of idle resources on the cluster, it's generally better to utilize them by submitting some "extra" jobs than to let them remain idle. The ideal state for a cluster is nearly 100% utilized, but without jobs pending for a long time.

However, keep in mind that other users may need resources *after* you've submitted. Hence, if you're using a lot of resources, please check the cluster load once per hour. If pending jobs appear after you've submitted, and none of your jobs are near completion, please kill some of them to allow other users a fair share of cluster resources.

## 7.2 Remote Execution

Since the early days of Unix, it has been possible to run commands on remote computers over a network connection using *remote shell* programs such as **rsh** and **ssh**:

```
mypc: ssh mylogin@peregrine.hpc.uwm.edu ls
Password: (enter password here)
Data
Desktop
Work
```

```
        notes
        scripts
```

**Caution** Older protocols such as rlogin, rsh, and telnet, should no longer be used due to their lack of security. These protocols transport passwords over the Internet in unencrypted form, so people who manage the gateway computers they pass through can easily read them.

On a typical network, the above command would prompt the user for a password to log into `peregrine` as the user bootcamp, run the **ls** command on `peregrine`, and then exit back to the local host from which **ssh** was run.

It is possible to configure remote computers to accept *password-less* logins from trusted hosts. With a password-less login, we can run a command on a remote system almost as easily as on the local system:

```
        mypc: ssh mylogin@peregrine.hpc.uwm.edu
        Password: (enter password here)
        peregrine: ssh compute-001 ls
        Data
        Desktop
        Work
        notes
        scripts
```

In the example above, the node `compute-001` does not ask for a password, since the request is coming from `peregrine`, which is a trusted host.

**Key Point** Password-less login makes it convenient to automate the execution of commands on many remote machines. This idea forms the basis for *distributed parallel computing*.

### 7.2.1 Self-test

1. Show a command for doing a long-listing of the `/etc` directory on the host some.unix.machine.edu, for which you have an account with login name "bob".
2. What is passwordless login?

## 7.3 Scheduler Basics

### 7.3.1 Purpose of a Scheduler

Unix systems can divide CPU *cores* and memory resources among multiple processes. Memory is divided up physically, and CPU time is divided temporally, with time slices of about 1/100 to 1/1000 of a second given to each process in a mostly round-robin fashion. This type of CPU sharing is known as *context switching* or *preemptive multitasking*. If you run **ps -ax**, **ps -ef** or **top** on any Unix system, you will see that there are far more processes running than there are cores in the system.

Sharing time on individual cores works well for programs that are idle much of the time, such as word processors and web browsers that spend most of the time waiting for user input. It can create the illusion that multiple processes are actually running at the same time, since the time it takes to respond to user input may be a small fraction of a second for any one of the processes sharing the CPU.

However, this sort of multitasking does not work well for intensive computational processes that use the CPU constantly for hours, days, or weeks.

There is overhead involved in switching a core between one process and another, so the total time to finish all processes would actually be lower if we ran one process to completion before starting the next. The more frequently a core switches between processes, the more overhead it incurs, and the longer it will take to complete all processes. More frequent context switching sacrifices overall throughput for better response times. Just how much overhead is incurred depends on how much the processes contend with each other for memory and disk resources. The difference could be marginal, or it could be huge.

There is also the possibility that there simply won't be enough memory available for multiple processes to run at the same time.

Hence, most systems that are used to run long, CPU-intensive processes employ a *scheduler* to limit the number of processes using the system at once. Typically, the scheduler will allow at most one process per core to be running at any given moment. Most schedulers also have features to track and balance memory use.

When you run a job under a scheduler, the scheduler locates nodes with sufficient cores, memory and any other resources your job requires. If then marks those resources as "occupied" and dispatches your processes to the nodes containing those resources. When your job completes, the resources are again marked as "available" so that they can be allocated to the next job.

**Caution** Out of courtesy to other users on a shared cluster or grid, you should never run anything that requires significant CPU time, memory, or disk I/O on the head node or any other node that users log into directly. Doing so can overload the computer can cause it to become sluggish or unresponsive for other users. It is generally OK to perform trivial tasks there such as editing programs and scripts, processing small files, etc. Everything else should be scheduled to run on compute nodes.

After submitting a job to the scheduler, you can usually log out of the head/submit node without affecting the job. Hence, you can submit a long-running job, leave, and check the results at a later time.

**Caution** When running jobs in a scheduled environment, *all* jobs must be submitted to the scheduler. The scheduler does not know what resources are being used by processes it did not start, so it will not be able to guarantee performance and stability of the system if anything is run outside the scheduler.

Another advantage of schedulers is their ability to queue jobs. If you submit a job at a time when insufficient resources are available, you can walk away knowing that it will begin executing as soon as the necessary resources become available.

### 7.3.2 Common Schedulers

#### HTCondor

HTCondor is a sophisticated scheduler designed primarily to utilize idle time on grids of existing personal computers across an organization. Many college campuses use HTCondor grids to provide inexpensive parallel computing resources to their users.

#### Load Sharing Facility (LSF)

Load Sharing Facility (LSF) is a proprietary scheduler primarily used on large Linux clusters.

#### Portable Batch System (PBS)

Portable Batch System is an open source scheduler used on clusters of all sizes.

The most popular and modern implementation of the PBS scheduler is Torque. There is also an open source extension for Torque called Maui, which provides enhanced scheduling features, as well as a more sophisticated, commercial version called MOAB.

#### Simple Linux Utility for Resource Management (SLURM)

SLURM is a relatively new open source resource manager and scheduler. It was originally developed on Linux, but naturally runs on other Unix-compatible systems as well. SLURM is distinguished from other schedulers by its high throughput, scalability, modularity, and fault-tolerance.

#### Sun Grid Engine (SGE)

Sun Grid Engine is an open source scheduler originally developed as a proprietary product at Sun Microsystems for their Solaris Unix system. Since it was open sourced, it has become fairly popular on other Unix variants such as BSD and Linux. It is used on clusters of all sizes.

### 7.3.3 Job Types

#### Batch Serial

A *batch serial* job allocates one core and executes a single process on it. All output that would appear on the terminal screen is redirected to a file. You do not need to remain logged into the submit node while a batch job is running, since it does not need access to your terminal.

This is not parallel computing, but clusters and grids may be used this way just to utilize software that users may not have on their own computers. Batch serial jobs are also often used for pre-processing (prep work done before a parallel job) and post-processing (wrap-up work done after a parallel job) that cannot be parallelized.

#### Interactive

An *interactive* job is like a batch serial job, in that a single core is usually used. However, terminal output is not redirected to a file, and the process can receive input from the keyboard. You cannot log out of the submit node while an interactive job is running. Interactive jobs are rarely used, but can be useful for short-running tasks such as program compilation.

#### Batch Parallel (Job Arrays)

A *batch parallel* job runs the same program simultaneously on multiple cores, each using different inputs. The processes are generally independent of each other while running, i.e. they do not communicate with each other. Batch parallel jobs are often referred to as *embarrassingly parallel*, since they are so easy to implement.

#### Multicore

A *multicore* job covers all other types of jobs. Typically, a multicore job is dispatched to a single core like a batch serial job. The scheduler is asked to allocate more than one core, but not to dispatch processes to them. The scheduler only dispatches a *master process* to one core, and it is up to the master process to dispatch and manage processes on the other scheduled cores. Processes within a multicore job usually communicate and cooperate with each other during execution. Hence, this is the most complex type of distributed parallel programming.

Most multicore jobs use the *Message Passing Interface* (MPI) system, which facilitates the creation and communication of cooperative distributed parallel jobs. MPI is discussed in detail in Chapter 34.

### 7.3.4 Checkpointing

The longer a program runs, the more likely it is to experience a problem such as a power outage or hardware failure before it completes.

Checkpointing is the process of periodically saving the partial results and/or current state of a process as it executes. If the exact state of a process can be saved, then it can, in theory, be easily resumed from where it left off after being interrupted. If only partial results can be saved, it may be harder to resume from where you left off, but at least there is a chance that you won't have to start over from the beginning.

If starting over will be a major inconvenience, then checkpointing is always a good idea. How to go about checkpointing depends heavily on the code you're running. Simple batch serial and batch parallel jobs can be checkpointed more easily than shared memory or MPI jobs. Tools exist that allow you to checkpoint simple jobs without any additional coding. More complex jobs may require adding to your code so that it checkpoints itself.

Consult the current scheduler documentation or talk to your facilitator for more information.

### 7.3.5 Self-test

1. What kind of problems would occur if users of a cluster or grid used remote execution directly to run jobs?
2. Why is time-sharing of cores not a good idea in a cluster or grid environment? What types of processes work well with time-sharing?
3. What does a job scheduler do? How does it solve the problem of multiple jobs competing for resources?
4. Describe several popular job schedulers and where they are typically used.
5. Describe the four common categories of jobs that run on clusters and grids.
6. What is checkpointing and what are some of its limitations?

# Chapter 8

# Job Scheduling with SLURM

**Before You Begin**
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 3), the Unix shell (Section 3.3.3, redirection (Section 3.13), shell scripting (Chapter 4) and job scheduling (Chapter 7.

## 8.1 Overview

*SLURM* is an acronym for *Simple Linux Utility for Resource Management*. SLURM manages resources such as CPU cores and memory on a cluster. Originally developed on Linux with the lessons of earlier job schedulers in mind, SLURM is also supported on FreeBSD and NetBSD, and has been used on Mac OS X. SLURM is also rich in advanced features, so the ’S’ and the ’L’ in SLURM could safely be removed at this point.

For complete information, see the official SLURM documentation at http://slurm.schedmd.com/.

This chapter covers the basics of SLURM along with some extensions provided by SPCM, *Simple, Portable Cluster Manager*. SPCM is a set of tools for building and managing SLURM clusters. SPCM was developed on CentOS Linux and FreeBSD, but is designed to be easily adapted to any POSIX platform. Some convenient tools provided by SPCM are mentioned below.

## 8.2 Using SLURM

### 8.2.1 SLURM Jargon

Before we begin, you should know a few common terms that will pop up throughout this chapter.

A *CPU* or *core* is a processor capable of running a program.

A *node* is a distinct computer within a cluster. See Section 6.3.2 for a description of node types.

A *partition* is a group of nodes designated in the SLURM configuration as a pool from which to allocate resources.

A *process* is a process as defined by Unix. (The execution of a program.) A parallel job on a cluster consists of multiple processes running at the same time. A serial job consists of only one process.

A *job* is all of the processes dispatched by SLURM under the same job ID. Processes within a job on a cluster may be running on the same node or different nodes.

A *task* is one or more processes within a job that are distinct from the rest of the processes. All processes within a given task must run on the same node. For example, a job could consist of 4 tasks running on different nodes, each consisting of 8 threads using shared-memory parallelism. This job therefore uses 32 cores and a maximum of 4 different nodes. Two tasks could run on the same node if the node has at least 16 cores.

### 8.2.2 Cluster Status

Before scheduling any jobs through SLURM, it is often useful to check the status of the nodes. Knowing how many cores are available may influence your decision on how many cores to request for your next job.

For example, if only 50 cores are available at the moment, and your job requires 200 cores, the job may have to wait in the queue until 200 cores (with sufficient associated memory) are free. Your job may finish sooner if you reduce the number of cores to 50 or less so that it can start right away, even though it will take longer to run. We can't always predict when CPU cores will become available, so it's often a guessing game. However, some experienced users who know their software may have a pretty good idea when their jobs will finish.

The **sinfo** command shows information about the nodes and partitions in the cluster. This can be used to determine the total number of cores in the cluster, cores in use, etc.

```
shell-prompt: sinfo
PARTITION         AVAIL  TIMELIMIT  NODES  STATE NODELIST
default-partitio*    up   infinite      2   idle compute-[001-002]

shell-prompt: sinfo --long
Wed Nov 27 13:34:20 2013
PARTITION         AVAIL  TIMELIMIT   JOB_SIZE ROOT SHARE     GROUPS  NODES       STATE  ←↩
    NODELIST
default-partitio*    up   infinite 1-infinite   no    NO        all      2        idle  ←↩
    compute-[001-002]

shell-prompt: sinfo -N
NODELIST           NODES         PARTITION STATE
compute-[001-002]      2 default-partitio* idle
```

As a convenience, SPCM clusters provide a script called **slurm-node-info**, which displays specifications on each compute node:

```
shell-prompt: slurm-node-info
HOSTNAMES CPUS MEMORY
compute-001 12 31751
compute-002 12 31751
compute-003 12 31751
compute-004 12 31751
compute-005 12 31751
compute-006 12 31751
compute-007 12 31751
```

### 8.2.3 Job Status

You can check on the status of running jobs using **squeue**.

```
shell-prompt: squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
                64 default-p bench.sl    bacon  R       0:02      2 compute-[001-002]
```

The Job id column shows the numeric job ID. The ST column shows the current status of the job. The most common status flags are 'PD' for pending (waiting to start) and 'R' for running.

The **squeue** `--long` flag requests more detailed information.

The **squeue** command has many flags for controlling what it reports. Run **man squeue** for full details.

As a convenience, SPCM clusters provide a script called **slurm-cluster-load**, which uses squeue and sinfo to display a quick summary on current jobs and overall load:

```
shell-prompt: slurm-cluster-load
JOBID USER       EXEC_HOST    CPUS NODES SHARED NAME       TIME       ST
186110 chen59   R  16   2   no  surf     1-08:46:19  compute-2-[14,16]
```

```
186112 chen59   R  16   2   no  surf     1-08:46:19  compute-2-33,compute-3-02
186113 chen59   R  16   2   no  surf     1-08:46:19  compute-3-[07,10]
187146 albertof R  1    1   yes bash     1-03:08:12  compute-2-04
187639 albertof R  1    1   yes submitSR 52:03       compute-4-02
187640 albertof R  1    1   yes submitSR 52:03       compute-4-02
187642 albertof R  1    1   yes submitSR 52:03       compute-4-02
187867 qium     R  8    1   no  ph/IC1x1 46:11       compute-1-03
187868 qium     R  8    1   no  ph/IC1x2 46:11       compute-1-04

CPUS(A/I/O/T)
785/264/87/1136

Load: 69%
```

The **slurm-user-cores** command displays only the number of cores currently in use by each user and the cluster usage summary:

```
shell-prompt: slurm-user-cores
Username    Running Pending
bahumda2    15      0
bkhazaei    300     0
sheikhz2    640     0
yous        1       0

batch: 88%
CPUS(A/I/O/T)
955/109/16/1080

128g: 6%
CPUS(A/I/O/T)
1/15/0/16
```

### 8.2.4 Using top

Using the output from **squeue**, we can see which compute nodes are being used by a job:

```
shell-prompt: squeue
 JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
  1017     batch     bash   oleary  R    2:44:00      1 compute-001
  1031     batch ARM97-pa  roberts  R    1:50:55      2 compute-[001-002]
  1032     batch   sbatch     joea  R    1:35:19      1 compute-002
  1034     batch   sbatch     joea  R    1:05:59      1 compute-002
  1035     batch   sbatch     joea  R      50:08      1 compute-003
  1036     batch   sbatch     joea  R      47:13      1 compute-004
  1041     batch     bash   oleary  R      36:41      1 compute-001
  1042     batch     bash  roberts  R       0:09      1 compute-001
```

From the above output, we can see that job 1031 is running on compute-001 through compute-002.

We can then examine the processes on any of those nodes using a remotely executed **top** command:

```
shell-prompt: ssh -t compute-001 top

top - 13:55:55 up 12 days, 24 min,  1 user,  load average: 5.00, 5.00, 4.96
Tasks: 248 total,   6 running, 242 sleeping,   0 stopped,   0 zombie
Cpu(s): 62.5%us,  0.1%sy,  0.0%ni, 37.3%id,  0.1%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  24594804k total, 15388204k used,  9206600k free,   104540k buffers
Swap: 33554424k total,    10640k used, 33543784k free, 14486568k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
30144 roberts   20   0  251m  69m  17m R 100.4  0.3 113:29.04 SAM_ADV_MPDATA_
30145 roberts   20   0  251m  69m  17m R 100.4  0.3 113:28.16 SAM_ADV_MPDATA_
```

```
30143 roberts   20   0  251m  69m  17m R 100.1  0.3 113:16.87 SAM_ADV_MPDATA_
30146 roberts   20   0  251m  69m  17m R 100.1  0.3 113:29.24 SAM_ADV_MPDATA_
30147 roberts   20   0  251m  69m  17m R 100.1  0.3 113:29.82 SAM_ADV_MPDATA_
  965 bacon     20   0 15168 1352  944 R  0.3  0.0   0:00.01 top
    1 root      20   0 19356 1220 1004 S  0.0  0.0   0:00.96 init
```

**Note** The `-t` flag is important here, since it tells ssh to open a connection with full terminal control, which is needed by **top** to update your terminal screen.

The column of interest is under "RES".

We can see from the top command above that the processes owned by job 1031 are using about 69 mebibytes of memory each. Watch this value for a while as it will change as the job runs.

Take the highest value you see in the RES column, add about 10%, and use this with --mem-per-cpu to set a reasonable memory limit.

```
#SBATCH --mem-per-cpu=76
```

SPCM clusters provide some additional convenience tools to save a little typing when running top.

The **node-top** command is equivalent to running top on a compute node via ssh as shown above, without typing the ssh command or the full host name:

```
shell-prompt: node-top 001
```

The **job-top** command takes a SLURM job ID and runs top on *all* of the node used by the job. The **job-top** command below is equivalent to running **node-top 001** followed by **node-top 001**:

```
shell-prompt: squeue
 JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
  1017     batch     bash   oleary  R    2:44:00      1 compute-001
  1031     batch ARM97-pa  roberts  R    1:50:55      2 compute-[001-002]
  1032     batch   sbatch     joea  R    1:35:19      1 compute-002
  1034     batch   sbatch     joea  R    1:05:59      1 compute-002
  1035     batch   sbatch     joea  R      50:08      1 compute-003
  1036     batch   sbatch     joea  R      47:13      1 compute-004
  1041     batch     bash   oleary  R      36:41      1 compute-001
  1042     batch     bash  roberts  R       0:09      1 compute-001
shell-prompt: job-top 1031
```

### 8.2.5 Job Submission

The purpose of this section is to provide the reader a quick start in job scheduling using the most common tools. The full details of job submission are beyond the scope of this document. For more information, see SLURM website http://slurm.schedmd.com/ and the man pages for individual SLURM commands.

```
man sbatch
```

#### Submission Scripts

Submitting jobs involves specifying a number of job parameters such as the number of cores, the job name (which is used by other SLURM commands), the name(s) of the output file(s), etc.

In order to document all of this information and make it easy to resubmit the same job, this information is usually incorporated into a *submission script*. Using a script saves you a lot of typing when you want to run-submit the same job, and also fully documents the job parameters.

A submission script is an ordinary shell script, with some *directives* inserted to provide information to the scheduler. For SLURM, the directives are specially formatted shell comments beginning with "#SBATCH".

**Note**
The ONLY difference between an sbatch submission script and a script that you would run on any Unix laptop or workstation is the #SBATCH directives.
You can develop and test a script to run your analyses or models on any Unix system. To use it on a SLURM cluster, you need only add the appropriate #SBATCH directives and possibly alter some command arguments to enable parallel execution.

**Caution** There cannot be any Unix commands above #SBATCH directives. SLURM will ignore any #SBATCH directives below the first Unix command.

Submission scripts are submitted with the sbatch command. The script may contain #SBATCH options to define the job, regular Unix commands, and srun or other commands to run programs in parallel.

**Note** The script submitted by sbatch is executed on one core, regardless of how many cores are allocated for the job. The commands within the submission script are responsible for dispatching multiple processes for parallel jobs. This strategy differs from other popular schedulers like Torque (PBS) and LSF, where the submission script is run in parallel on all cores.

Suppose we have the following text in a file called `hostname.sbatch`:

```
#!/usr/bin/env bash

# A SLURM directive
#SBATCH --job-name=hostname

# A command to be executed on the scheduled node.
# Prints the host name of the node running this script.
hostname
```

The script is submitted to the SLURM scheduler as a command line argument to the **sbatch** command:

```
                shell-prompt: sbatch hostname.sbatch
```

The SLURM scheduler finds a free core on a compute node, reserves it, and then remotely runs hostname.sbatch on the compute node using **ssh** or some other remote execution command.

Recall from Chapter 4 that everything in a shell script from a '#' character to the end of the line is considered a comment by the shell, and ignored.

However, comments beginning with #SBATCH, while ignored by the shell, are interpreted as directives by **sbatch**.

The directives within the script provide command line flags to **sbatch**. For instance, the line

```
                #SBATCH --mem-per-cpu=10
```

causes **sbatch** to behave as if you had typed

```
                shell-prompt: sbatch --mem-per-cpu=10 hostname.sbatch
```

By putting these comments in the script, you eliminate the need to remember them and retype them every time you run the job. It's generally best to put all **sbatch** flags in the script rather than type any of them on the command line, so that you have an exact record of how the job was started. This will help you determine what went wrong if there are problems, and allow you to reproduce the results at a later date.

**Note** If you want to disable a #SBATCH comment, you can just add another '#' rather than delete it. This will allow you to easily enable it again later as well as maintain a record of options you used previously.

```
                ##SBATCH This line is ignored by sbatch
                #SBATCH This line is interpreted by sbatch
```

**Practice Break**

```
#!/usr/bin/env bash

# A SLURM directive
#SBATCH --job-name=hostname

# A command to be executed on the scheduled node.
# Prints the host name of the node running this script.
hostname
```

Type in the `hostname.sbatch` script shown above and submit it to the scheduler using **sbatch**. Then check the status with **squeue** and view the output and error files.

### Common Flags

```
#SBATCH --output=standard-output-file
#SBATCH --error=standard-error-file
#SBATCH --nodes=min[-max]
#SBATCH --ntasks=tasks
#SBATCH --array=list
#SBATCH --cpus-per-task=N
#SBATCH --exclusive
#SBATCH --mem=MB
#SBATCH --mem-per-cpu=MB
#SBATCH --partition=name
```

The `--output` and `--error` flags control the names of the files to which the standard output and standard error of the processes are redirected. If omitted, both standard output and standard error are written to slurm-JOBID.out.

**Note** Commands in a submission script should not use output redirection (>) or error redirection (2>, >&), since these would conflict with --output and --error.

The `--ntasks` flag indicates how many tasks the job will run, usually for multicore jobs.

The `--nodes` flag is used to specify how many nodes (not tasks) to allocate for the job. We can use this to control how many tasks are run on each node. For example, if a job consists of 20 I/O-intensive processes, we would not want many of them running on the same node and competing for the local disk. In this case, we can specify --nodes=20 --ntasks=20 to force each process to a separate node.

**Note** The sbatch command with --nodes or --ntasks will not cause multiple processes to run. The sbatch command simply runs the script on one node. To run multiple tasks, you must use srun, mpirun, or some other dispatcher within the script. This differs from many other schedulers such as LSF and Torque.

The --cpus-per-task=N flag indicates that we need N cpus on the same node. Using --ntasks=4 --cpus-per-task=3 will indicate 4 tasks using 3 cores each, in effect requesting 12 cores in groups of 3 per node.

The --exclusive flag indicates that the job can not share nodes with other jobs. This is typically used for shared memory parallel jobs to maximize the number of cores available to the job. It may also be used for jobs with high memory requirements, although it is better to simple specify the memory requirements using --mem or --mem-per-cpu.

The --mem=MB flag indicates the amount of memory needed on each node used by the job, in megabytes.

The --mem-per-cpu=MB flag indicates the amount of memory needed by each process within a job, in megabytes.

The --partition=name flag indicates which partition (set of nodes) on which the job should run. Simply run **sinfo** to see a list of available partitions.

### SLURM Resource Requirements

When using a cluster, it is important to develop a feel for the resources required by your jobs, and inform the scheduler as accurately as possible what will be needed in terms of CPU time, memory, etc.

If a user does not specify a given resource requirement, the scheduler uses default limits. Default limits are set low, so that users are encouraged to provide an estimate of required resources for all non-trivial jobs. This protects other users from being blocked by long-running jobs that require less memory and other resources than the scheduler would assume.

The --mem=MB flag indicates that the job requires MB megabytes of memory per node.

The --mem-per-cpu=MB flag indicates that the job requires MB megabytes per core.

**Note** It is a very important to specify memory requirements accurately in all jobs, and it is generally easy to predict based on previous runs by monitoring processes within the job using **top** or **ps**. Failure to do so could block other jobs run running, even though the resources it requires are actually available.

### Batch Serial Jobs

A batch serial submission script need only have optional flags such as job name, output file, etc. and one or more commands.

```
#!/usr/bin/env bash

hostname
```

### Interactive Jobs

Interactive jobs involve running programs under the scheduler on compute nodes, where the user can interact directly with the process. I.e., output is displayed on their terminal instead of being redirected to files (as controlled by the sbatch --output and --error flags) and input can be taken from the keyboard.

The use of interactive jobs on a cluster is uncommon, but sometimes useful.

For example, an interactive shell environment might be useful when porting a new program to the cluster. This can be used to do basic compilation and testing in the compute node environment before submitting larger jobs. Programs should not be compiled or tested on the head node and users should not ssh directly to a compute node for this purpose, as this would circumvent the scheduler, causing their compilations, etc. to overload the node.

**Caution** A cluster is *not* a good place to do code development and testing. Code should be fully developed and tested on a separate development systems such as a laptop or workstation before being run on a cluster, where buggy programs will waste valuable shared resources and may cause other problems for other users.

On SPCM clusters, a convenience script called **slurm-shell** is provided to make it easy to start an interactive shell under the scheduler. It requires the number of cores and the amount of memory to allocate as arguments. The memory specification is in mebibytes by default, but can be followed by a 'g' to indicate gibibytes:

```
shell-prompt: slurm-shell 1 500
==========================================================================
Note:

slurm-shell is provided solely as a convenience for typical users.

If you would like an interactive shell with additional memory, more
than 1 CPU, etc., you can use salloc and srun directly.  Run

    more /usr/local/bin/slurm-shell

for a basic example and

    man salloc
    man srun

for full details.
==========================================================================

 1:05PM  up 22 days, 2 hrs, 0 users, load averages: 0.00, 0.00, 0.00
To repeat the last command in the C shell, type "!!".
        -- Dru <genesis@istar.ca>
FreeBSD compute-001. bacon ~ 401:
```

### Batch Parallel Jobs (Job Arrays)

A job array is a set of independent processes all started by a single job submission. The entire job array can be treated as a single job by SLURM commands such as **squeue** and **scancel**.

A batch parallel submission script looks almost exactly like a batch serial script. There are a couple of ways to run a batch parallel job. It is possible to run a job array using --ntasks and an srun command in the sbatch script. However, the --array flag is far more convenient for most purposes.

The --array flag is followed by an index specification which allows the user to specify any set of task IDs for each job in the array. The specification can use '-' to specify a range of task IDs, commas to specify an arbitrary list, or both. For example, to run a job array with task IDs ranging from 1 to 5, we would use the following:

**Note**
Some SLURM clusters use an operating system feature known as task affinity to improve performance. The feature binds each process to a specific core, so that any data in the cache RAM of that core does not have to be flushed and reloaded into the cache of a different core. This can dramatically improve performance for certain memory-intensive processes, though it won't make any noticeable difference for many jobs.
In order to activate task affinity, you must use **srun** or another command to launch the processes.

```
#!/usr/bin/env bash

#SBATCH  --array=1-5

# OK, but will not use task affinity
hostname
```

```
#!/usr/bin/env bash

#SBATCH  --array=1-5

# Activates task affinity, so each hostname process will stick to one core
srun hostname
```

If we wanted task IDs of 1,2,3,10, 11, and 12, we could use the following:

```
#!/usr/bin/env bash

#SBATCH  --array=1-3,10-12

srun hostname
```

The reasons for selecting specific task IDs are discussed in Section 8.2.5.

**Note** Each task in an array job is treated as a separate job by slurm. Hence, --mem and --mem-per-cpu mean the same thing in an array job. Both refer to the memory required by one task.

Another advantage of using --array is that we don't need all the cores available at once in order for the job to run. For example, if we submit a job using --array=1-200 while there are only 50 cores free on the cluster, it will begin running as many processes as possible immediately and run more as more cores become free. In contrast, a job started with --ntasks will remain in a pending state until there are 200 cores free.

Furthermore, we can explicitly limit the number of array jobs run at once with a simple addition to the flag. Suppose we want to run 10,000 processes, but be nice to other cluster users and only use 100 cores at a time. All we need to do is use the following:

```
#SBATCH --array=1-10000%100
```

**Caution** Do not confuse this with --array=1-1000:4, which simply increments the job index by 4 instead of one, and will attempt to run 10,000 processes at once using indices 1, 5, 9, ...!

Finally, we can run arrays of multithreaded jobs by adding `--cpus-per-task`. For example, the BWA sequence aligner allows the user to specify multiple threads using the `-t` flag:

```
#SBATCH --array=1-100 --nodes=1 --cpus-per-task=8

srun bwa mem -t $SLURM_CPUS_PER_TASK reference.fa input.fa > output.sam
```

**Practice Break**
Copy your `hostname.sbatch` to `hostname-parallel.sbatch`, modify it to run 5 processes, and submit it to the scheduler using **sbatch**. Then check the status with **squeue** and view the output and error files.

### Multi-core Jobs

A multi-core job is any job that runs a parallel program. This means that it uses multiple processes that communicate and cooperate with each other in some way. There is no such communication or cooperation in batch parallel (also known as embarrassingly parallel) jobs, which use SLURM's --array flag.

Cores are allocated for multi-core jobs using the --ntasks flag.

Using --ntasks ONLY tells the SLURM scheduler how many cores to reserve. It does NOT tell your parallel program how many cores or which compute nodes to use. It is the responsibility of the user to ensure that the command(s) in their script utilize the correct number of cores and the correct nodes within the cluster.

Some systems, such as OpenMPI (discussed in Section 34.7), will automatically detect this information from the SLURM environment and dispatch the correct number of processes to each allocated node.

If you are not using OpenMPI, you may need to specify the number of cores and/or which nodes to use in the command(s) that run your computation. Many commands have a flag argument such as -n or -np for this purpose.

OpenMP (not to be confused with OpenMPI) is often used to run multiple threads on the same node (shared memory parallelism, as discussed in Section 6.2 and Section 34.5).

OpenMP software will look for the OMP_NUM_THREADS environment variable to indicate how many cores to use. It is the user's responsibility to ensure that OMP_NUM_THREADS is set when using OpenMP software.

When running OpenMP programs under SLURM, we can ensure that the right number of cores are used by using the `--ntasks` flag and passing `$SLURM_NTASKS` to `$OMP_NUM_THREADS`.

```
#!/bin/sh -e

#SBATCH --ntasks=4

OMP_NUM_THREADS=$SLURM_NTASKS
export OMP_NUM_THREADS

OpenMP-based-program arg1 arg2 ...
```

#### MPI Multi-core Jobs

Scheduling MPI jobs in SLURM is much like scheduling batch parallel jobs.

MPI programs cannot be executed directly from the command line as we do with normal programs and scripts. Instead, we must use the **mpirun** or **srun** command to start up MPI programs.

**Note** Whether you should use mpirun or srun depends on how your SLURM scheduler is configured. See slurm.conf or talk to your systems manager to find out more.

```
mpirun [mpirun flags] mpi-program [mpi-program arguments]
```

**Caution** Like any other command used on a cluster or grid, **mpirun** must not be executed directly from the command line, but instead must be used in a scheduler submission script.

For MPI and other multicore jobs, we use the sbatch --ntasks flag to indicate how many cores we want to use. Unlike --array, sbatch with --ntasks runs the sbatch script on only one node, and it is up to the commands in the script (such as mpirun or srun) to dispatch the parallel processes to all the allocated cores.

Commands like mpirun and srun can retrieve information about which cores have been allocated from the environment handed down by their parent process, the SLURM scheduler. The details about SLURM environment variables are discussed in Section 8.2.5.

```
#!/bin/sh

#SBATCH --ntasks=2

PATH=${PATH}:/usr/local/mpi/openmpi/bin
export PATH

mpirun ./mpi_bench
```

When running MPI jobs, it is often desirable to have as many processes as possible running on the same node. Message passing is generally faster between processes on the same node than between processes on different nodes, because messages passed within the same node need not cross the network. If you have a very fast network such as Infiniband or a low-latency Ethernet, the difference may be marginal, but on more ordinary networks such as gigabit Ethernet, the difference can be enormous.

SLURM by default will place as many processes as possible on the same node. We can also use --exclusive to ensure that "leftover" cores on a partially busy node will not be used for out MPI jobs. This may improve communication performance, but may also delay the start of the job until enough nodes are completely empty.

### Environment Variables

SLURM sets a number of environment variables when a job is started. These variables can be used in the submission script and within other scripts or programs executed as part of the job.

```
shell-prompt: srun printenv | grep SLURM
SLURM_SRUN_COMM_PORT=27253
SLURM_TASKS_PER_NODE=1
SLURM_NODELIST=compute-001
SLURM_JOB_NUM_NODES=1
SLURM_NNODES=1
SLURM_STEPID=0
SLURM_STEP_ID=0
SLURM_JOBID=81
SLURM_JOB_ID=81
SLURM_DISTRIBUTION=cyclic
SLURM_NPROCS=1
SLURM_NTASKS=1
SLURM_JOB_CPUS_PER_NODE=1
SLURM_JOB_NAME=/usr/bin/printenv
SLURM_SUBMIT_HOST=finch.cs.uwm.edu
SLURM_SUBMIT_DIR=/usr/home/bacon
SLURM_PRIO_PROCESS=0
SLURM_STEP_NODELIST=compute-001
SLURM_STEP_NUM_NODES=1
SLURM_STEP_NUM_TASKS=1
SLURM_STEP_TASKS_PER_NODE=1
SLURM_STEP_LAUNCHER_PORT=27253
SLURM_SRUN_COMM_HOST=192.168.0.2
SLURM_TOPOLOGY_ADDR=compute-001
SLURM_TOPOLOGY_ADDR_PATTERN=node
SLURM_CPUS_ON_NODE=1
SLURM_TASK_PID=5945
SLURM_NODEID=0
SLURM_PROCID=0
SLURM_LOCALID=0
SLURM_LAUNCH_NODE_IPADDR=192.168.0.2
SLURM_GTIDS=0
SLURM_CHECKPOINT_IMAGE_DIR=/usr/home/bacon
SLURMD_NODENAME=compute-001
```

The `SLURM_JOB_NAME` variable can be useful for generating output file names within a program, among other things.

When submitting a job array using --array, the submission script is dispatched to every core by sbatch. To distinguish between tasks in this type of job array, we would examine SLURM_ARRAY_TASK_ID. This variable will be set to a different value for each task, from the specification given with --array.

For example, suppose we have 100 input files named input-1.txt through input-100.txt. We could use the following script to process them:

```
                #!/usr/bin/env bash

                #SBATCH --array=1-100
                ./myprog < input-$SLURM_ARRAY_TASK_ID.txt \
                    > output-$SLURM_ARRAY_TASK_ID.txt
```

Suppose our input files are not numbered sequentially, but according to some other criteria, such as a set of prime numbers. In this case, we would simply change the specification in --array:

```
                #!/usr/bin/env bash

                #SBATCH --array=2,3,5,7,11,13

                ./myprog < input-$SLURM_ARRAY_TASK_ID.txt \
                    > output-$SLURM_ARRAY_TASK_ID.txt
```

### 8.2.6 Terminating a Job

If you determine that a job is not behaving properly (by reviewing partial output, for example), you can terminate it using **scancel**, which takes a job ID as a command line argument.

```
shell-prompt: sbatch bench.sbatch
Submitted batch job 90
shell-prompt: squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
                90 default-p bench.sl    bacon  R       0:03      2 compute-[001-002]
shell-prompt: scancel 90
shell-prompt: squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

### 8.2.7 Terminating Stray Processes

Occasionally, a SLURM job may fail and leave processes running on the compute nodes. These are called *stray processes*, since they are no longer under the control of the scheduler.

Do not confuse stray processes with *zombie processes*. A zombie process is a process that has terminated but has not been reaped. I.e., it is still listed by the **ps** command because it's parent process has not become aware that it is terminated. Zombie processes are finished and do not consume any resources, so we need not worry about them.

Stray processes are easy to detect on nodes where you have no jobs running. If **squeue** or **slurm-cluster-load** do not show any of your jobs using a particular node, but **top** or **node-top** show processes under your name, then you have some strays. Simply ssh to that compute node and terminate them with the standard Unix **kill** or **killall** commands.

If you have jobs running on the same node as your strays, then detecting the strays may be difficult. If the strays are running a different program than your legitimate job processes, then they will be easy to spot. If they are running the same program as your legitimate job processes, then they will likely have a different run time than your legitimate processes. Be very careful to ensure that you kill the strays and not your active job in this situation.

```
Linux login.mortimer bacon ~ 414: squeue -u bacon
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)

Linux login.mortimer bacon ~ 415: node-top 003
top - 09:33:55 up 2 days, 10:44,  1 user,  load average: 16.02, 15.99, 14.58
Tasks: 510 total,  17 running, 493 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.3%us,  0.0%sy,  0.0%ni, 97.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  49508900k total,  2770060k used, 46738840k free,    62868k buffers
Swap: 33554428k total,        0k used, 33554428k free,   278916k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 7530 bacon     20   0  468m 112m 4376 R 81.5  0.2  37:55.11 mpi-bench
 7531 bacon     20   0  468m 112m 4524 R 81.5  0.2  37:55.44 mpi-bench
 7532 bacon     20   0  468m 112m 4552 R 81.5  0.2  37:55.27 mpi-bench
 7533 bacon     20   0  468m 112m 4552 R 81.5  0.2  37:55.83 mpi-bench
 7537 bacon     20   0  468m 112m 4556 R 81.5  0.2  37:55.71 mpi-bench
 7548 bacon     20   0  468m 112m 4556 R 81.5  0.2  37:55.51 mpi-bench
 7550 bacon     20   0  468m 112m 4552 R 81.5  0.2  37:55.87 mpi-bench
 7552 bacon     20   0  468m 112m 4532 R 81.5  0.2  37:55.82 mpi-bench
 7554 bacon     20   0  468m 112m 4400 R 81.5  0.2  37:55.83 mpi-bench
 7534 bacon     20   0  468m 112m 4548 R 79.9  0.2  37:55.78 mpi-bench
 7535 bacon     20   0  468m 112m 4552 R 79.9  0.2  37:55.84 mpi-bench
 7536 bacon     20   0  468m 112m 4552 R 79.9  0.2  37:55.87 mpi-bench
 7542 bacon     20   0  468m 112m 4552 R 79.9  0.2  37:55.86 mpi-bench
 7544 bacon     20   0  468m 112m 4552 R 79.9  0.2  37:55.84 mpi-bench
 7546 bacon     20   0  468m 112m 4544 R 79.9  0.2  37:55.88 mpi-bench
 7529 bacon     20   0  468m 110m 4408 R 78.3  0.2  37:41.46 mpi-bench
    4 root      20   0     0    0    0 S  1.6  0.0   0:00.04 ksoftirqd/0
 7527 bacon     20   0  146m 4068 2428 S  1.6  0.0   0:11.39 mpirun
 7809 bacon     20   0 15280 1536  892 R  1.6  0.0   0:00.07 top
    1 root      20   0 19356 1536 1240 S  0.0  0.0   0:02.03 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.03 kthreadd
Connection to compute-003 closed.

Linux login.mortimer bacon ~ 416: ssh compute-003 kill 7530

Linux login.mortimer bacon ~ 417: ssh compute-003 killall mpi-bench
```

If the normal kill command doesn't work, you can use **kill -9** to do a sure kill.

### 8.2.8 Viewing Output of Active Jobs

Unlike many other schedulers, SLURM output files are available for viewing while the job is running. Hence, SLURM does not require a "peek" command. The default output file or files specified by --output and --error are updated regularly while a job is running and can be viewed with standard Unix commands such as "more".

### 8.2.9 Checking Job Stats with sacct

The sacct command is used to view accounting statistics on completed jobs. With no command-line arguments, sacct prints a summary of your past jobs:

```
shell-prompt: sacct
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
70            build.slu+ default-p+     (null)          1  COMPLETED      0:0
71            build.slu+ default-p+     (null)          1  COMPLETED      0:0
72            bench.slu+ default-p+     (null)         24  COMPLETED      0:0
...
184             hostname default-p+     (null)         40  COMPLETED      0:0
184.0           hostname                (null)         40  COMPLETED      0:0
185           bench-fre+ default-p+     (null)         48  CANCELLED      0:0
185.0              orted                (null)          3     FAILED      1:0
186           env.sbatch default-p+     (null)          1  COMPLETED      0:0
```

For detailed information on a particular job, we can use the -j flag to specify a job id and the -o flag to specify which information to display:

```
shell-prompt: sacct -o alloccpus,nodelist,elapsed,cputime -j 117
 AllocCPUS        NodeList    Elapsed    CPUTime
---------- --------------- ---------- ----------
        12     compute-001   00:00:29   00:05:48
```

On SPCM clusters, the above command is provided in a convenience script called slurm-job-stats:

```
shell-prompt: slurm-job-stats 117
 AllocCPUS        NodeList    Elapsed    CPUTime
---------- --------------- ---------- ----------
        12     compute-001   00:00:29   00:05:48
```

For more information on sacct, run "man sacct" or view the online SLURM documentation.

### 8.2.10 Checking Status of Running Jobs with scontrol

Detailed information on currently running jobs can be displayed using scontrol.

```
shell-prompt: scontrol show jobid 10209
JobId=10209 JobName=bench-freebsd.sbatch
   UserId=bacon(4000) GroupId=bacon(4000)
   Priority=4294901512 Nice=0 Account=(null) QOS=(null)
   JobState=FAILED Reason=NonZeroExitCode Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=213:0
   RunTime=00:00:02 TimeLimit=UNLIMITED TimeMin=N/A
   SubmitTime=2015-09-09T11:25:31 EligibleTime=2015-09-09T11:25:31
   StartTime=2015-09-09T11:25:32 EndTime=2015-09-09T11:25:34
   PreemptTime=None SuspendTime=None SecsPreSuspend=0
   Partition=default-partition AllocNode:Sid=login:7345
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=compute-[003-006]
   BatchHost=compute-003
   NumNodes=4 NumCPUs=48 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
   MinCPUsNode=1 MinMemoryCPU=1024M MinTmpDiskNode=0
   Features=(null) Gres=(null) Reservation=(null)
   Shared=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/share1/Data/bacon/Facil/Software/Src/Bench/MPI/bench-freebsd.sbatch
      WorkDir=/share1/Data/bacon/Facil/Software/Src/Bench/MPI
   StdErr=/share1/Data/bacon/Facil/Software/Src/Bench/MPI/slurm-10209.out
   StdIn=/dev/null
   StdOut=/share1/Data/bacon/Facil/Software/Src/Bench/MPI/slurm-10209.out
```

On SPCM clusters, the above command is provided in a convenience script called slurm-job-status:

```
shell-prompt: squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
             20537     batch bench.sb    bacon  R       0:02      1 compute-001

shell-prompt: slurm-job-status 20537
JobId=20537 JobName=bench.sbatch
   UserId=bacon(4000) GroupId=bacon(4000)
   Priority=4294900736 Nice=0 Account=(null) QOS=(null)
   JobState=RUNNING Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
   RunTime=00:00:12 TimeLimit=UNLIMITED TimeMin=N/A
   SubmitTime=2016-07-18T11:38:31 EligibleTime=2016-07-18T11:38:31
   StartTime=2016-07-18T11:38:31 EndTime=Unknown
   PreemptTime=None SuspendTime=None SecsPreSuspend=0
   Partition=batch AllocNode:Sid=login:7485
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=compute-001
   BatchHost=compute-001
   NumNodes=1 NumCPUs=12 CPUs/Task=1 ReqB:S:C:T=0:0:*:*
   TRES=cpu=12,mem=3072,node=1
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
   MinCPUsNode=1 MinMemoryCPU=256M MinTmpDiskNode=0
   Features=(null) Gres=(null) Reservation=(null)
   Shared=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/share1/Data/bacon/Testing/mpi-bench/trunk/bench.sbatch freebsd
   WorkDir=/share1/Data/bacon/Testing/mpi-bench/trunk
   StdErr=/share1/Data/bacon/Testing/mpi-bench/trunk/slurm-20537.out
   StdIn=/dev/null
   StdOut=/share1/Data/bacon/Testing/mpi-bench/trunk/slurm-20537.out
   Power= SICP=0
```

### 8.2.11 Job Sequences

If you need to submit a series of jobs in sequence, where one job begins after another has completed, the simplest approach is to simply submit job N+1 from the sbatch script for job N.

It's important to make sure that the current job completed successfully before submitting the next, to avoid wasting resources. It is up to you to determine the best way to verify that a job was successful. Examples might include grepping the log file for some string indicating success, or making the job create a marker file using the **touch** command after a successful run. If the command used in your job returns a Unix-style exit status (0 for success, non-zero on error), then you can simply use the shell's exit-on-error feature to make your script exit when any command fails. Below is a template for scripts that might run a series of jobs.

```
#!/bin/sh

#SBATCH job-parameters
set -e  # Set exit-on-error

job-command

# This script will exit here if job-command failed

sbatch job2.sbatch  # Executed only if job-command succeeded
```

### 8.2.12 Self-test

1. What is the SLURM command for showing the current state of all nodes in a cluster?

2. What is the SLURM command to show the currently running jobs on a cluster?

3. Write and submit a batch-serial SLURM script called `list-etc.sbatch` that prints the host name of the compute node on which it runs and a long-listing of `/etc` directory on that node.

   The script should store the output of the commands in `list-etc.stdout` and error messages in `list-etc.stderr` in the directory from which the script was submitted.

   The job should appear in **squeue** listings under the name "list-etc".

   Quickly check the status of your job after submitting it.

4. Copy your `list-etc.sbatch` script to `list-etc-parallel.sbatch`, and modify it so that it runs the **hostname** and **ls** commands on 10 cores instead of just one.

   The job should produce a separate output file for each process named `list-etc-parallel.o<jobid>-<arrayid>` and a separate error file for each process named `list-etc-parallel.e<jobid>-<arrayid>`.

   Quickly check the status of your job after submitting it.

5. What is the SLURM command for terminating a job with job-id 3545?

6. What is the SLURM command for viewing the terminal output of a job with job-id 3545 while it is still running?

7. What is the SLURM command for showing detailed job information about the job with job-id 3254?

## 8.3 Local Customizations

TBD

# Chapter 9

# Job Scheduling with HTCondor

This chapter is based on content developed by Dr. Lars Olson, Marquette University.

**Before You Begin**
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 3), the Unix shell (Section 3.3.3, redirection (Section 3.13), shell scripting (Chapter 4), and job scheduling (Chapter 7.

For complete information, see the official HTCondor documentation at http://research.cs.wisc.edu/htcondor/.

## 9.1 The HTCondor Resource Manager

The goal of this section is to provide a quick introduction to the HTCondor scheduler. For more detailed information, consult the HTCondor man pages and the HTCondor website.

### 9.1.1 A Unique Approach to Resource Management

HTCondor is very different from other common resource managers because it serves different goals.

HTCondor is intended for use on "distributively owned" resources, i.e. PCs that are owned by people other than the HTCondor users and used primarily for purposes other than parallel computing. Most other resource managers are designed for use on dedicated resources, such as a cluster of computers built specifically for parallel computation.

As such, HTCondor is designed to "borrow" resources from their owners while the owners are not otherwise using them. The PC owner has the right to "evict" (terminate) an HTCondor process at any time and without warning. In fact, if an owner logs into a PC while an HTCondor process is running, the process is generally evicted and restarted elsewhere.

HTCondor can be configured so that processes are merely suspended or even allowed to continue running, but this requires permission from the owner of the PC. Since the PCs on an HTCondor pool may be owned by many different people, it's best to assume that at least some of the PCs are configured to evict HTCondor processes when a local user is active.

A computer used by HTCondor should therefore *not* be expected to remain available for long periods of time. Plan for some of the processes running under HTCondor to be terminated and restarted on a different machine.

The longer an HTCondor process runs, the less likely it is to complete. As a rule of thumb, an HTCondor process should ideally run no more than a few hours. This is usually easy to achieve. Most HTCondor users are running embarrassingly parallel jobs that simply divide up input data or parameter space among an arbitrary number of independent processes. If such a user has 1,000 hours worth of computation, they can just as easily do it with 10 jobs running for 100 hours each, 100 jobs running for 10 hours each, or 500 jobs running for 2 hours each.

Don't make your processes *too* short, though. Each process entails some scheduling overhead, and may take up to a minute or so to start. We want our jobs to spend much more time running than sitting in a queue in order to maximize throughput. There is little benefit to breaking computation into jobs much shorter than an hour and scheduling overhead will become significant if you do.

One of the advantages of HTCondor is that the number of available cores is usually irrelevant to how you divide up your job. HTCondor will simply run as many processes as it can at any given moment and start others as cores become available. For example, even if you only have 200 cores available, you can run a job consisting of 500 processes. HTCondor will run up to 200 at a time until all 500 are finished. 500 processes running for 2 hours each is usually preferable to 100 processes of 10 hours each, since this reduces the risk of individual processes being evicted from a PC.

### 9.1.2 HTCondor Terminology

- An *execute host* is a computer available for use by HTCondor for running a user's processes.
- A *job* is usually one process running on an execute host managed by HTCondor.
- A *cluster* (not to be confused with a hardware cluster) is the group of jobs from a single condor_submit, all of which share the same numeric ID.
- A *submit host* is a computer from which HTCondor jobs can be submitted.
- A *central manager* is a computer that manages a specific *pool* of computers which include execute hosts, submit hosts, and possibly other types of hosts. Every HTCondor pool has it's own central manager.
- A *grid* is a group of computers available for parallel computing. It consists of one or more pools, each with it's own central manager.
- *Flocking* is a system that allows jobs from one pool that has run out of resources to migrate to another pool, much like a flock of birds migrating when they run out of food.

### 9.1.3 Pool Status

Before scheduling any jobs on through HTCondor, it is often useful to check the status of the hosts.

The **condor_status** command shows the status of all the execute hosts in the pool.

```
FreeBSD peregrine bacon ~ 402: condor_status | more

Name               OpSys      Arch   State     Activity LoadAv Mem   ActvtyTime

povbEEFF73C040D8.l LINUX      INTEL  Backfill  Idle     0.050  1968  0+00:00:03
povbEEFF73C04485.l LINUX      INTEL  Backfill  Idle     0.080  1968  0+00:00:04
povbEEFF73C046A1.l LINUX      INTEL  Backfill  Idle     0.020  1968  0+00:00:04
povbEEFF73C048DB.l LINUX      INTEL  Backfill  Idle     0.000  1968  0+00:00:04
povbEEFF73C04B9F.l LINUX      INTEL  Backfill  Idle     0.130  1968  0+00:00:04
povbEEFF73C04BF1.l LINUX      INTEL  Backfill  Idle     0.110  1968  0+00:00:04
povbEEFF73C0516E.l LINUX      INTEL  Backfill  Idle     0.050  1968  0+00:00:04
povbEEFF73C061C5.l LINUX      INTEL  Backfill  Idle     0.070  1968  0+00:00:04
povbEEFF73C06F85.l LINUX      INTEL  Backfill  Idle     0.130  1968  0+00:00:04
povbEEFF73C0873B.l LINUX      INTEL  Backfill  Idle     0.150  1968  0+00:00:04
povbEEFF73C08BA6.l LINUX      INTEL  Backfill  Idle     0.020  1968  0+00:00:04
povbEEFF73C08E57.l LINUX      INTEL  Backfill  Idle     0.070  1968  0+00:00:04
povbEEFFA0A4F157.l LINUX      INTEL  Owner     Idle     0.290  1009  0+09:32:43
povbEEFFA0AA6F42.l LINUX      INTEL  Backfill  Idle     0.090  1009  0+00:00:04
povbEEFFA0AA717C.l LINUX      INTEL  Backfill  Idle     0.370  1009  0+00:00:04
povbEEFFA0AD6A4C.l LINUX      INTEL  Backfill  Idle     0.100  1009  0+00:00:04
povbEEFFA0B8E890.l LINUX      INTEL  Backfill  Idle     0.080  1009  0+00:00:04
povbEEFFA0DDD5C2.l LINUX      INTEL  Backfill  Idle     0.140  1009  0+00:00:03
povbEEFFA0DDD60F.l LINUX      INTEL  Backfill  Idle     0.200  1009  0+00:00:05
povbEEFFA0DDD652.l LINUX      INTEL  Backfill  Idle     0.200  1009  0+00:00:03
--More--(byte 1682)
```

The **condor_status** command shows only resources available in the local pool. If flocking is configured, it will be possible for jobs to utilize resources in other pools, but those resources are not shown by **condor_status** unless specifically requested.

### 9.1.4 Job Submission

The purpose of this section is to provide the reader a quick start in job scheduling using the most common and useful tools. The full details of job submission are beyond the scope of this document.

#### Submit Description Files

Submitting jobs involves specifying a number of job cluster parameters such as the number of cores, the job cluster name (which is displayed by qstat), the name(s) of the output file(s), etc.

In order to record all of this information and make it easy to resubmit the same job cluster, this information is incorporated into a *submit description file*. Using a submit description file saves you a lot of typing when you want to run-submit the same job cluster, and also fully documents the job cluster parameters.

An HTCondor submit description file is a file containing a number of HTCondor variable assignments and commands that indicate the resource requirements of the program to be run, the number of instances to run in parallel, the name of the executable, etc.

**Note**
Unlike the submission scripts of other schedulers such as LSF, PBS, and SLURM, a condor submit description file is *not* a Unix shell script and hence is not executed on the compute hosts. It is a special type of file specific to HTCondor.
Recall that in most other schedulers, the job cluster parameters are embedded in the script as specially formatted comments beginning with "#SBATCH", "#PBS", or something similar. In HTCondor, the job cluster parameters and the script or program to be run on the compute hosts are kept in separate files.

HTCondor pools are often heterogeneous, i.e. the hosts in the HTCondor pool may run different operating systems, have different amounts of RAM, etc. For this reason, resource requirements are a must in many HTCondor description files.

Furthermore, HTCondor pools typically do not have any shared storage available to all hosts, so the executable, the input files, and the output files are often transferred to and from the execute hosts. Hence, the names of these files must be specified in the submit description file.

Suppose we have the following text in a file called `hostname.condor`:

```
############################################################################
#
# hostname.condor
#
# Condor Universe
#
# standard:
#      Defaults to transfer executables and files.
#      Use when you are running your own script or program.
#
# vanilla:
# grid:
#      Explicitly enable file transfer mechanisms with
#      'transfer_executable', etc.
#      Use when you are using your own files and some installed on the
#      execute hosts.
#
# parallel:
#      Explicitly enable file transfer mechanism. Used for MPI jobs.

universe = vanilla

# Macros (variables) to use in this description file
# This is our own custom macro.  It has no special meaning to HTCondor.
Process_count = 5
```

```
############################################################################
# Specify the executable filename.  This can be a binary file or a script.
# HTCondor does not search $PATH!  A relative pathname here refers to an
# executable in the current working directory.  To run a standard Unix command
# use an absolute pathname, or set executable to a script.
executable = /bin/hostname

# Output from executable running on the execute hosts
# $(Process) is a predefined macro containing a different integer value for
# each process, ranging from 0 to Process_count-1.
output = hostname.out-$(Process)
error = hostname.err-$(Process)

# Log file contains status information from HTCondor
log = hostname.log

############################################################################
# Condor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!!

# Requirements to match any FreeBSD or Linux host, 32 or 64 bit processors.
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
               ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Memory requirements in mebibytes
request_memory = 10

# Executable is a standard Unix command, already on every execute host
transfer_executable = false

# Specify how many jobs you would like to submit to the queue.
queue $(Process_count)
```

The description file is submitted to the HTCondor scheduler as a command line argument to the **condor_submit** command:

```
shell-prompt: condor_submit hostname.condor
```

This will run /bin/hostname on each of 5 execute hosts.

```
FreeBSD login.peregrine bacon ~ 540: cat hostname.out-*
FBVM.10-4-60-177.meadows
FBVM.10-4-60-231.meadows
FBVM.10-4-60-227.meadows
FBVM.10-4-60-219.meadows
FBVM.10-4-60-231.meadows
```

The **condor_submit** command, which is part of the HTCondor scheduler, finds a free core on a execute host, reserves it, transfers the executable and input files from the submit host to the execute host if necessary, and then runs the executable on the execute host.

The executable named in the description is dispatched to the host(s) containing the core(s) allocated by HTCondor, using **ssh**, **rsh** or any other remote shell HTCondor is configured to use.

Since an HTCondor submit description file specifies a single executable, we must create a script in addition to the submit description file in order to run multiple commands in sequence.

```
#!/bin/sh -e

# hostname.sh

hostname
```

```
uname
pwd
ls
```

We must then alter the submit description file so that our script is the executable, and it is transferred to the execute host:

```
############################################################################
#
# hostname-script.condor
#
# Condor Universe
#
# standard:
#      Defaults to transfer executables and files.
#      Use when you are running your own script or program.
#
# vanilla:
# grid:
#      Explicitly enable file transfer mechanisms with
#      'transfer_executable', etc.
#      Use when you are using your own files and some installed on the
#      execute hosts.
#
# parallel:
#      Explicitly enable file transfer mechanism. Used for MPI jobs.

universe = vanilla

# Macros (variables) to use in this description file
# This is our own custom macro.  It has no special meaning to HTCondor.
Process_count = 5

############################################################################
# Specify the executable filename.  This can be a binary file or a script.
# HTCondor does not search $PATH!  A relative pathname here refers to an
# executable in the current working directory.  To run a standard Unix command
# use an absolute pathname, or set executable to a script.
executable = hostname.sh

# Output from executable running on the execute hosts
# $(Process) is a predefined macro containing a different integer value for
# each process, ranging from 0 to Process_count-1.
output = hostname.out-$(Process)
error = hostname.err-$(Process)

# Log file contains status information from HTCondor
log = hostname.log

############################################################################
# Condor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!!

# Requirements to match any FreeBSD or Linux host, 32 or 64 bit processors.
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
               ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Memory requirements in mebibytes
request_memory = 10

# Executable is our own script
transfer_executable = true
```

```
# Specify how many jobs you would like to submit to the queue.
queue $(Process_count)
```

The output of one job will appear as follows:

```
FreeBSD login.peregrine bacon ~ 542: cat hostname.out-0
FBVM.10-4-60-177.meadows
FreeBSD
/htcondor/Data/execute/dir_7104
_condor_stderr
_condor_stdout
condor_exec.exe
```

We can write a simple script to remove output and log files:

```
#!/bin/sh -e

# hostname-cleanup.sh

rm -f hostname.out-* hostname.err-* hostname.log
```

**Note**
Since many HTCondor pools are heterogeneous, you must make sure that the executable is portable, or that you specify resource requirements to ensure that it will only be dispatched to hosts on which it will work.
Using Bourne shell scripts (not bash, ksh, csh, or tcsh) will maximize the portability of a shell script, as discussed in Chapter 4. If you must use different shell, be sure to use a portable shebang line (e.g. #!/usr/bin/env bash, not #!/bin/bash).

### Running Your Own Compiled Programs

As HTCondor grids may be heterogeneous (running a variety of operating systems and CPU architectures), we may need more than one binary (compiled) file in order to utilize all available hosts. Maintaining multiple binaries and transferring the right one to each host can be tedious and error-prone.

As long as the source code is portable to all execute hosts, we can avoid this issue by compiling the program on each execute host as part of the job.

Below is a submit description file that demonstrates how to use an executable script to compile and run a program on each execute host. Note that the program source code is sent to the execute host as an input file, while the script that compiles it is the executable.

```
############################################################################
#
# hostname-c.condor
#
# Condor Universe
#
# standard:
#      Defaults to transfer executables and files.
#      Use when you are running your own script or program.
#
# vanilla:
# grid:
#      Explicitly enable file transfer mechanisms with
#      'transfer_executable', etc.
#      Use when you are using your own files and some installed on the
#      execute hosts.
#
# parallel:
```

```
#       Explicitly enable file transfer mechanism. Used for MPI jobs.

universe = vanilla

# Macros (variables) to use in this description file
# This is our own custom macro.  It has no special meaning to HTCondor.
Process_count = 5

############################################################################
# Specify the executable filename.  This can be a binary file or a script.
# HTCondor does not search $PATH!  A relative pathname here refers to an
# executable in the current working directory.  To run a standard Unix command
# use an absolute pathname, or set executable to a script.
executable = hostname-c.sh

# Output from executable running on the execute hosts
# $(Process) is a predefined macro containing a different integer value for
# each process, ranging from 0 to Process_count-1.
output = hostname.out-$(Process)
error = hostname.err-$(Process)

# Log file contains status information from HTCondor
log = hostname.log

############################################################################
# Condor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!!

# Requirements to match any FreeBSD or Linux host, 32 or 64 bit processors.
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
               ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Memory requirements in mebibytes
request_memory = 50

# Executable is our own script
transfer_executable = true

# Send the source code as an input file for the executable
transfer_input_files = hostname.c

# Specify how many jobs you would like to submit to the queue.
queue $(Process_count)
```

The executable script is below. We set the sh -x flag in this example so that we can see the commands executed by the script on the execute hosts.

```
#!/bin/sh -e

# hostname-c.sh

# HTCondor environment contains a minimal PATH, so help it find cc
PATH=/bin:/usr/bin
export PATH

# Echo commands
set -x

pwd
cc -o hostname hostname.c
./hostname
```

Sample output from one of the jobs:

```
FreeBSD login.peregrine bacon ~ 402: cat hostname.out-0
/htcondor/Data/execute/dir_7347
Hello from FBVM.10-4-60-177.meadows!

FreeBSD login.peregrine bacon ~ 403: cat hostname.err-0
+ pwd
+ cc -o hostname hostname.c
+ ./hostname
```

#### Common Features

- Anything from a '#' character to the end of a line is considered a comment, and is ignored by HTCondor.
- universe: Categorizes jobs in order to set reasonable defaults for different job types and thus reduce the number of explicit settings in the HTCondor script.
- executable: Name of the program or script to run on the execute host(s). This is often transferred from the submit host to the execute hosts, or even compiled on the execute hosts at the start of the job.
- output: Name of a file to which the standard output (normal terminal output) is redirected. Jobs have no connection to the terminal while they run under HTCondor, so the output must be saved to a file for later viewing.
- error: Name of a file to which the standard error (error messages normally sent to the terminal) is redirected.
- input: Name of a file from which to redirect input the program would expect from the standard input (normally the keyboard).
- log: Name of a file to which HTCondor saves it's own informative messages about the job cluster.
- requirements: Used to describe the requirements for running the program, such as operating system, memory needs, CPU type, etc.
- transfer_executables = yes|no: Indicate whether the executable specified with the executable variable must be transferred to the execute host before the job executes.
- should_transfer_files = yes|no: Indicate whether input and output files must be transferred to/from the execute host. Normally yes unless a shared file system is available.
- when_to_transfer_output_files: Indicates when output files should be transferred from the execute host back to the submit host. Normally use "on_exit".
- transfer_input_files = list: Comma-separated list of input files to transfer to the execute host.
- transfer_input_files = list: Comma-separated list of output files to transfer to the execute host.
- queue [count]: Command to submit the job cluster to one or more execute hosts. If no count is given, if defaults to 1 host.

#### HTCondor Resource Requirements

When using a grid, it is important to develop a feel for the resources required by your jobs, and inform the scheduler as accurately as possible what will be needed in terms of CPU time, memory, etc.

You may not know this the first time you run a given job, but after examining the log files from one or more runs, you will have a pretty good idea.

This allows the scheduler to maximize the utilization of precious resources and thereby provide the best possible run times for all users.

HTCondor resource requirements are specified with the requirements variable. There are many parameters available to specify the operating system, memory requirements, etc. Users should try to match requirements as closely as possible to the actual requirements of the job cluster.

For example, if your job requires only 300 megabytes of RAM, then by specifying this, you can encourage HTCondor to save the hosts with several gigabytes of RAM for the jobs that really need them.

#### Batch Serial Jobs

A batch serial submit description file need only specify basic information such as the executable name, input and output files, and basic resource requirements.

```
universe = vanilla
executable = hostname.sh
output = $(Process).stdout
error = $(Process).stderr
log = hostname.log
transfer_executable = yes
should_transfer_files = yes
when_to_transfer_output = on_exit

queue 1
```

#### Batch Parallel Jobs (Job Clusters)

A job cluster is a set of independent processes all started by a single job submission.

A batch parallel submit description file looks almost exactly like a batch serial file, but indicates a process count greater than 1 following the queue command:

```
universe = vanilla
executable = hostname.sh
output = $(Process).stdout
error = $(Process).stderr
log = hostname.log
transfer_executable = yes
should_transfer_files = yes
when_to_transfer_output = on_exit

queue 10
```

#### MPI (Multi-core) Jobs

Most pools are not designed to run MPI jobs. MPI jobs often require a fast dedicated network to accommodate extensive message passing. A pool implemented on lab or office PCs is not suitable for this, a typical local area network in a lab is not fast enough to offer good performance, and MPI traffic may in fact overload it, causing problems for other users.

HTCondor does provide facilities for running MPI jobs, but they are usually only useful where HTCondor is employed as the scheduler for a cluster.

#### A Submit Description File Template

Below is a sample submit description file with typical options and extensive comments. This script is also available in `/share1/Examples` on Peregrine.

```
############################################################################
# Sample HTCondor submit description file.
#
# Use \ to continue an entry on the next line.
#
# You can query your jobs by command:
# condor_q

############################################################################
# Choose which universe you want your program is running with
# Available options are
#
# - standard:
#      Defaults to transfer executables and files.
#      Use when you are running your own script or program.
#
# - vanilla:
# - grid:
#      Explicitly enable file transfer mechanisms with
#      'transfer_executable', etc.
#      Use when you are using your own files and some installed on the
#      execute hosts.
#
# - parallel:
#      Explicitly enable file transfer mechanism. Used for MPI jobs.

universe = vanilla

# Macros (variables) to use in this submit description file
Points = 1000000000
Process_count = 10

############################################################################
# Specify the executable filename.  This can be a binary file or a script.
# NOTE: The POVB execute hosts currently support 32-bit executables only.
# If compiling a program on the execute hosts, this script should compile
# and run the program.
#
# In template.sh, be sure to give the executable a different
# name for each process, since multiple processes could be on the same host.
# E.g. cc -O -o prog.$(Process) prog.c

executable = template.sh

# Command-line arguments for executing template.sh
# arguments =

############################################################################
# Set environment variables for use by the executable on the execute hosts.
# Enclose the entire environment string in quotes.
# A variable assignment is var=value (no space around =).
# Separate variable assignments with whitespace.

environment = "Process=$(Process) Process_count=$(Process_count) Points=$(Points)"

############################################################################
# Where the standard output and standard error from executables go.
# $(Process) is current job ID.

# If running template under both PBS and HTCondor, use same output
# names here as in template-run.pbs so that we can use the same
# script to tally all the outputs from any run.

output = template.out-$(Process)
error = template.err-$(Process)

############################################################################
# Logs for the job, produced by HTCondor.  This contains output from
# HTCondor, not from the executable.

log = template.log
```

```
############################################################################
# Custome job requirements
# HTCondor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!!
# For example, if the jobs is submitted from peregrine, target.arch is
# "X86_64" and target.opsys is "FREEBSD8", which do not match
# POVB execute hosts.
#
# You can query if your submitting host is accepted by command:
#  condor_q -analyze

# Memory requirements in megabytes
request_memory = 50

# Requirements for a binary compiled on CentOS 4 (POVB hosts):
# requirements = (target.arch == "INTEL") && (target.opsys == "LINUX")

# Requirements for a Unix shell script or Unix program compiled on the
# execute host:
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
               ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Requirements for a job utilizing software installed via FreeBSD ports:
# requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
#    (target.opsys == "FREEBSD")

# Match specific compute host names
# requirements = regexp("compute-s.*.meadows", Machine)

############################################################################
# Explicitly enable executable transfer mechanism for vanilla universe.

# true | false
transfer_executable = true

# yes | no | if_needed
should_transfer_files = if_needed

# All files to be transferred to the execute hosts in addition to the
# executable.  If compiling on the execute hosts, list the source file(s)
# here, and put the compile command in the executable script.
transfer_input_files = template.c

# All files to be transferred back from the execute hosts in addition to
# those listed in "output" and "error".
# transfer_output_files = file1,file2,...

# on_exit | on_exit_or_evict
when_to_transfer_output = on_exit

############################################################################
# Specify how many jobs you would like to submit to the queue.

queue $(Process_count)
```

### 9.1.5 Job Status

While your jobs are running, you can check on their status using **condor_q**.

```
FreeBSD peregrine bacon ~ 533: condor_q

-- Submitter: peregrine.hpc.uwm.edu : <129.89.25.224:37668> : peregrine.hpc.uwm.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
  63.0   bacon           6/5  13:45   0+00:00:00 I  0   0.0  hostname.sh
  64.0   bacon           6/5  13:46   0+00:00:00 I  0   0.0  hostname.sh

2 jobs; 2 idle, 0 running, 0 held
```

The ST column indicates job status (R = running, I = idle, C = completed). Run **man condor_q** for full details.

### 9.1.6 Terminating a Job or Cluster

If you determine that a job is not behaving properly (by reviewing partial output, for example), you can terminate it using **condor_rm**, which take a job ID or job cluster ID as a command line argument.

To terminate a single job within a cluster, use the job ID form cluster-id.job-index:

```
FreeBSD peregrine bacon ~ 534: condor_rm 63.0
Job 63.0 marked for removal
FreeBSD peregrine bacon ~ 535: condor_q


-- Submitter: peregrine.hpc.uwm.edu : <129.89.25.224:37668> : peregrine.hpc.uwm.edu
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
  64.0   bacon           6/5  13:46   0+00:00:00 I  0   0.0  hostname.sh

1 jobs; 1 idle, 0 running, 0 held
```

To terminate an entire job cluster, just provide the job cluster ID:

```
FreeBSD peregrine bacon ~ 534: condor_rm 63
```

### 9.1.7 Job Sequences

If you need to submit a series of jobs in sequence, where one job begins after another has completed,

It's important to make sure that the current job completed successfully before submitting the next, to avoid wasting resources. It is up to you to determine the best way to verify that a job was successful. Examples might include grepping the log file for some string indicating success, or making the job create a marker file using the **touch** command after a successful run. If the command used in your job returns a Unix-style exit status (0 for success, non-zero on error), then you can simply use the shell's exit-on-error feature to make your script exit when any command fails. Below is a template for scripts that might run a series of jobs.

```
#!/bin/sh

condor_submit job1.condor
condor_wait job1-log-file

# Verify that job1 completed successfully, by the method of your choice
if ! test-to-indicate-job1-succeeded; then
    exit 1
fi

condor_submit job2.condor
```

### 9.1.8 Self-test

1. What command is used to check the status of an HTCondor pool?
2. What kind of file is used to describe an HTCondor job? Is this file a shell script?
3. Does HTCondor support MPI jobs? Explain.
4. How can you check the status of your HTCondor jobs?
5. How can you terminate a running HTCondor job?
6. Write and submit a batch-serial HTCondor script called `list-etc.condor` that prints the host name of the execute host on which it runs and a long-listing of `/etc` directory on that host.

   The script should store the output of the commands in `list-etc.stdout` and error messages in `list-etc.stderr` in the directory from which the script was submitted.

   Quickly check the status of your job after submitting it.
7. Copy your `list-etc.condor` script to `list-etc-parallel.condor`, and modify it so that it runs the **hostname** and **ls** commands on 10 cores instead of just one.

   The job should produce a separate output file for each process named `list-etc-parallel.o<jobid>` and a separate error file for each process named `list-etc-parallel.e<jobid>`.

   Quickly check the status of your job after submitting it.

## 9.2 Local Customizations

TBD

# Chapter 10

# Job Scheduling with PBS (TORQUE)

**Before You Begin**
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 3), the Unix shell (Section 3.3.3, redirection (Section 3.13), shell scripting (Chapter 4) and job scheduling (Chapter 7.

For complete information, see the official TORQUE documentation at http://www.adaptivecomputing.com/resources/docs/torque/-3-0-2/index.php.

## 10.1 The PBS Scheduler

### 10.1.1 Cluster Status

The goal of this section is to provide a quick introduction to the PBS scheduler. Specifically, this section covers the current mainstream implementation of PBS, known as TORQUE.

Before scheduling any jobs on through PBS, it is often useful to check the status of the nodes. Knowing how many cores are available may influence your decision on how many cores to request for your next job.

For example, if only 50 cores are available at the moment, and your job requires 200 cores, the job will have to wait in the queue until 200 cores are free. You may end up getting your results sooner if you reduce the number of cores to 50 or less so that the job can begin right away.

The **pbsnodes** command shows information about the nodes in the cluster. This can be used to determine the total number of cores in the cluster, cores in use, etc.

```
FreeBSD peregrine bacon ~/Facil 408: pbsnodes|more
compute-001
     state = job-exclusive
     np = 12
     ntype = cluster
     jobs = 0/1173[20].peregrine.hpc.uwm.edu, 1/1173[20].peregrine.hpc.uwm.edu, 2/1173[20]. ↩
         peregrine.hpc.uwm.edu, 3/1173[20].peregrine.hpc.uwm.edu, 4/1173[20].peregrine.hpc. ↩
         uwm.edu, 5/1173[20].peregrine.hpc.uwm.edu, 6/1173[20].peregrine.hpc.uwm.edu,  ↩
         7/1173[20].peregrine.hpc.uwm.edu, 8/1173[20].peregrine.hpc.uwm.edu, 9/1173[20]. ↩
         peregrine.hpc.uwm.edu, 10/1173[20].peregrine.hpc.uwm.edu, 11/1173[20].peregrine.hpc ↩
         .uwm.edu
     status = rectime=1331737185,varattr=,jobs=1173[20].peregrine.hpc.uwm.edu,state=free, ↩
         netload=? 15201,gres=,loadave=2.00,ncpus=12,physmem=32515040kb,availmem=? 15201, ↩
         totmem=? 15201,idletime=6636140,nusers=3,nsessions=3,sessions= 72397 13348 1005, ↩
         uname=FreeBSD compute-001.local 8.2-RELEASE-p3 FreeBSD 8.2-RELEASE-p3 #0: Tue Sep  ↩
         27 18:45:57 UTC 2011     root@amd64-builder.daemonology.net:/usr/obj/usr/src/sys/ ↩
         GENERIC amd64,opsys=freebsd5
     mom_service_port = 15002
     mom_manager_port = 15003
     gpus = 0

[text removed for brevity]

compute-08
     state = free
     np = 12
     ntype = cluster
     status = rectime=1331737271,varattr=,jobs=,state=free,netload=? 15201,gres=,loadave ←
         =0.00,ncpus=12,physmem=32515040kb,availmem=? 15201,totmem=? 15201,idletime=6642605, ←
         nusers=2,nsessions=2,sessions= 18723 1034,uname=FreeBSD compute-08.local 8.2- ←
         RELEASE-p3 FreeBSD 8.2-RELEASE-p3 #0: Tue Sep 27 18:45:57 UTC 2011     root@amd64- ←
         builder.daemonology.net:/usr/obj/usr/src/sys/GENERIC amd64,opsys=freebsd5
     mom_service_port = 15002
     mom_manager_port = 15003
     gpus = 0
```

As a convenience, a script called **cluster-load** is installed on Peregrine. This script extracts information from **pbsnodes** and displays the current cluster load in an abbreviated, easy-to-read format.

```
            FreeBSD peregrine bacon ~/Facil 406: cluster-load
            Nodes in use: 5
            Cores in use: 60
            Total cores:  96
            Free cores:   36
            Load:         62%
```

#### The Ganglia Resource Monitor

In addition to the command line tools used to control and monitor jobs, there are also web-based tools for monitoring clusters and grids in a more visual manner.

The *Ganglia Resource Monitor* is a web-based monitoring tool that provides statistics about a collection of computers on a network.

The status of the student cluster, Peregrine, can be viewed at http://www.peregrine.hpc.uwm.edu/ganglia/.

The status of the faculty research cluster, Mortimer, can be viewed at http://www.mortimer.hpc.uwm.edu/ganglia/.

### 10.1.2 Job Status

You can check on their status of running jobs using **qstat**.

```
            peregrine: qstat
            Job id           Name             User    Time Use S Queue
            ---------------- ---------------- ------ --------- - -----
            52[].peregrine   ...allel-example bacon          0 C batch
```

The Job id column shows the numeric job ID. [] indicate a job array. The S column shows the current status of the job. The most common status flags are ’Q’ for queued (waiting to start), ’R’ for running, and ’C’ for completed.

The scheduler retains job status information for a short time after job completion. The amount of time is configured by the systems manager, so it will be different on each site.

The **qstat** `-f` flag requests more detailed information. Since it produces a lot of output, it is typically piped through more and/or used with a specific job-id:

```
peregrine: qstat -f 2151 | more
Job Id: 2151.peregrine.hpc.uwm.edu
    Job_Name = list-etc.pbs
    Job_Owner = bacon@peregrine.hpc.uwm.edu
    resources_used.cput = 00:00:00
    resources_used.mem = 0kb
    resources_used.vmem = 0kb
    resources_used.walltime = 00:00:00
    job_state = C
    queue = batch
    server = peregrine.hpc.uwm.edu
    Checkpoint = u
    ctime = Mon Jun  4 14:44:39 2012
    Error_Path = peregrine.hpc.uwm.edu:/home/bacon/Computer-Training/Common/li
  st-etc.stderr
    exec_host = compute-03/0
    exec_port = 15003
    Hold_Types = n
    Join_Path = n
    Keep_Files = n
    Mail_Points = a
    mtime = Mon Jun  4 14:44:39 2012
    Output_Path = peregrine.hpc.uwm.edu:/home/bacon/Computer-Training/Common/l
  ist-etc.stdout
--More--(byte 720)
```

The **qstat** command has many flags for controlling what it reports. Run **man qstat** for full details.

### 10.1.3 Using top

Using the output from **qstat -f**, we can see which compute nodes are being used by a job.

We can then examine the processes on a given node using a remotely executed **top** command:

```
            peregrine: ssh -t compute-003 top
```

**Note** The `-t` flag is important here, since it tells ssh to open a connection with full terminal control, which is needed by **top** to update your terminal screen.

On peregrine, a convenience script is provided to save typing:

```
            peregrine: topnode 003
```

### 10.1.4 Job Submission

The purpose of this section is to provide the reader a quick start in job scheduling using the most common and useful tools. The full details of job submission are beyond the scope of this document.

#### Submission Scripts

Submitting jobs involves specifying a number of job parameters such as the number of cores, the job name (which is displayed by **qstat**), the name(s) of the output file(s), etc.

In order to record all of this information and make it easy to resubmit the same job, this information is usually incorporated into a *submission script*. Using a script saves you a lot of typing when you want to run-submit the same job, and also fully documents the job parameters.

A submission script is an ordinary shell script, with some *directives* inserted to provide information to the scheduler. For PBS, the directives are specially formatted shell comments beginning with "#PBS".

Suppose we have the following text in a file called `hostname.pbs`:

```
#!/usr/bin/env bash

# A PBS directive
#PBS -N hostname

# A command to be executed on the scheduled node(s)
# Prints the host name of the node running this script.
hostname
```

The script is submitted to the PBS scheduler as a command line argument to the **qsub** command:

```
peregrine: qsub hostname.pbs
```

The **qsub** command, which is part of the PBS scheduler, finds a free core on a compute node, reserves it, and then runs the script on the compute node using **ssh** or some other remote execution command.

Comments beginning with #PBS are interpreted as directives by **qsub** and as any other comment by the shell. Recall that the shell ignores anything on a line after a '#'.

The command(s) in the script are dispatched to the node(s) containing the core(s) allocated by PBS, using **ssh**, **rsh** or any other remote shell PBS is configured to use.

The directives within the script provide command line flags to **qsub**. For instance, the line

```
#PBS -N hostname
```

causes **qsub** to behave as if you had typed

```
peregrine: qsub -N hostname hostname.pbs
```

By putting these comments in the script, you eliminate the need to remember them and retype them every time you run the job. It's generally best to put all **qsub** flags in the script rather than type any of them on the command line, so that you have an exact record of how the job was started. This will help you determine what went wrong if there are problems, and allow you to reproduce the results at a later time without worrying about whether you did something different.

The script itself can be any valid Unix script, using the shell of your choice. Since all Unix shells interpret the '#' as the beginning of a comment, the #PBS lines will be interpreted only by qsub, and ignored by the shell.

**Note** If you want to disable a #PBS comment, you can just add another '#' rather than delete it. This will allow you to easily enable it again later.

```
##PBS This line is ignored by qsub
#PBS This line is interpreted by qsub
```

It's a good idea to use a modern shell such as **bash**, **ksh**, or **tcsh**, simply because they are more user-friendly than **sh** or **csh**.

**Practice Break**
Type in the `hostname.pbs` script shown above and submit it to the scheduler using **qsub**. Then check the status with **qstat** and view the output and error files.

### Common Flags

```
#PBS -N job-name
#PBS -o standard-output-file  (default = <job-name>.o<job-id>)
#PBS -e standard-error-file   (default = <job-name>.e<job-id>)
#PBS -l resource-requirements
#PBS -M email-address
#PBS -t first-last
```

The `-N` flag gives the job a name which will appear in the output of **qstat**. Choosing a good name makes it easier to keep tabs on your running jobs.

The `-o` and `-e` flags control the name of the files to which the standard output and standard error of the processes are redirected. If omitted, a default name is generated using the job name and job ID.

The `-l` flag is used to specify resource requirements for the job, which are discussed in Section 10.1.4.

The `-t` flag indicates the starting and ending subscripts for a job array. This is discussed in Section 10.1.4.

### PBS Resource Requirements

When using a cluster, it is important to develop a feel for the resources required by your jobs, and inform the scheduler as accurately as possible what will be needed in terms of CPU time, memory, etc.

This allows the scheduler to maximize the utilization of precious resources and thereby provide the best possible run times for all users.

If a user does not specify a given resource requirement, the scheduler uses default limits. Default limits are set low, so that users are encouraged to provide an estimate of required resources for all non-trivial jobs. This protects other users from being blocked by long-running jobs that require less memory and other resources than the scheduler would assume.

PBS resource requirements are specified with the **qsub** `-l` flag.

```
#PBS -l procs=count
#PBS -l nodes=node-count:ppn=procs-per-node
#PBS -l cput=seconds
#PBS -l cput=hours:minutes:seconds
#PBS -l vmem=size[kb|mb|gb|tb]
#PBS -l pvmem=size[kb|mb|gb|tb]
```

The `procs` resource indicates the number of processors (cores) used by the job. This resource must be specified for parallel jobs such as MPI jobs, which are discussed in Section 10.1.4. `Count` cores are allocated by the scheduler according to its own policy configuration. Some cores may be on the same node, depending on the current load distribution.

The `nodes` and `ppn` resources allow the user to better control the distribution of cores allocated to a job. For example, if you want 8 cores all on the same node, you could use `-l nodes=1:ppn=8`. If you want 20 cores, each on a different node, you could use `-l nodes=20:ppn=1`. The best distribution depends on the nature of both your own job and other jobs running at the time. Spreading a job across more nodes will generally reduce memory and disk contention between processes within the job, but also increase communication cost between processes in an MPI job.

Note that specifying a ppn value equal to the number of cores in a node ensures exclusive access to that node while the job is running. This is especially useful for jobs with high memory or disk requirements, where we want to avoid contending with other users' jobs.

The `cput` resource limits the total CPU time used by the job. Most jobs should specify a value somewhat higher than the expected run time simply to prevent program bugs from consuming excessive cluster resources. As noted in the example above, CPU time may be specified either as a number of seconds, or in the format HH:MM:SS.

Memory requirements can be specified either as the total memory for the job (`vmem`) or as memory per process within the job (`pvmem`). The `pvmem` flag is generally more convenient, since it does not have to be changed when `procs`, `nodes`, or `ppn` is changed.

**Note** It is a very important to specify memory requirements accurately in all jobs, and it is generally easy to predict based on previous runs by monitoring processes within the job using **top** or **ps**. Failure to do so could block other jobs run running, even though the resources it requires are actually available.

### Batch Serial Jobs

A batch serial submission script need only have optional PBS flags such as job name, output file, etc. and one or more commands.

```
#!/usr/bin/env bash

#PBS -N hostname

hostname
```

For simple serial jobs, a convenience script called **qsubw** is provided. This script submits a job, waits for the job to complete, and then immediately displays the output files. This type of operation is convenient for compiling programs and performing other simple tasks that must be scheduled, but for which we intend to wait for completion and immediately perform the next step.

**Caution** As currently implemented, **qsubw** ignores #PBS lines in the submit script containing `-N`, `-o` and `-e`. The entire line is ignored, including other flags. Hence, these flags should not be combined in the same line with other flags in any scripts run through **qsubw**.

```
FreeBSD peregrine bacon ~/Data/Testing/RLA 503: qsubw compile.pbs
    Job ID = 298
    exec_host = compute-02/0

compile.pbs.tmp.e298:
RLA.cpp: In function 'double Random()':
RLA.cpp:271: warning: integer overflow in expression
RLA.cpp: In function 'void SelectBidAction()':
RLA.cpp:307: warning: integer overflow in expression
FreeBSD peregrine bacon ~/Data/Testing/RLA 504: qsub RLA.pbs
```

### Batch Parallel Jobs (Job Arrays)

A job array is a set of independent processes all started by a single job submission. The entire job array can be treated as a single job by PBS commands such as **qstat** and **qdel**, but individual processes are also jobs in and of themselves, and can therefore by manipulated individually via PBS commands.

A batch parallel submission script looks almost exactly like a batch serial script, but requires just one additional flag:

```
#!/usr/bin/env bash

#PBS -N hostname-parallel
#PBS -t 1-5

hostname
```

The `-t` flag is followed a list of integer array IDs. The specification of array IDs can be fairly sophisticated, but is usually a simple range.

The example above requests a job consisting of 5 identical processes which will all be under the job name hostname-parallel. Each process within the job produces a separate output file with the array ID as a suffix. For example, process 2 produces an output file named hostname-parallel.o51-2. ( 51 is the job ID, and 2 is the array ID within the job. )

**Caution** The syntax `-t N` does not work as advertised in the official TORQUE documentation, which states that it is the same as `-t 0-N`. In reality, it creates a single job with subscript N.

**Practice Break**
Copy your `hostname.pbs` to `hostname-parallel.pbs`, modify it to run 5 processes, and submit it to the scheduler using **qsub**. Then check the status with **qstat** and view the output and error files.

### MPI (Multi-core) Jobs

Scheduling MPI jobs is actually much like scheduling batch serial jobs. This may not seem intuitive at first, but once you understand how MPI works, it makes more sense.

MPI programs cannot be executed directly from the command line as we do with normal programs and scripts. Instead, we must use the **mpirun** command to start up MPI programs.

```
mpirun [mpirun flags] mpi-program [mpi-program arguments]
```

**Caution** Like any other command used on a cluster or grid, **mpirun** must not be executed directly from the command line, but instead must be used in a scheduler submission script.

Hence, unlike batch parallel jobs, the scheduler does not directly dispatch all of the processes in an MPI job. Instead, the scheduler dispatches a single **mpirun** command, and the MPI system takes care of dispatching and managing all of the MPI processes that comprise the job.

Since the scheduler is only dispatching one process, but the MPI job may dispatch others, we must add one more item to the submit script to inform the scheduler how many processes MPI will dispatch. In PBS, this is done using a *resource specification*, which consists of a `-l` followed by one or more resource names and values.

```
#!/bin/sh

#PBS -N MPI-Example

# Use 48 cores for this job
#PBS -l procs=48

mpirun mpi-program
```

When running MPI jobs, it is often desirable to have as many processes as possible running on the same node. Message passing is generally faster between processes on the same node than between processes on different nodes, because messages passed within the same node need not cross the network. If you have a very fast network such as Infiniband or 10 gigabit Ethernet, the difference may be marginal, but on more ordinary networks such as gigabit Ethernet, the difference can be enormous.

### Environment Variables

PBS sets a number of environment variables when a job is started. These variables can be used in the submission script and within other scripts or programs executed as part of the job.

One of the most important is the `PBS_O_WORKDIR` variable. By default, PBS runs processes on the compute nodes with the home directory as the current working directory. Most jobs, however, are run from a project directory under or outside the home directory that is shared by all the nodes in the cluster. Usually, the processes on the compute nodes should all run in the same project directory. In order to ensure this, we can either use the -d flag, or add this command to the script before the other command(s):

```
cd $PBS_O_WORKDIR
```

PBS_O_WORKDIR is set by the scheduler to the directory from which the job is submitted. Note that this is the directory must be shared by the submit node and all compute nodes via the same pathname: The scheduler takes the pathname from the submit node and then attempts to **cd** to it on the compute node(s).

Using this variable is more convenient than using the -d flag, since we would have to specify a hard-coded path following -d. If we move the project directory, scripts using -d would have to be modified, while those using **cd $PBS_O_WORKDIR** will work from any starting directory.

The PBS_JOBNAME variable can be useful for generating output filenames within a program, among other things.

The PBS_ARRAYID variable is especially useful in jobs arrays, where each job in the array must use a different input file and/or generate a different output file. This scenario is especially common in Monte Carlo experiments and parameter sweeps, where many instances of the same program are run using a variety of inputs.

```
#!/usr/bin/env bash

#PBS -t 1-100

cd $PBS_O_WORKDIR
./myprog input-$PBS_ARRAYID.txt
```

### A Submit Script Template

Below is a sample submit script with typical options and extensive comments. This template is also available as a macro in APE (Another Programmer's Editor) and in /share1/Examples on Peregrine.

```
#!/bin/sh

###########################################################################
#   Torque (PBS) job submission script template
#   "#PBS" denotes PBS command-line flags
#   "##PBS" is ignored by torque/PBS
###########################################################################

############################################################################
# Job name that will be displayed by qstat, used in output filenames, etc.

#PBS -N pbs-template

###########################################################################
# Job arrays run the same program independently on multiple cores.
# Each process is treated as a separate job by PBS.
# Each job has the name pbs-template[index], where index is
# one of the integer values following -t.  The entire array can
# also be treated as a single job with the name "pbs-template[]".

##PBS -t 1-10       # 10 jobs with consecutive indexes
##PBS -t 2,4,5,6    # Explicitly list arbitrary indexes

##################################################
# Specifying cores and distribution across nodes

# Arbitrary cores: the most flexible method.  Use this for all jobs with
# no extraordinary requirements (e.g. high memory/process).  It gives the
# scheduler the maximum flexibility in dispatching the job, which could
# allow it to start sooner.

##PBS -l procs=6

# Specific number of cores/node.  Use this for high-memory processes, or
# any other time there is a reason to distribute processes across multiple
# nodes.
#
# For multicore jobs (e.g. MPI), this requirement is applied once to the
# entire job.  To spread an MPI job out so that there is only one process
# per node, use:
#
# nodes=8:ppn=1
#
# For job arrays, it is applied to each job in the array individually.
# E.g.
#
# To spread a job array across multiple nodes, so that there is only one
# process per node, use:
#
# nodes=1:ppn=N
#
# where N is the total number of cores on a node.  This reserves an entire
# node for each job in the array, i.e. other jobs will not be able to use
# any cores on that node.  Useful for high-memory jobs that need all the
# memory on each node.

##PBS -l nodes=3:ppn=1

###########################################################################
# Specifying virtual memory requirements for each process within the job.
# This should be done for all jobs in order to maximize utilization of
# cluster resources.  The scheduler will assume a small memory limit
# unless told otherwise.

##PBS -l pvmem=250mb

######################################################################
# CPU time and wall time.  These should be specified for all jobs.
# Estimate how long your job should take, and specify 1.5 to 2 times
# as much CPU and wall time as a limit.  This is only to prevent
# programs with bugs from occupying resources longer than they should.
# The scheduler will assume a small memory limit unless told otherwise.
#
# cput refers to total CPU time used by all processes in the job.
# walltime is actual elapsed time.
# Hence, cput ~= walltime * # of processes

##PBS -l cput=seconds or [[HH:]MM:]SS
##PBS -l walltime=seconds or [[HH:]MM:]SS

#########################################################################
# Environment variables set by by the scheduler.  Use these in the
# commands below to set parameters for array jobs, control the names
# of output files, etc.
#
# PBS_O_HOST    the name of the host where the qsub command was run
# PBS_JOBID     the job identifier assigned to the job by the batch system
# PBS_JOBNAME   the job name supplied by the user.
# PBS_O_WORKDIR the absolute path of the current working directory of the
#               qsub command
# PBS_NODEFILE  name of file containing compute nodes
```

```
##########################################################################
# Shell commands
##########################################################################

# Torque starts from the home directory on each node by default, so we
# must manually cd to the working directory to ensure that output files
# end up in the project directory, etc.

cd $PBS_O_WORKDIR

# Optional preparation example:
# Remove old files, alter PATH, etc. before starting the main process

# rm output*.txt
# PATH=/usr/local/mpi/openmpi/bin:${PATH}
# export PATH

# MPI job example:
# mpirun ./pbs-template

# Serial or array job example:
# ./pbs-template -o output-$PBS_JOBID.txt
```

### 10.1.5 Terminating a Job

If you determine that a job is not behaving properly (by reviewing partial output, for example), you can terminate it using **qdel**, which take a job ID as a command line argument.

```
peregrine: qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
53.peregrine               MPI-Benchmark    bacon                  0 R batch
peregrine: qdel 53
peregrine: qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
53.peregrine               MPI-Benchmark    bacon           00:04:20 C batch
```

### 10.1.6 Viewing Output of Active Jobs

As mentioned in Section 10.1.4, the output sent by processes to standard output and standard error is redirected to files, as named by the -o and -e flags.

However, these files are not created until the job ends. Output is stored in temporary files until then.

A convenience script, called **qpeek**, is provided for viewing the output of a job stored in the temporary files. The **qpeek** is not part of PBS, but is provided as an add-on. It takes a single job ID as an argument.

```
FreeBSD peregrine bacon ~ 499: qsub solveit.pbs
297.peregrine.hpc.uwm.edu
FreeBSD peregrine bacon ~ 500: qpeek 297
Computing the solution...
1 2 3 4 5
FreeBSD peregrine bacon ~ 501:
```

### 10.1.7 Self-test

1. What is the PBS command for showing the current state of all nodes in a cluster?
2. What is the PBS command to show the currently running jobs on a cluster?
3. Write and submit a batch-serial PBS script called `list-etc.pbs` that prints the host name of the compute node on which it runs and a long-listing of `/etc` directory on that node.

   The script should store the output of the commands in `list-etc.stdout` and error messages in `list-etc.stderr` in the directory from which the script was submitted.

   The job should appear in **qstat** listings under the name "list-etc".

   Quickly check the status of your job after submitting it.
4. Copy your `list-etc.pbs` script to `list-etc-parallel.pbs`, and modify it so that it runs the **hostname** and **ls** commands on 10 cores instead of just one.

   The job should produce a separate output file for each process named `list-etc-parallel.o<jobid>-<arrayid>` and a separate error file for each process named `list-etc-parallel.e<jobid>-<arrayid>`.

   Quickly check the status of your job after submitting it.
5. What is the PBS command for terminating a job with job-id 3545?
6. What is the PBS command for viewing the terminal output of a job with job-id 3545 while it is still running?
7. What is the PBS command for showing detailed job information about the job with job-id 3254?

# Chapter 11

# Job Scheduling with LSF

**Before You Begin**
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 3), the Unix shell (Section 3.3.3, redirection (Section 3.13), and shell scripting (Chapter 4).

## 11.1 The LSF Scheduler

*LSF*, which stands for  Load Sharing Facility, is part of a commercial cluster management suite called Platform Cluster Management. The LSF scheduler allows you to schedule jobs for immediate queuing or to run at a later time. It also allows you to monitor the progress of your jobs and manipulate them after they have started. This document describes the basics of using LSF, and how to get more detailed information from the official LSF documentation.

### 11.1.1 Submitting Jobs

Jobs are submitted to the LSF scheduler using the **bsub** command. The **bsub** command finds available cores within the cluster and executes your job(s) on the selected cores. If the resources required by your job are not immediately available, the scheduler will hold your job in a queue until they become available.

You can provide the Unix command you want to schedule as part of the **bsub** command, but the preferred method of using **bsub** involves creating a simple shell script. Several examples are given below. Writing shells scripts is covered in Chapter 4.

#### Using bsub without a script

```
[user@hd1 ~]$ bsub -J hostname_example -o hostname_output%J.txt hostname
```

The **bsub** command above finds an available core and dispatches the Unix command **hostname** to that core. The output of the command (the host name of the node containing the core) is appended to `hostname_output%J.txt`, where %J is replaced by the job number. The `-o` flag in the command tells **bsub** that any output the **hostname** command tries to send to the terminal should be appended to the filename following `-o` instead.

The job is given the name `hostname_example` via the `-J` flag. This job name can be used by other LSF commands to refer to the job while it is running. A few of these commands are described below.

#### Using a Submission Script

The job in the example above can also be submitted using a shell script.

When used with the scheduler, scripts document exactly how the **bsub** command is invoked, as well as the exact sequence of Unix commands to be executed on the cluster. This allows you to re-execute the job in exactly the same manner without having to remember the details of the command. It also allows you to perform preparation steps within the script before the primary command, like removing old files, going to a specific directory, etc.

To script the example above, enter the appropriate Unix commands into a text file, e.g. `hostname.bsub`, using your favorite text editor.

```
[user@hd1 ~]$ nano hostname.bsub
```

Once you're in the editor, enter the following text, and then save and exit:

**Example 11.1** Simple LSF Batch Script

```
#!/usr/bin/env bash

#BSUB -J hostname_example -o hostname_output%J.txt
#BSUB -v 1000000

hostname
```

Then submit the job by running:

```
[user@hd1 ~]$ bsub < hostname.bsub
```

Note that the script is fed to **bsub** using input redirection.

The first line of the script:

```
#!/usr/bin/env bash
```

indicates that this script should be executed by bash (Bourne Again shell).

The second and third lines:

```
#BSUB -J hostname_example -o hostname_output%J.txt
#BSUB -v 1000000
```

specify command line flags used to be used with the **bsub** command. Note that they're the same flags we entered on the Unix command line when running **bsub** without a script. You can place any number of command line flags on each #BSUB line. It makes no difference to **bsub**, so it's strictly a matter of readability and personal taste.

Recall that to the shell, anything following a '#' is a comment. Lines that begin with '#BSUB' are specially formatted shell comments that are recognized by **bsub** but ignored by the shell running the script.

The beginning of the line must be *exactly* '#BSUB' in order to be recognized by **bsub**. Hence, we can disable an option without removing it from the script by simply adding another '#':

```
##BSUB -o hostname
```

The new command-line flag here, `-v 1000000`, limits the memory use of the job to 1,000,000 kilobytes, or 1 gigabyte.

**Caution** All jobs should use memory-limits like this to prevent them from accidentally overloading the cluster. Programs that use too much memory can cause compute nodes to crash, which may kill other users' jobs as well as your own. It's possible to cause another user to lose several weeks worth of work by simply miscalculating how much memory your job requires.

All lines in the script that are not comments are interpreted by bash as Unix commands, and are executed on the core(s) selected by the scheduler.

### 11.1.2 Job Types

Scheduled jobs fall under one of several classifications. The differences between these classifications are mainly conceptual and the differences in the commands for submitting them via **bsub** can be subtle.

#### Batch Serial Jobs

The example above is what we call a batch serial job. Batch serial jobs run on a single core, and any output they send to the standard output (normally the terminal) is redirected to the file named using `-o filename` with **bsub**.

Batch jobs do not display output to the terminal window, and cannot receive input from the keyboard.

#### Interactive Jobs

If you need to see the output of your job on the screen during execution, or provide input from the keyboard, then you need an interactive job.

From the user's point of view, an interactive job runs as if you had simple entered the command at the Unix prompt instead of running it under the scheduler. The important distinction is that with a scheduled interactive job, the scheduler decides which core to run it on. This prevents multiple users from running interactive jobs on the same node and potentially swamping the resources of that node.

From the scheduler's point of view, an interactive job is almost the same as a batch serial, except that the output is not redirected to a file, and interactive jobs can receive input from the keyboard.

To run an interactive job in **bsub**, simply add `-I` (capital i), `-Ip`, or `-Is`. The `-I` flag alone allows the job to send output back from the compute node to your screen. It does not, however, allow certain interactive features required by editors and other full-screen programs. The `-Ip` flag creates a pseudo-terminal, which enables terminal features required for most programs. The `-Is` flag enables shell mode support in addition to creating a pseudo-terminal. `-Ip` should be sufficient for most purposes.

**Example 11.2** Batch Interactive Script

```
#!/usr/bin/env bash

#BSUB -J hostname_example -Ip

hostname
```

Note that the `-o` flag cannot be used with interactive jobs, since output is sent to the terminal screen, not to a file.

#### Waiting for Jobs to Complete

The **bsub** `-K` flag causes **bsub** to wait until the job completes.

This provides a very simple mechanism to run a series of jobs, where one job requires the output of another.

It can also be used in place of interactive jobs for tasks such as program compilation, if you want to save the output for future reference.

#### Batch Parallel Jobs

Batch serial and interactive jobs don't really make good use of a cluster, since they only run your job a single core. In order to utilize the real power of a cluster, we need to run multiple processes in parallel (at the same time).

If your computations can be decomposed into N completely independent tasks, then you may be able to use a batch parallel job to reduce the run time by nearly a factor of N. This is the simplest form of parallelism, and it maximizes the performance gain on a cluster. This type of parallel computing is often referred to as *embarrassingly parallel*, *loosely coupled*, *high throughput* *computing* (HTC), or *grid computing*. It is considered distinct from *tightly coupled* or *high performance computing (HPC)*, where the processes that make up a parallel job communicate and cooperate with each other to complete a task.

In LSF, batch parallel jobs are distinguished from batch serial jobs in a rather subtle way; by simply appending a job index specification to the job name:

**Example 11.3** Batch Parallel Script

```
#!/usr/bin/env bash

#BSUB -J parallel[1-10] -o parallel_output%J.txt
printf "Job $LSB_JOBINDEX running on `hostname`\n" > output_$LSB_JOBINDEX. ↩
    txt
```

This script instructs the scheduler to allocate 10 cores in the cluster, and start a job consisting of 10 simultaneous processes with names parallel[1], parallel[2], ... parallel[10] within the LSF scheduler. The scheduler allocates 10 cores, and the commands in the script are executed on all 10 cores at (approximately) the same time.

The environment variable $LSB_JOBINDEX is created by the scheduler, and assigned a different value for each process within the job. Specifically, it will be a number between 1 and 10, since these are the subscripts specified with the `-J` flag. This allows the script, as well as commands executed by the script, to distinguish themselves from other processes. This can be useful when you want all the processes to read different input files or store output in separate output files.

Note that the output of the printf command above is redirected to a different file for each job index. Normally, printf displays its output in the terminal window, but this example uses a shell feature called output redirection to send it to a file instead. When the shell sees a ">" in the command, it takes the string after the ">" as a filename, and causes the command to send its output to that file instead of the terminal screen.

After all the processes complete, you will have a series out output files called output_1.txt, output_2.txt, and so on. Although each process in the job runs on a different core, all of the cores have direct access to the same files and directories, so all of these files will be in the same place when the job finishes.

Each file will contain the host name on which the process ran. Note that since each node in the cluster has multiple cores, some of the files may contain the same host name. Remember that the scheduler allocates cores, not nodes.

### MPI Jobs

As mentioned earlier, if you can decompose your computations into a set of completely independent parallel processes that have no need to talk to each other, then a batch parallel job is probably the best solution.

When the processes within a parallel job must communicate extensively, special programming is required to make the processes exchange information.

MPI (Message Passing Interface) is the de facto standard library and API (Application Programming Interface) for such tightly-coupled distributed programming. MPI can be used with general-purpose languages such as C, Fortran, and C++ to implement complex parallel programs requiring extensive communication between processes. Parallel programming can be very complex, but MPI will make the program implementation as easy as it can be.

There are multiple implementations of the MPI standard, several of which are installed on the cluster. The OpenMPI implementation is the newest, most complete, and is becoming the standard implementation. However, some specific applications still depend on other MPI implementations, so many clusters have multiple implementation installed. If you are building and running your own MPI applications, you may need to select a default implementation using the mpi-selector-menu command. A default selection of openmpi_gcc_qlc is usually configured for new users, but you can change it at any time, and as often as you like.

All MPI jobs are started by using the **mpirun** command, which is part of the MPI installation. The **mpirun** command dispatches the mpi program to all the nodes that you specify with the command, and sets up the communication interface that allows them to pass messages to each other while running.

**Caution** On a cluster shared by many users, it is impractical to use **mpirun** directly, since it requires the user to specify which cores to use, and cores should be selected by the scheduler instead. All MPI jobs should use the wrappers described below instead of using **mpirun** directly.

To facilitate the use of **mpirun** under the LSF scheduler, LSF provides *wrapper* commands for each MPI implementation. For example, to use OpenMPI, the command is **openmpi_wrapper**. These wrappers allow the scheduler to pass information such as the list of allocated cores down to **mpirun**.

**Example 11.4** LSF and OpenMPI

```
#!/usr/bin/env bash

#BSUB -o matmult_output%J.txt -n 10
openmpi_wrapper matmult
```

In this script, **matmult** is an MPI program, i.e. a program using the MPI library functions to pass messages between multiple cooperating matmult processes.

The scheduler executes **openmpi_wrapper** on a single core, and openmpi_wrapper in turn executes matmult on 10 cores.

Note that MPI jobs in LSF are essentially batch serial jobs where the command is **openmpi_wrapper** or one of the other wrappers. The scheduler only allocates multiple cores, and executes the provided command on one of them. From there, MPI takes over.

Although openmpi_wrapper executes on only one core, the MPI program as a whole requires multiple cores, and these cores must be allocated by the scheduler. The **bsub** flag `-n 10` instructs the LSF scheduler to allocate 10 cores for this job, even though the scheduler will only dispatch openmpi_wrapper to one of them. The openmpi_wrapper command then dispatches **matmult** to all of the cores provided by the scheduler, including the one on which openmpi_wrapper is running.

Note that you do *not* need a cluster to develop and run MPI applications. You can develop and test MPI programs on your PC provided that you have a compiler and an MPI package installed. The speedup provided by MPI will be limited by the number of cores your PC has, but the program should work basically the same way on your PC as it does on a cluster, except that you will probably use **mpirun** directly on your PC rather than submit the job to a scheduler.

### 11.1.3 Job Control and Monitoring

#### Listing jobs

After submitting jobs with **bsub**, you can monitor their status with the bjobs command.

```
[user@hd1 ~]$ bjobs
```

By default, bjobs limits the amount of information displayed to fit an 80-column terminal. To ensure complete information is printed for each job, add the `-w` (wide output) flag.

```
[user@hd1 ~]$ bjobs -w
```

By default, the bjobs command will list jobs you have running, waiting to run, or suspended. It displays basic information such as the numeric job ID, job name, nodes in use, and so on.

If you would like to see what other users are running on the cluster, run

```
[user@hd1 ~]$ bjobs -w -u all
```

There are many options for listing a subset of your jobs, other users' jobs, etc. Run **man bjobs** for a quick reference.

#### Checking Progress

When you run a batch job, the output file may not be available until the job finishes. You can view the output generated so far by an unfinished job using the bpeek command:

```
[user@hd1 ~]$ bpeek numeric-job-id
```

or

```
[user@hd1 ~]$ bpeek -J job-name
```

The job-name above is the same name you specified with the `-J` flag in **bsub**. The numeric job id can be found by running bjobs.

#### Using top

Using the output from **bjobs**, we can see which compute nodes are being used by a job.

We can then examine the processes on a given node using a remotely executed **top** command:

```
avi: ssh -t compute-1-03 top
```

---

**Note** The `-t` flag is important here, since it tells ssh to open a connection with full terminal control, which is needed by **top** to update your terminal screen.

---

#### Terminating Jobs

A job can be terminated using the bkill command:

```
[user@hd1 ~]$ bkill numeric-job-id
```

or

```
[user@hd1 ~]$ bkill -J job-name
```

Again, the numeric job id can be found by running bjobs, and the job-name is what you specified with the `-J` option in **bsub**.

#### Email Notification

LSF has the ability to notify you by email when a job begins or ends. All notification emails are sent to your PantherLINK account.

To receive an email notification when your job ends, add the following to your **bsub** script:

```
#BSUB -N
```

To receive an email notification when your job begins, add the following to your **bsub** script:

```
#BSUB -B
```

The -B flags is only useful when the load in the cluster is too high to accommodate your job at the time it is submitted.

## 11.2 Good Neighbor Scheduling Strategies

### 11.2.1 Job Distribution

Modern clusters consist of nodes with many cores.This raises the question of how processes should be distributed across nodes.

If independent processes such as serial jobs and job arrays are spread out across as many nodes as possible, there may be less resource contention within each node, leading to better overall performance.Spreading jobs out this way can also be a necessity.For example, if each process in a job array requires 20 gigabytes of RAM, and each node has 24 gigabytes available, then we can 't run more than one of these processes per node.

On the other hand, shared memory jobs require their processes to be on the same node, and multicore jobs(such as MPI jobs) that use a lot of interprocess communication will perform better when their processes are on the same node. (Communication between any two processes is generally faster if they are on the same node.)

When large numbers of serial and job array processes are spread out across the cluster, shared memory jobs may be prevented from starting, because there are no individual nodes with sufficient free cores available, even though there are plenty of free cores across the cluster.Multicore jobs may see degraded performance in this situation, since their processes are forced to run on different nodes and therefore must use the network to communicate.

Default scheduler policies tend to favor spreading out serial and array jobs across many nodes, and clumping multicore jobs on the same node.

Users running serial and array jobs can take an extra step toward being a good citizen by overriding the default scheduler policies. If you know that your serial or array jobs won 't have a problem with resource contention, you can tell the scheduler to dispatch as many of them as possible to heavily loaded nodes using a simple **bsub** resource specifier:

```
    #BSUB -R "order[-ut]"
```

This will leave more nodes with a large number of cores available, which will help users running shared memory and multicore jobs.

## 11.3 More Information

For details of any of the LSF commands or any other Unix, run **man command** on the cluster. For example:

```
        [user@hd1 ~]$ man bsub
```

This will display the man page for the **bsub** command. Most other Unix commands also have man pages. Note that the man pages are meant to provide a quick reference to users who already have a basic understanding of the command. Beginners looking for a more complete explanation of a topic may be better off finding a good book or tutorial.

The LSF User's Guide is a more complete reference for the LSF scheduler.

The HPC Community website is a gathering place for LSF users. It offers many documents for download as well as a user forum.

Links to the LSF User's Guide, the HPC Community website, and more can be found at the cluster user documentation page: http://www4.uwm.edu/hpc/support_resources.

# Part III

# High Performance Programming

# Chapter 12

# Computer Hardware

## 12.1 Why Learn about Hardware?

A basic understanding of computer hardware is essential to understand what computer programs actually do, and how to make them efficient and correct.

The sections below will provide some understanding of each of the major components of a computer, to help you make better decisions as a programmer.

## 12.2 Central Processing Unit

The *Central Processing Unit*, or *CPU* directs the activities of all other hardware. The CPU is in turn directed by programs, which are sequences of instructions. Put simply, a CPU converts programs into actions.

Modern computers usually have more than one CPU, so they are capable of running multiple processes at the same time. The term CPU has become somewhat ambiguous, because a single CPU chip often contains multiple independent CPU circuits. Hence, the term CPU sometimes means the whole CPU chip, and sometimes means one of the CPU circuits within the chip. To avoid confusion, most people now use the term *core* to refer to one of multiple CPU circuits within a chip.

## 12.3 Non-volatile Auxiliary Storage: Disk, Tape and Flash

*Non-volatile* storage is storage that retains its contents while the power is off. This includes magnetic disks (hard disks), optical discs such as CD, DVD, and Blu-Ray, Flash devices such as USB thumb drives and SSDs, and magnetic tape.

Non-volatile storage is mainly used to store files containing programs or data, for use at a later time. It can also be used for virtually any other storage needs, however, as we will see shortly.

## 12.4 Electronic Memory

### 12.4.1 RAM and ROM

Electronic memory is categorized into two major types:

- *RAM*, or *Random Access Memory*, is both readable and writable. The acronym RAM comes from the fact that we can access any part of RAM instantly, as opposed to a tape device which can only be accessed sequentially. I.e., to read something from the middle of a tape, we must read everything before it in order to find it. RAM is a poor acronym, however, because it is generally used to refer to read-write memory as opposed to read-only memory, which is also randomly accessible. RWM would be a better initialism than RAM.

RAM is *volatile*, which means it requires power in order to retain its contents. The term volatile is borrowed from chemistry, where it refers to something that evaporates.

Computer programs and data are normally stored in disk files when they are not in use. When a program is run (executed), the program and some of the data it manipulates are loaded from disk into RAM, which is about 1,000,000 times faster than disk. This greatly improves speed when the same data is accessed repeatedly.

- *ROM*, or *Read-Only Memory* is not generally writable. The original ROMs were set to a specific content at the factory and could never be changed again.

  Today, we use *EEPROM*, or *electronically erasable programmable ROM*, such as Flash memory. EEPROMS are writable, but not as easily as RAM. There are special procedures to altering the contents of a EEPROM. The important feature of ROM and EEPROM is that it is non-volatile, so it retains its content even when the power is cut.

  Non-volatile memory such as EEPROM is used to store *firmware*, which is essentially software that stays in memory when the power is off. Firmware makes it possible for computers to start when the power is turned on (*cold boot*), and allows small and embedded devices which are often powered down to function. The boot sequence cannot be started from a program on a disk, since reading a program from disk requires a program! Hence, there must be a minimal amount of program code in memory when the power comes on to start the boot sequence. In a personal computer, the core firmware, sometimes called *BIOS* (*Basic Input/Output System* or *Built-In Operating System*), initializes the hardware and loads the first part of the operating system from disk into RAM. From there, the operating system takes over.

### 12.4.2 Memory Structure

Memory is a one-dimensional array of storage cells. In virtually all computers today, each cell holds one byte. Memory cells are selected using an integer index called the *memory address*. Memory addresses begin at zero.

| **Address** | **Content** |
|---|---|
| 0 | 01001010 (decimal 74, character '<', or anything else!) |
| 1 | 01010010 |
| 2 | 11010101 |
| 3 | 00001011 |
| ... | ... |
| Memory size - 1 | 10100110 |

Table 12.1: Example Memory Map

Each 8-bit cell can hold a character such as 'a' or '?', an integer between 0 and 255, or part of a larger piece of information such as a 64-bit floating point number, which would occupy eight consecutive cells.

### 12.4.3 The Memory Hierarchy

Memory is actually multilayered in what we call the *memory hierarchy*, depicted in Table 12.2.

*Cache*, which is French for "hidden", is a small, fast memory, where the hardware places copies of data read from larger, slower memory levels. Since programs tend to access the same small sections of memory repeatedly for a while (due to program loops), the CPU can usually get the data it needs from the cache. Cache is "hidden" because programs are unaware of it. They ask for data from a DRAM address, and much of the time the hardware gives them the cached copy much more quickly.

Virtual memory uses disk to extend the amount of RAM that is apparently available to programs. When a computer using virtual memory runs out of RAM, it *swaps* blocks of data from RAM out to a special area of disk known as *swap space*.

Since disk is typically 1,000 to 1,000,000 times slower than RAM, swapping is very expensive. A single swap to or from disk usually takes only a few milliseconds, but this is far longer than RAM access, which is on the order of nanoseconds, so if a program causes many swaps, it can become unbearably slow. Swap space is therefore useful mainly for inactive programs such as word processors, which spend most of their time waiting for user input. Parts of the program or data that are not actively in

| Name | Technology | Typical size | Access time |
|---|---|---|---|
| Registers | Static RAM | 256 bytes (32 words) | 1 clock cycle |
| Level 1 cache | Static RAM | 4 MiB | 1 to a few clock cycles |
| Level 2 cache | Static RAM | 16 MiB | A few clock cycles |
| Level 3 cache | Static RAM | 256 MiB | Several clock cycles |
| Main memory | Dynamic RAM | 16 GiB | Dozens of clock cycles |
| Solid State Drive | Flash RAM | 1 terabyte | Tens of microseconds |
| Magnetic disk | Platters and moving heads | 4 terabytes | A few milliseconds |
| Magnetic tape | Reel to reel tape | Many terabytes | Seconds to hours |

Table 12.2: The Memory Hierarchy

use can be swapped out to disk to make room for more active programs. If it takes a fraction of a second to swap something back into RAM when the user presses a key or clicks an icon, the user won't generally notice.

For very active programs, swap is of little use. Programs that actively use more memory than the system has available as RAM may spend most of their time waiting for swap operations. When this happens, the system is said to be *thrashing*, like someone struggling to stay afloat, but not moving forward through the water. Therefore, it is important for computationally intensive programs to use as little memory as possible.

So, memory, from a program's perspective, consists of everything from level 1 cache to swap space. The hardware stores as much as it can in the level 1 cache. When it is full, it must use the level 2 cache. And so on all the way down to swap space. Hence, the less memory a program uses, the faster memory access will be. If you can reduce memory use to the point where the machine code and data all fit into the level 1 cache, the program will run significantly faster than the same program frequently accessing DRAM.

## 12.5 Computer Word Size

What is a "64-bit" computer? The word size of a computer generally indicates the largest integer it can process in a single instruction, and the size of a memory address, which is usually, but not necessarily the same as the integer size.

The main indication of the word size is how much memory the processor can address. The maximum amount of memory a processor is capable of accessing is known as the *address space*.

A processor with a 32-bit address, such as an Intel Pentium or AMD Athlon, can only represent $2^{32}$ distinct memory addresses. Hence, if each address contains one byte, then the computer can have at most $2^{32}$ bytes, or 4 gibibytes (a little more than 4 billion bytes) of memory. This is more than enough RAM for most purposes, but some applications can benefit from using much more, so 64-bit processors and cheap RAM are blessings to some types of computing.

**Note** Note that 32-bit processors such as the 486, Pentium, and PowerPC G4 have supported 64-bit floating point numbers for a long time, but were still regarded as 32-bit processors due to their 32-bit integers and addresses.

An AMD64 processor, also known as x86_64, has a 64-bit memory address, and uses byte-addressable memory (one byte of memory per address). Hence, the memory address space is $2^{64}$ bytes (16 exbibytes, or roughly 16 billion gibibytes). Recall that a gibibyte is $2^{30}$ bytes, or a little more than a gigabyte ($10^9$). This is far more memory than any single computer is ever likely to have. Most PCs have between a few gibibytes and a few hundred gibibytes.

## 12.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What does CPU stand for and what does it do?
2. What is non-volatile auxiliary storage? What is it used for?
3. What is RAM? What is it used for and why?
4. What is ROM? What is it used for and why?
5. How is RAM/ROM structured on most computers?
6. What limits the size of a computer’s address space? How is this related to the actual amount of memory in a computer?
7. How is cache used and why?
8. What does virtual memory do?
9. What type of program is virtual memory good for and what type is it not good for? Why?

# Chapter 13

# Software Development

In this chapter, we introduce some high-level concepts behind high-performance scientific programming, to lay a foundation for the programming instruction that follows. Before we can learn to write high-quality software, we first must understand what makes software high-quality.

## 13.1 Goals of a Top-notch Scientific Software Developer

### 13.1.1 Maximize Portability

If possible, use an open language such as C, C++, Fortran, Octave, Perl, Python, or R, rather than proprietary languages such as MATLAB or SPSS, which only run on a few specific operating systems and CPUs (central processing units, the part of the computer that a program directly controls).

If possible, write code following Unix standards, so people will be able to run your programs on virtually any operating system including BSD, Linux, macOS, and even Windows with Cygwin or WSL. If you code for Windows or using Apple's proprietary Xcode environment, then your programs will only run on that platform.

As discussed in Chapter 3, most research computing is done on Unix-compatible operating systems. Every operating system you are likely to use, with the exception of Microsoft Windows, is Unix-compatible. There are many commercial Unix systems used in corporate data centers, but researchers are most likely to be running Mac OS X or a free Unix system such as one of the many BSD or Linux distributions.

Windows users can run Unix software using a compatibility layer such as Cygwin (Section 3.4.1), or by running a Unix system on a virtual machine (Chapter 40) such as WSL or VirtualBox. Preconfigured virtual machine installations are available for many Unix systems, so obtaining a Unix environment on your Windows PC is easy.

Code developed in proprietary Windows development environments such as Visual Studio is usually difficult to port to Unix systems. This is often a major problem for researchers who discover down the road that their PC is not fast enough and want to run their code on high-power Unix servers or clusters. The only option for them is to heavily modify or rewrite the code so that it is Unix-compatible.

Although Apple's Mac OS X is Unix-compatible, the Xcode development environment is proprietary, and projects developed in Xcode are not portable to other Unix systems. If you develop using an Xcode project, you will need to maintain a separate build system for other Unix platforms. Alternatively, you can develop under Mac OS X using a single, portable build system such as a simple Makefile that will work in all Unix environments (including Cygwin).

For most researchers, it makes no difference which operating system they use to develop and test their code. Unix systems are highly compatible with each other and most of the features of any one of them are available in the others. Each system does have some of its own special features, but most of them are not relevant to most scientific software, so it is generally easy to write code that is portable among all Unix systems.

### 13.1.2 WORF: Write Once, Run Forever

Unfortunately, much of the software used in science is disposable scripts that don't perform well, are not robust (don't handle bad input), and are reinvented many times over by other researchers doing similar work. This will always be part of the field, as most scientists lack the time and/or the skills to do otherwise.

What the community needs in order to move forward is high-quality application programs that will serve people for many years. If a few more scientists decide to develop such applications, it will bring about a huge reduction in duplicated effort for a long time to come.

If you choose to contribute to the collection of quality software, you'll need to consider carefully how to develop it. You may not have much time to maintain and update the code a few years from now, so write code that will require as little maintenance as possible.

Code in a stable language (one that is not changing), such as C or POSIX Bourne shell. Test on multiple platforms, such as BSD, Cygwin, Linux, and Mac. This is easy to do by running free, open source systems in virtual machines. The fact that a program works on your development platform does not mean it is free of bugs. It probably contains many bugs that simply are not visible, since they don't cause incorrect output or program crashes. Testing on another platform will *almost always* reveal bugs that you were not aware of.

Most scientists don't understand that writing software is like adopting a puppy. It's not something you do and then forget about. It's a 10 to 20 year commitment in maintenance and support. Almost all software will need to be upgraded periodically to work with new operating systems, compilers, interpreters, libraries, and other programs. Bugs will continue to be discovered and fixed over the entire life of the software.

*Abandonware* is software that is still available, but no longer maintained. This is the state of most scientific software. The vast majority of scientific software is abandoned within a few years after publication. Much of it is written as someone's thesis and the author either has no time or no interest in maintaining it after graduation.

*Paperware* is a special category of abandonware, written for the sake of publishing a paper, and then immediately abandoned. Many researchers work in a *publish or perish*, where they must continually get papers published in respected journals in order to advance, or even maintain, their career. It is rarely feasible for people in this environment to maintain software after the associated paper is published, since they must focus on publishing the next study.

Abandoned software may still be useful, but more often than not, it cannot be installed on newer systems because it was not written for portability. It may have been written in Python 2 and is not compatible with Python 3. Notable exceptions are POSIX Bourne shell and C, which have changed very little in recent years and are unlikely to change much in the future. Software written in these languages will likely continue to function with little or no modification indefinitely.

Complex build systems are also problematic in that they change over time, often breaking compatibility with older versions. A simple makefile is more likely to work for people 10 years from now than a complex **cmake** setup. Writing makefiles is covered in Chapter 21.

### 13.1.3 Minimize CPU and Memory Requirements

Many scientists do not have easy access to an HPC cluster or even a powerful workstation. Being able to run your program on a laptop may be the difference between success and failure for you and other researchers. Running highly inefficient programs on an HPC cluster is unwise and unethical. It wastes resources that are actually needed for other work, doing things that could be done on much less expensive hardware. The fastest languages by far, as we will see in Section 13.4, are C, C++, and Fortran.

### 13.1.4 Minimize Deployment Effort

Make the program and build-system package-friendly and discourage the use of caveman installs. If it is easy to add to existing package managers, then users won't waste their time struggling with caveman installs and won't waste your time with problem reports and help requests. More on this in Chapter 21 and Chapter 31.

### 13.1.5 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do we maximize portability of the software we write?
2. Why is it important to minimize the future maintenance requirements of the free software we write?
3. How do we minimize future maintenance of our software?
4. What is abandonware? How common is it in scientific computing?
5. What are the benefits of minimizing CPU requirements?
6. How do we help users avoid problems with deploying our software?

## 13.2 You are the Cause of your own Suffering

As stated earlier in this text, most people make most things far more complicated than they need to be. Nowhere will you see this fact documented better than in a typical computer program. Most computer programs could be reduced to a fraction of their present size, made to run orders of magnitude faster, and made far easier to read and maintain. The majority of complexity in most code is due to bad decisions on the part of the programmer.

What this means for you as you begin to learn programming, is that the experience can be much simpler and more enjoyable than the typical case. You may have heard from other students that their computer programming class was really tough (often paraphrased in more colorful language). If so, their struggle was mostly a result of following bad practices, due either to poor teaching or failure to follow their teacher's advice.

Pain is *not* inherent in the programming process. If you are taught properly and have the basic self-discipline to follow those teachings, programming can be a fun creative process with minimal struggle. It will often be time-consuming, but time flies when you're having fun, which in software development typically means making steady progress. It ceases to be fun when you're stuck on a problem for a long time, but this will happen very rarely if you take an intelligent approach to the programming process. If you find yourself getting stuck often, you need to re-evaluate your coding procedures.

Depth before breadth. Take your time learning to program and PRACTICE the basics until you understand them well. Master variables before learning types. Master types before learning conditionals. Master conditionals before learning loops. Master loops before learning subprograms. Master subprograms before learning arrays. And so on... A little time invested early in the learning process will save you a lot of wasted time and frustration later.

This text focuses on good practices from the beginning and leads you down the right path so that your experience as a programming requires minimal effort and produces solid results: Simple, fast, readable, easily maintainable program code. We'll begin with some important background knowledge and then learn to program one step at a time, exploring relevant best practices and pitfalls along the way.

### 13.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How good is the quality of most existing software? Explain.
2. If you are getting frustrated, struggling to make progress on a program, what should this tell you?

## 13.3 Language Levels

Programming languages are categorized into a few different levels, which are described in the following sections.

### 13.3.1 Machine Language

The lowest level of programming languages are the *machine languages*. A machine language is a set of binary instruction codes which direct the activities of a Central Processing Unit (CPU). Everything a computer does is ultimately the result of running machine language instructions. No matter what language you use to program, the computer is always running a sequence of machine instructions in order to execute your program. The run time of any program is determined by how long that sequence is and to some extent, which instructions it contains. Multiply and divide instructions take longer than add and subtract instructions, for example.

Each instruction consists of an *operation code (opcode for short)* and possibly some operands. For example, an *add* instruction contains a binary opcode that causes the CPU to initiate a sequence of operations to add two numbers, and usually two or three operands that specify where to get the terms to be added, and where to store the result.

The CPU reads instructions from memory, and the bits in the instruction trigger "switches" in the CPU, causing it to *execute* the instruction. For example, the following is an example of a machine code add instruction for the MIPS microprocessor:

```
            00000000010000110001100000100000
```

This instruction would cause the processor to add the contents of registers 2 and 3, and store the result back into register 3. The meaning of each bit in this instruction is depicted in Table 13.1

| **opcode** | **source1** | **source2** | **destination** | **unused** | **opcode continued** |
|---|---|---|---|---|---|
| 000000 | 00010 | 00011 | 00011 | 00000 | 100000 |
| add | register 2 | register 3 | register 3 | - | add |

Table 13.1: Example MIPS Instruction

A CPU's *instruction set architecture* is defined by its instruction set. For example, the Intel x86 family of architectures has a specific set of instructions, with variations that have evolved over time (8086, 8088, 80286, 80386, 80486, Pentium, Xeon, Core Duo, AMD Athlon, etc.) The x86 architectures have a completely different instruction set than the MIPS architecture, the ARM architecture, the RISC-V architecture, and so on.

The fact that machine language is specific to one architecture presents an obvious problem: Machine language is not portable. Machine language programs written for one architecture have to be completely rewritten in order to run on a different architecture.

In addition to the lack of portability, machine language programs tend to be very long, since the machine instructions are primitive. Most machine instructions can only perform a single, simple operation such as adding two numbers. It takes a sequence of dozens of instructions to evaluate a simple polynomial.

### 13.3.2 Assembly Language

In order to program in machine language, one would have to memorize or look up binary codes for opcodes and operands in order to read or write the instructions. This process is far too tedious and error prone to allow for productive (or enjoyable) programming.

One of the first things early programmers did to make the job easier is create a *mnemonic*, or symbolic form of machine language that is easier for people to read. For example, instead of writing

```
            00000000010000110001100000100000
```

a programmer could write

```
            add     $3, $2, $3
```

which is obviously much more intuitive.

Assembly language also makes it possible for the programmer to use named variables instead of numeric memory addresses and many other convenient features. However, the CPU can't understand this mnemonic form, so it has to be translated, or *assembled* into machine language before the computer can run it. Hence, it was given the name *assembly language*.

While assembly language is much easier to read and write than machine language, it still suffers from two major problems:

- It is still specific to one architecture, i.e. it is not portable.
- The instructions are still very primitive, so the programs are long and difficult to follow.

### 13.3.3 High Level Languages (HLLs)

Early programmers yearned for the ability to write a mathematical expression or an English-like statement, and have the equivalent machine code generated automatically. In the 1950's, a team at IBM led by John Backus set out to do just that, and their efforts produced the first widely used high-level language, FORTRAN, which is short for FORmula TRANslator. The program that performed the translation from algebraic expressions and other convenient constructs to machine language was named a *compiler*.

FORTRAN made it much easier to write programs, since we could now write a one-line algebraic expressions and let the compiler convert it to the long sequence of machine instructions. We could write a single *print* statement instead of the hundreds of assembly language instructions needed to perform common tasks like converting a number to the sequence of characters that are sent to our terminal.

In addition to making our programs much shorter and easier to understand, FORTRAN paved the way for another major benefit: *portability*. We could now write programs in FORTRAN, and by modifying the compiler to output machine code for different CPU architectures, we could run the same program on any type of computer.

### 13.3.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is machine language?
2. What is assembly language?
3. What are two disadvantages of machine language and assembly language, compared to high level languages?
4. What are two advantages of high level languages over machine language and assembly language?

## 13.4 Language Selection

### 13.4.1 Decision Factors

Selecting a language can be a difficult and controversial decision. Ultimately, the decision should come down to costs, which may include some or all of the following:

- Hardware
- Software purchases
- Labor (development, testing, maintenance)
- Training
- Operational costs (infrastructure, electricity, etc.)

### 13.4.2 Language or Religion?

In the 1980's, if you weren't using C, you were a heretic. In the 1990's, if you were still using C, you were a heretic. The fervor is often the result of the naive "newer is better" mentality. In recent years, C has been making a comeback, as people have discovered that their cool new languages turned out to be false messiahs. C++ was the rage in application development in the early 1990s, then Java, then Scala, yada, yada. Perl was the rage in server-side web programming until PHP and Ruby-on-rails came along. The saga continues.

There is no shortage of evangelists in the computing arena. Many of them will angrily educate you on why their favorite language or operating system is "better" and why people who use anything else are idiots. Most such arguments take the form of "my language or OS has feature X and the others don't", where "feature X" is something you will probably never need.

Don't pay attention to arguments that are obviously based on a bizarre emotional attachment to a piece of software. Those individuals are less interested in helping you be productive than they are in getting you to validate their choices and admire their cleverness. As with operating system evangelists, a Socratic examination, where you simply ask them to objectively clarify their beliefs based on knowledge of the languages they are comparing, will usually expose their ignorance.

Then there is the economic reality. Billions of lines of C, C++ and Fortran code have been written at great expense and rewriting all this code would be a foolish waste of man-hours.

Base your choice of language on practical considerations: Portability, performance, availability, etc. Ultimately, this should be a sound economic decision. Which language will lead to the lowest overall cost for the project in the long term?

People who understand programming will never tell you that one language is "better" than another. They will talk about specific capabilities, like performance, portability, etc. They will ask questions about your needs, demonstrate knowledge of multiple languages, and try to help you find a language that fits your needs, rather than evangelize their personal favorite.

### 13.4.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What should be the main criterion (singular) for choosing a programming language?
2. What factors contribute to the cost of developing and using software?
3. How do you expose invalid arguments of a language evangelist?

## 13.5 Compiled vs Interpreted Languages

### 13.5.1 Language Performance

When performance is a concern, use a purely compiled language. Interpreted programs are run by another program called an interpreter. Even the most efficient interpreter spends more than 90% of its time parsing (interpreting) your code and less than 10% performing the useful computations it was designed for. Most spend more than 99% of their time parsing and less than 1% running. All of this wasted CPU time is incurred every time you run the program.

Note also that when you run an interpreted program, the interpreter is competing for memory with the very program it is running. Hence, in addition to running an order of magnitude or more slower than a compiled program, an interpreted program will generally require more memory resources to accommodate both your program and the interpreter at the same time.

With a compiled program, the compiler does all this parsing ahead of time, before you run the program. You need only compile your program once, and can then run it as many times as you want. Hence, compiled code tends to run anywhere from tens to thousands of time faster than interpreted code.

For interactive programs, maximizing performance is generally not a major concern, so little effort goes into optimization. Users don't usually care whether a program responds to their request in 1/2 second or 1/100 second.

In High Performance Computing, on the other hand, the primary goal is almost always to minimize run time. Most often, it's big gains that matter - reducing run time from months or years to hours or days. Sometimes, however, researchers are willing to expend a great deal of effort to reduce run times by even a few percent.

There is a middle class of languages, which we will call *pseudo-compiled* for lack of a better term. The most popular among them are Java and Microsoft .NET languages. These languages are "compiled" to a byte code that looks more like machine language than the source code. However, this byte code is not the native machine language of the hardware, so an interpreter is still required to run it. Interpreting this byte code is far less expensive than interpreting human-readable source code, so such programs run significantly faster than many other interpreted languages.

In addition to pseudo-compilation, some languages such as Java include a Just-In-Time (JIT) compiler. The JIT compiler converts the byte code of a program to native machines language *while the program executes*. This actually makes the interpreted code even slower the first time it executes each statement, but it will then run at compiled speed for subsequent iterations. Since most programs contain many loops, the net effect is program execution closer to the speed of compiled languages.

Table 13.2 shows the run time (wall time) of a selection sort program written in various languages and run on a 2.9GHz Intel i5 processor under FreeBSD 13.0. FreeBSD was chosen for this benchmark in part because it provides for simple installation of all the latest compilers and interpreters, except for MATLAB, which is a commercial product that must be installed manually along with a Linux compatibility module.

**Note** FreeBSD's Linux compatibility is *not* an emulation layer, and there is no performance penalty for running Linux binaries on FreeBSD. In fact, Linux binaries sometimes run slightly faster on FreeBSD than on Linux.

This selection sort benchmark serves to provide a rough estimate of the relative speeds of languages when you use explicit loops and arrays. There are, of course, better ways to sort data. For example, the standard C library contains a `qsort()` function which is far more efficient than selection sort. The Unix **sort** command can sort the list in less than 1 second on the same machine. Selection sort was chosen here because it contains a typical nested loop representative of many scientific programs.

All compiled programs were built with standard optimizations. Run time was determined using the **time** command and memory use was determined by monitoring with **top**. The code for generating these data is available on Github.

Memory is allocated for programs from a pool of *virtual memory*, which includes RAM (fast electronic memory) + an area on the hard disk known as *swap*, where blocks of data from RAM are sent when RAM is in short supply. The first number shows the virtual memory allocated, and the second shows the amount of data that resides in RAM. Both numbers are important.

Programs that run 100 times as fast don't just save time, but also extend battery life on a laptop or handheld device and reduce your electric bill and pollution. The electric bill for a large HPC cluster is thousands of dollars per month, so improving software performance by orders of magnitude can have a huge financial impact.

### 13.5.2 Vectorizing Interpreted Code

Most interpreted languages offer a variety of built-in features and functions, as well as external libraries that execute operations on *vectors* (arrays, matrices, lists, etc) at compiled speed (because they are written in compiled languages). These features and functions use compiled loops under the hood rather than explicit interpreted loops. Some programmers may not even realize that they are using a loop, but most operations that process multiple values require a loop at the hardware level, whether or not it is visible in the source code.

For example, MATLAB allows us to perform many operations on entire vectors or matrices without writing an explicit loop:

```
# MATLAB explicit interpreted loop to initialize an array
for c = 1:100000
    list1(c) = c;
end

# MATLAB vectorized list initialization
# Achieves exactly the same result, but faster than interpreted loop above
list1 = [1:100000];
```

| Language (Compiler) | Execution method | Time (seconds) | Memory |
|---|---|---|---|
| C (clang 11) | Compiled | 7.8 | 11 MB (2.7 resident) |
| C++ (clang++ 11) | Compiled | 8.1 | 13 MB (3.9 resident) |
| C (gcc 10) | Compiled | 12.3 | 11 MB (2.7 resident) |
| C++ (g++ 10) | Compiled | 12.4 | 14 MB (4.54 resident) |
| Fortran (gfortran 10) | Compiled | 12.4 | 15 MB (3.65 resident) |
| Fortran (flang 7) | Compiled | 12.5 | 20 MB (7.11 resident) |
| D (LDC 1.23.0) | Compiled | 30.5 | 16 MB (5.2 resident) |
| Rust 1.57 | Compiled | 30.6 | 13 MB (3.5 resident) |
| Java 11 with JIT | Quasi-compiled + JIT compiler | 31.4 | 3,430 MB (44 resident) |
| Octave 6.4.0 vectorized (use min(list(start:list_size) to find minimum) | Interpreted (no JIT compiler yet) | 34.9 | 345 MB (103 resident) |
| GO 1.17 | Compiled | 37.7 | 689 MB (14 resident) |
| Pascal (free pascal 3.2.2) | Compiled | 43.0 | 2.2 MB (1.1 resident) |
| Python 3.8 with numba JIT 0.51 | Interpreted + JIT compiler | 44.5 | 305 MB (107 resident) |
| MATLAB 2018a vectorized (use min(list(start:list_size) to find minimum) | Interpreted + JIT compiler | 51.7 | 5,676 MB (251 resident) |
| R 4.1.2 vectorized (use which.min(nums[top:last]) to find minimum) | Interpreted | 97.4 | 1,684 MB (1,503 resident) |
| MATLAB 2018a | Interpreted + JIT compiler non-vectorized (use explicit loop to find minimum) | 112.2 (1.87 minutes) | 5,678 MB (248 resident) |
| Java 11 without JIT | Quasi-compiled | 408.8 (6.8 minutes) | 3,417 MB (38 resident) |
| Python 3.8 vectorized (use min() and list.index() to find minimum) | Interpreted | 758.8 (12.6 minutes) | 28 MB (18 resident) |
| Perl 5.32 vectorized (use reduce to find minimum) | Interpreted | 2,040.3 (34.0 minutes) | 41 MB (32 resident) |
| R 4.1.2 non-vectorized (use explicit loop to find minimum) | Interpreted | 2159.5 (36.0 minutes) | 120 MB (70 resident) |
| Python 3.8 non-vectorized (use explicit loop to find minimum) | Interpreted | 2362.3 (39.4 minutes) | 26 MB (16 resident) |
| Perl 5.32 non-vectorized (use explicit loop to find minimum) | Interpreted | 2564.5 (42.7 minutes) | 33 MB (23 resident) |
| Octave 6.4.0 non-vectorized (use explicit loop to find minimum) | Interpreted (no JIT compiler yet) | 80096.0 (1334.9 minutes, 22.2 hours) | 345 MB (103 resident) |

Table 13.2: Selection Sort of 200,000 Integers

An another example, many interpreted languages provide a min() function for finding the smallest element in a list. This function may be written in a compiled language, in which case it will be many times faster than an explicit interpreted loop that searches the list. This, along with the MATLAB example above, are examples of *vectorizing* code, i.e. using built-in vector operations instead of explicit loops.

If your interpreted program does most of its computation using compiled functions and vector operations, it may run several times faster than the same program using explicit interpreted loops. However, it will still not approach the speed of a compiled program, and only tasks that can utilize the finite set of compiled functions and vector features will perform well. Not every array and matrix operation can be accomplished using built-in functions or vector operations. When you have to use an explicit interpreted loop, it will be very slow.

This often leaves you with no good options when dealing with large amounts of data. In order to vectorize an operation, the program must inhale all of the data into an array or similar data structure. While this will speed up the code by replacing an interpreted loop with a compiled one, it also slows down memory access by overflowing the cache, and in some cases could even cause memory exhaustion, where the system runs out of swap. In that scenario, the program simply cannot finish.

Arrays are often not inherently necessary to implement the algorithm. For example, adding two matrices stored in files and writing the sum to a third file, does not require reading the matrices into arrays. If we can process one value at a time without using an array, the code will be simpler, and not limited by memory capacity. However, using an explicit loop in an interpreted language in order to avoid using arrays will reduce performance by orders of magnitude. The only option to maximize speed and minimize memory use may be rewriting the code in a compiled language. If using a compiled language, it is almost always preferable not to use arrays, unless there is a block of data that must be accessed repeatedly.

As an example, the R version of the selection sort program was written in two ways: One with entirely explicit loops (the slowest approach with an interpreted language) and the other using an intrinsic function, `which.min()`. The `which.min()` function uses a compiled loop behind the scenes to find the smallest element in the list, the most expensive part of the selection sort algorithm. As you can see in Table 13.2, this leads to a dramatic improvement in speed, but still does not bring R close to the speed and memory efficiency of a compiled language. Using `which.min()` with a subset of the list also has a trade-off in that it vastly increases memory use.

Vectorized variants of the MATLAB, Python and perl sort scripts were also tested, with varying results. Vectorizing Python and perl did not result in nearly as much performance improvement, but on the bright side, the memory trade-off was negligible.

### 13.5.3 Compiled Languages Don't Guarantee a Fast Program

Choosing a compiled language is very important to execution speed, but is not the whole story. It will not guarantee that your code is as fast as it could be, although it will be far less slow than the same algorithm implemented in an interpreted language. Choosing the best algorithms can be actually far more important in some cases. For this reason, computer science departments focus almost exclusively on algorithms when teaching software performance.

Selection sort, for example, is an $O(N^2)$ algorithm, which means that the run time of a selection sort program is proportional to $N^2$ for a list of N elements. I.e., the run time is $N^2$ times some constant $K_{\text{C-selection}}$.

Heap sort is an O(N * log(N)) algorithm, which means that the execution time of the heap sort in Python is proportional to N * log(N) for a list of N elements:

If N is large, say $10^9$, then $N^2$ is $10^{18}$, whereas N * log(N) is 9 * $10^9$, or approximately $10^{10}$. Hence, the heap sort will be about 10 billion times as fast as selection sort for a list of a billion values. Obviously this is more important than the 100-fold speedup we get from a compiled language. Using the best algorithms *and* a compiled language is clearly the best way to maximize performance.

### 13.5.4 No, Compiled Languages are Not Hard

You may encounter criticism of compiled languages for being harder to use than interpreted languages. This is another opportunity for a Socratic examination. If you ask the people telling you this to clarify with some specific examples, you will usually find that they don't know very much about the compiled languages they fear.

For example, many MATLAB users believe that MATLAB is easier because it has "built-in" matrix capabilities. However, most common vector and matrix operations in MATLAB are readily available to C, C++, and Fortran programs in free libraries such as BLAS and LAPACK. In fact, MATLAB uses these same libraries for many of its calculations.

It is often (erroneously) stated that interpreted languages are easier to use because you don't have to compile the program before you run it. I struggled for years trying to understand the origin of this myth. Asking people to explain only revealed their ignorance on the subject, as Socrates often experienced.

I know that compiling programs is no harder than interpreting programs, because I have been doing both, using many languages, for decades. My best guess regarding why people believe this, aside from accepting hearsay on faith, is that there are many existing open source projects with seriously brain-damaged build systems, that most people struggle to install. However, this is in no way the fault of the language. There is no reason that a build system (such as a makefile, discussed in Chapter 21) should be difficult to use. This is entirely on the developer who wrote it. Furthermore, if you install programs via a package manager, difficulties with building the program are not your concern. Any problems will have been resolved by the package manager.

The fact that Random Dude made a mess of his compiled code and build system does not mean that *you* will have a hard time programming in a compiled language. Go about it intelligently, and it will not be much different than using an interpreted language.

### 13.5.5 Summary

The pool of different programming languages in existence today is insurmountable and growing at an accelerating rate. As a result, it is becoming harder and harder to find programmers with skills in a specific language.

The rational approach to overcoming this situation is to develop good general programming skills that will apply to most languages. Stay away from proprietary or esoteric language constructs, tools, and techniques, and focus on practices that will be useful in many languages and on many operating systems.

To develop depth of knowledge, focus on *mastering* one general-purpose scripting language and one general-purpose compiled language. The skills you develop this way are more likely to serve you directly, but will also help you conquer other languages more easily if it proves necessary.

Don't be taken in by the false promises of "higher level" languages. The reality is that programming doesn't get much easier than it is in C or Fortran. Other languages may seem easier to the beginning programmer, but as you progress you will find that overcoming their limitations is impossible or at least more difficult than coding in C.

If you only know how to use interpreted languages, you will have access to a wide range of high performance code written by others, such as the C and Fortran code behind Matlab, Octave, and Python libraries such as Numpy. You will not, however, be able to create your own high performance code. To do that you should learn C (any maybe later expand into C++) or Fortran. Other compiled languages exist, but the vast majority of existing code is written in C, C++, and Fortran. You will not always be writing new code from scratch, but will often find a need to modify or improve existing code instead.

### 13.5.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Which runs faster, a program written in a compiled language or the same program in an interpreted language? Why? By how much?
2. Which uses less memory, a program written in a compiled language or the same program in an interpreted language? Why? By how much?
3. Which is better for an interactive program that doesn't do much computation, a compiled language, or an interpreted language?
4. Which is better for an interactive program that does heavy computation, a compiled language, or an interpreted language?
5. Is Java compiled or interpreted? Explain.
6. List four advantages of a program that runs faster.
7. What does it mean to vectorize code?

8. What is one advantage and one disadvantage of vectorizing? Explain.
9. If you need to improve performance of an interpreted language program, but don't want to increase memory use, what can you do?
10. Does using a compiled language ensure that your programs will be fast? Explain.
11. Are compiled languages inherently harder to use than interpreted languages? Explain.
12. How can one cope with the enormous number of different programming languages available today?

## 13.6 Common Compiled Languages

### 13.6.1 C/C++

C and C++ are the most popular compiled languages for general use over the past several decades. Both are commonly used for scientific programming. Most compilers and interpreters are written in C and C++.

C is a very simple, but powerful high level language that was developed in unison with the Unix operating system. In fact, most of the Unix operating system is written in C. C became the dominant general-purpose programming language on other operating systems such as DOS and Windows during the 1980's. DOS and Windows are also largely written in C. C is best known for producing the fastest code possible in a portable high-level language.

C++ is an extension of C that adds a wealth of new features, many of which are aimed at *object-oriented* programming. Object-oriented programming encourages more modular design of programs, which can be critical for large applications. Most modern languages now have features to support object-oriented programming.

Unlike C, C++ is an extremely complex language that requires a great deal of knowledge and experience in order to use effectively. It is a powerful tool for scientific programming, but has a much high learning curve than C. Mastering the C++ language and keeping up with the new features that are frequently added is a career in and of itself.

Note also that it is possible to write object-oriented code in any language, as covered in Chapter 29. There are numerous resources on the web detailing how to apply object-oriented design principles in C.

### 13.6.2 Fortran

Fortran (Formula Translator) was the first major high-level language, created by a team at IBM led by John Backus in the 1950's. Fortran is a compiled language, originally designed for scientific computing, and has been the most popular scientific programming language throughout most of computing history. Fortran has always had many convenient features for scientific computing, such as built-in mathematical functions, and seamless, efficient support for complex numbers. Performance of Fortran programs is comparable to C or C++.

Fortran is still the language of choice in many areas of scientific computing, including weather forecasting, molecular modeling, etc. A great deal of Fortran software is under active development today, and the future of Fortran appears to be bright.

### 13.6.3 Conclusion

For the average scientist or engineer who needs to write fast code, C is a good starting point. It is a simple language that can be mastered in a relatively short time, unlike C++. It also provides all the features that are needed by most scientific programmers.

### 13.6.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the primary difference between C and C++ from the perspective of a scientist learning to program?
2. Is Fortran still useful, despite being from the 1950s?

## 13.7 Common Interpreted Languages

### 13.7.1 MATLAB and Octave

MATLAB is an interpreted scripting language designed specifically for vector and matrix calculations common in science and engineering. MATLAB handles vectors and matrices with an intuitive syntax, offers a wide variety of built-in, pre-compiled routines for numerical computations, and has other useful features such as sophisticated graphing and plotting routines.

**Caution**
MATLAB is not a general-purpose programming language and is not substitute for high-performance, general-purpose programming languages like C, C++, and Fortran.

*If used properly, and for its intended purpose (vector and matrix calculations)*, MATLAB is a very useful tool. Unfortunately, not all calculations can be done efficiently in MATLAB. Some operations are difficult or impossible to vectorize, and MATLAB has a finite number of built-in functions. Even if the code can be vectorized, doing so is often more difficult than writing the code in a compiled language.

Many people use MATLAB where it is ill-suited for the simple reason that it is the language they are most familiar with. MATLAB programs that use explicit loops to process data one element at a time will run orders of magnitude slower than equivalent compiled programs or a MATLAB program utilizing MATLAB's vector capabilities and/or pre-compiled functions.

To illustrate this point, the following program shows three ways of accomplishing the same basic vector initialization in MATLAB. The first uses the MATLAB colon operator to iterate through values from 1 to 100,000 using a compiled loop within the interpreter. The second uses the pre-compiled linspace() function (which is probably written in C) to accomplish the same iteration. The third uses an explicit interpreted loop, which must be performed by the MATLAB interpreter. Each **toc** statement reports the time elapsed since the previous tic statement.

```
fprintf('Colon operator:\n');
tic
list1 = [1:100000];
toc

fprintf('linspace function:\n');
tic
list2 = linspace(1,100000);
toc

fprintf('Explicit loop:\n');
tic
for c = 1:100000
    list3(c) = c;
end
toc
```

Below is the output produced by MATLAB running on a 2GHz Core Duo system:

```
Colon operator:
Elapsed time is 0.002485 seconds.
linspace function:
Elapsed time is 0.008148 seconds.
Explicit loop:
Elapsed time is 81.661672 seconds.
```

Below is the output produced by Octave running on the same Core Duo system:

```
Colon operator:
Elapsed time is 0.0000169277 seconds.
```

```
linspace function:
Elapsed time is 0.000270128 seconds.
Explicit loop:
Elapsed time is 0.994528 seconds.
```

Note that all three methods do in fact use loops, but the first two use compiled loops while the third uses an interpreted loop. Note also that C code below, using an explicit but compiled loop, executes in about 0.002 seconds, comparable to the MATLAB code without an explicit loop.

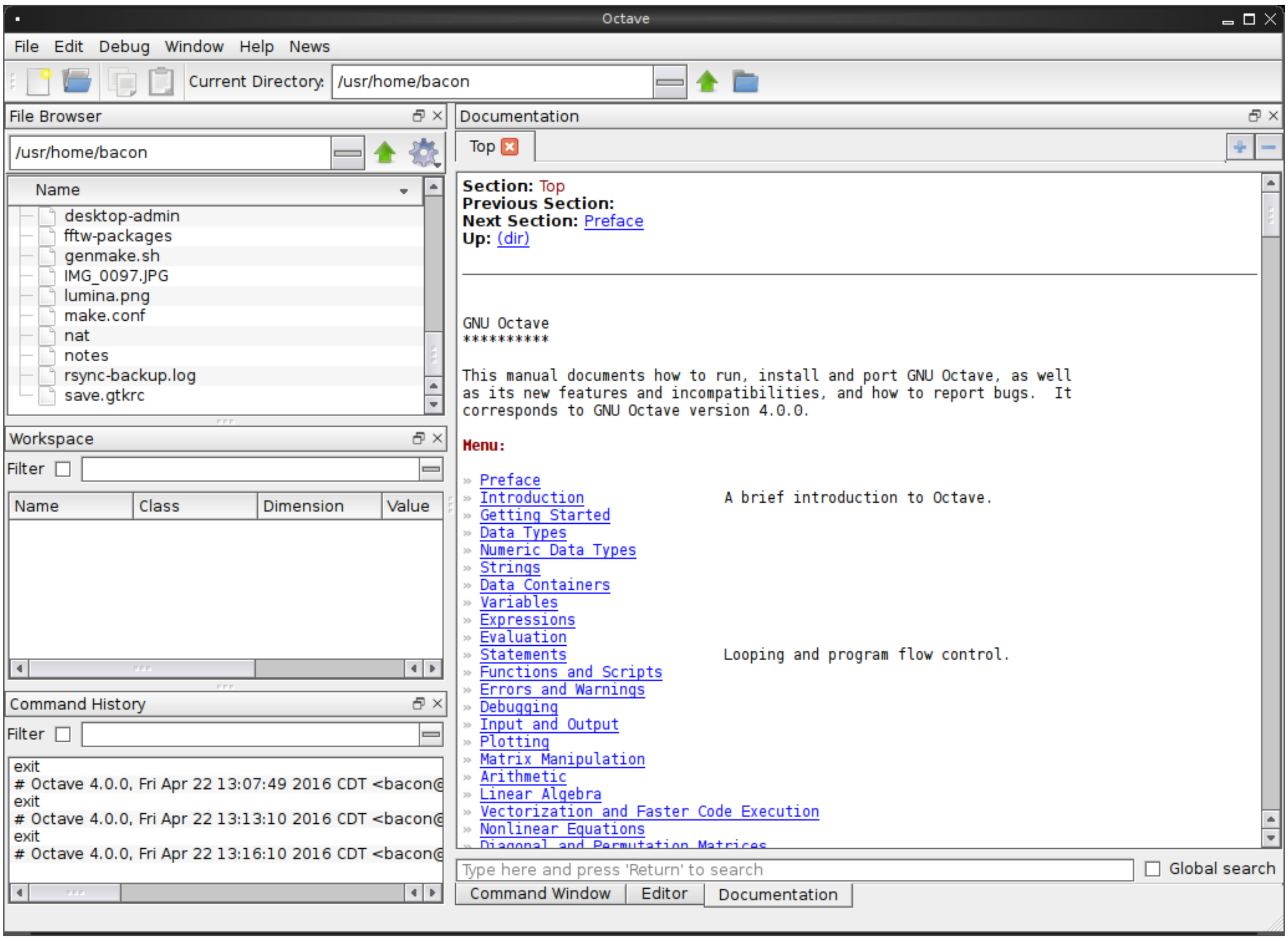
```
size_t  c;
double  list[LIST_SIZE];

for (c=0; c < 100000; ++c)
    list[c] = c;
```

This example illustrates that if MATLAB has no built-in feature such as the vector operation or pre-compiled routine such as linspace() to perform the computations you require, you will need to write code in a compiled language in order to achieve good performance.

One more potential barrier, especially in parallel computing, is the fact that MATLAB is an expensive commercial product and requires a license server in order to run on a cluster. The need to allocate funds, procure licenses, and set up a license server inevitably delays results. In addition, you will not be able to run MATLAB while the license server or the network is down.

One possible solution to this issue is Octave, a free and open-source tool very similar to MATLAB. The Octave interpreter is nearly 100% compatible with MATLAB. In fact, the Octave project considers any differences between MATLAB and Octave to be bugs.

Octave has a graphical user interface (GUI) based on the QT cross-platform GUI tools. The Octave GUI is very similar to MATLAB's GUI, but faster. (The Octave GUI is written in C++, whereas MATLAB's is written in Java.)



```
exit
# Octave 4.0.0, Fri Apr 22 13:07:49 2016 CDT <bacon@
exit
# Octave 4.0.0, Fri Apr 22 13:13:10 2016 CDT <bacon@
exit
# Octave 4.0.0, Fri Apr 22 13:16:10 2016 CDT <bacon@
```

**Section:** Top
**Previous Section:**
**Next Section:** Preface
**Up:** (dir)

```
GNU Octave
**********

This manual documents how to run, install and port GNU Octave, as well
as its new features and incompatibilities, and how to report bugs.  It
corresponds to GNU Octave version 4.0.0.

Menu:

» Preface
» Introduction                   A brief introduction to Octave.
» Getting Started
» Data Types
» Numeric Data Types
» Strings
» Data Containers
» Variables
» Expressions
» Evaluation
» Statements                     Looping and program flow control.
» Functions and Scripts
» Errors and Warnings
» Debugging
» Input and Output
» Plotting
» Matrix Manipulation
» Arithmetic
» Linear Algebra
» Vectorization and Faster Code Execution
» Nonlinear Equations
» Diagonal and Permutation Matrices
```

*The Octave GUI*

The general consensus at the time of this writing is that MATLAB and Octave perform about the same for properly vectorized code, while MATLAB performs better on "bad" code (using explicit loops). Octave is faster at some algorithms and MATLAB is faster at others. In fact, MATLAB and Octave utilize many of the same open source C, C++, and Fortran libraries for many of their common functions. See Table 13.2 for some performance comparison.

### 13.7.2 A Real Example for Comparison to C

Now let's consider a simple real-world example, namely approximating the area under a curve. There are a variety of approaches, such as the trapezoid rule. This example uses rectangular slices with the height equal to the function value at the center of the slice, which provides a balanced estimate and is very easy to program.

Each of the following examples was run using 10,000,000 slices. The first example program illustrates a typical approach to solving the problem in MATLAB:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Program description:
%
%   Compute the area under the curve of the square root function.
%
%   Demonstrates the value of MATLAB's just-in-time compiler.
%   Unlike the more complex selection sort example, MATLAB is able
%   to compile the explicit loop below at run-time, making it almost
%   as fast as a vector operation.
%
%   History:
```

```
%   2013-07-04  Jason Bacon Begin
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic
first_x = 0;
last_x = 1;
slices = 10000000;
slice_width = (last_x - first_x) / slices;

% Use x in the middle of each slice
x = first_x + slice_width / 2;
estimated_area = 0;
while x < last_x
    slice_area = sqrt(x) * slice_width;
    estimated_area = estimated_area + slice_area;
    x = x + slice_width;
end

fprintf('Final x = %0.16f  Estimated area = %0.16f\n', x - slice_width, estimated_area);
toc
```

```
Final x = 0.9999999497501700  Estimated area = 0.6666666666390187
Elapsed time is 0.538943 seconds.
```

The example above uses an explicit loop, which could be too slow for a large number of slices, because the MATLAB interpreter has to execute the statements in the loop many times. Below is a vectorized version of the same program:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Program description:
%
%   Compute the area under the curve of the square root function.
%
%   Demonstrates the use of vector operations.  Vector operations are
%   generally much faster than explicit loops, but require much more
%   memory in cases where vectors or matrices are not otherwise needed.
%   This may cause the program to actually run slower than one without
%   vectors, or even fail if the vectors used are too large to fit in
%   memory.
%
%   History:
%   2013-07-04  Jason Bacon Begin
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic
first_x = 0;
last_x = 1;
slices = 10000000;
slice_width = (last_x - first_x) / slices;

% Use x in the middle of each slice
x = linspace(first_x + slice_width / 2, last_x - slice_width / 2, slices);
slice_area = sqrt(x) * slice_width;
estimated_area = sum(slice_area);
fprintf('Final x = %0.16f  Estimated area = %0.16f\n', x(slices), estimated_area);
toc
```

```
Final x = 0.9999999500000000  Estimated area = 0.6666666666685899
Elapsed time is 0.434779 seconds.
```

The vectorized version uses no explicit loops, so the MATLAB interpreter will not be a bottleneck. Interestingly, however, it does not run much faster than the non-vectorized code. This is partly because the program is so simple that MATLAB's just-in-time (JIT) compiler can compile the explicit loop for us. More complex programs will not benefit from the JIT compiler as much, however, so explicit loops should be avoided where speed is essential. Speed is also limited by the increase in memory use due to the use of large arrays of 10,000,000 elements. This memory use overwhelms the CPU cache (fast memory that holds only a few megabytes), forcing the program to use the larger, slower main memory.

While using vector operations will generally speed up MATLAB code immensely, the down side is generally an increase in memory use. If we were to increase the number of slices used by this program to 1 billion, the program would now use two vectors of 8 billion bytes (8 gigabytes) each. (Each number in MATLAB program typically requires 8 bytes of memory). If your computer does not have 16 gigabytes of available RAM, the program simply won't run. Even if it does, it will not be able to utilize the CPU cache well, and will take much more time per slice than it does for smaller slice counts.

When programming in MATLAB, we are often faced with a choice between using explicit loops, which are very slow, and using huge amounts of memory, which also slows down the program and potentially prevents it from running at all due to memory exhaustion.

A C version of the program is shown below:

```
/***************************************************************************
 *  Description:
 *      Compute the area under the curve if the square root function.
 *
 *  Arguments:
 *      None.
 *
 *  Returns:
 *      Standard codes defined by sysexit.h
 *
 *  History:
 *  Date        Name        Modification
 *  2013-07-04  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  first_x,
            last_x,
            x,
            slice_width,
            estimated_area;
    unsigned long   slices;

    first_x = 0.0;
    last_x = 1.0;
    slices = 10000000;
    slice_width = (last_x - first_x) / slices;

    // Use x in the middle of each slice
    x = first_x + slice_width / 2;
    estimated_area = 0.0;
    while (x < last_x)
    {
        estimated_area += sqrt(x) * slice_width;
        // Fast, but causes x to drift due to accumulated round off when
        // adding imperfect x + imperfect slice_width
        x = x + slice_width;
    }
    printf("Final x = %0.16f  Estimated area = %0.16f\n",
        x - slice_width, estimated_area);

    return EX_OK;
}
```

```
Final x = 0.9999999497501700  Estimated area = 0.6666666666390187
        0.11 real         0.10 user         0.00 sys
```

Sine C is a compiled language, there is no disadvantage to using explicit loops, and no need to use arrays. As a result, this program is much faster than either MATLAB script, and will never be limited by available memory.

Note that this program, as well as the MATLAB program with an explicit loop, has a potential issue with *drift* caused by round-off error, which is inherent in all computer floating point systems. (Floating point is a storage format similar to scientific notation, which is used by computers to approximate the real number set.)

If the width of a slice cannot be represented perfectly in the computer's floating point format, then each successive value of x will be off by a little more than the previous. For example, representing 1/10 in computer floating point is like trying to represent 1/3 in decimal. It requires an infinite number of digits to represent accurately, so a computer can only approximate it to about 16 decimal digits.

After a large number of slices are processed, the value of x could drift far from the center of the theoretical slice we were aiming for. The MATLAB linspace() function, used in the vectorized example, avoids this issue. The C program below demonstrates how to avoid this problem by basing each value of x on the initial value, instead of the previous value:

```
/***************************************************************************
 *  Description:
 *      Compute the area under the curve if the square root function.
 *
 *  Arguments:
 *      None.
 *
 *  Returns:
 *      Standard codes defined by sysexit.h
 *
 *  History:
 *  Date        Name        Modification
 *  2013-07-04  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  first_x,
            last_x,
            base_x,
            x,
            slice_width,
            estimated_area;
    unsigned long
            slices,
            current_slice;

    first_x = 0.0;
    last_x = 1.0;
    slices = 10000000;
    slice_width = (last_x - first_x) / slices;
    current_slice = 0;
    estimated_area = 0.0;
    base_x = first_x + slice_width / 2; // Start in middle of first slice
    while (current_slice <= slices)
    {
        // Compute each x directly from a base value to avoid accumulating
        // round-off error.  This is slightly slower than x = x + slice_width
        // due to the multiplication, but produces more accurate results.
        x = base_x + (slice_width * current_slice++);
        estimated_area += sqrt(x) / slice_width;
    }
    printf("Final x = %0.16f  Estimated area = %0.16f\n",
        x - slice_width, estimated_area);

    return EX_OK;
}
```

This version produces a final value of x that is exactly what it should be.

```
Final x = 0.9999999500000000  Estimated area = 0.6666667666686176
        0.11 real         0.11 user         0.00 sys
```

Because this version of the program uses an additional multiply operation to compute each value of x, it is slightly slower. It does, however, produce more accurate results, especially for high slice counts.

### 13.7.3 Perl

Perl is an interpreted language with many of the same convenient syntactic features of the Unix shells, plus a wealth of built-in functions such as you would find in the standard C libraries. Perl is designed primarily for text-processing, and is commonly used on WEB servers to process data from WEB forms. It supports a rich set of features for handling regular expressions and is preferred by some people over traditional Unix tools like **grep**, **awk**, and **sed** for many text processing tasks.

### 13.7.4 Python

Python is another interpreted scripting language, but is popular enough in scientific programming to be worth special mention. Compiled libraries such as SciPy (sigh-pie) and NumPy (num-pie), written in C, C++ and Fortran, make it possible to write Python scripts to do some heavy computation, with the same caveats as MATLAB or Octave. As long as most of the calculations are performed inside the compiled library routines, performance should be acceptable. Iterative code written in Python to do intensive computation, however, will result in run times orders of magnitude slower than equivalent C, C++, or Fortran code.

Performance of raw Python code can be improved using **Cython**, a compiler for building Python modules from slightly modified Python code, or using **Numba**, a just-in-time (JIT) compiler for normal Python code. Performance and memory use will not match that of a true compiled language, but will be vastly improved over purely interpreted code. ( See Table 13.2. )

Python is becoming a popular alternative to MATLAB for many tasks. While it is not at all similar to MATLAB, it does offer many of the same capabilities. For example, one of the strengths of MATLAB is it's plotting capabilities, which are often used to visualize data. The Matplotlib Python library allows Python scripts to easily create plots, graphs, and charts as well.

One of Python's primary advantages over MATLAB is that it is free, open source, and highly portable. Python scripts can be run on most Unix-compatible systems as well as Windows, at no monetary cost and with minimal installation effort.

The Python community does a good job of encouraging adherence to standards through tools like *Distutils* and the *Python Package Index* (PyPI), a website and repository where Python packages are quality-checked before inclusion on the site. This facilitates the development and sharing of software written in Python in much the same way as language-independent package managers. In fact, Python projects that have met the standards to be included on PyPI are almost invariably trivial to add to other package systems like FreeBSD ports and pkgsrc.

### 13.7.5 R

R is an open source statistics package with a built-in scripting language. It is a very popular substitute for commercial statistics packages such as SPSS and SAS. Like MATLAB/Octave, the R scripting language is a vector based language and can perform many common vector operations (operations on whole arrays) quickly, because the R interpreter uses compiled loops behind the scenes.

Explicit loops in R scripts are interpreted and extremely slow, and seen in Table 13.2. Like MATLAB/Octave, R code must therefore be vectorized as much as possible to achieve best performance. Again, this will likely increase memory use, an unavoidable trade off when using interpreted languages.

The R CRAN project provides add-on R packages that can be easily installed from within R by any user. Many of these packages integrate compiled C, C++, or Fortran code in order to maximize speed and flexibility. Bioconductor is another repository similar to R CRAN that provides many packages for bioinformatics analyses.

R has many frustrating quirks, such as the inability to use certain features within a loop, different behavior when code is run from a script than when entered from the keyboard, etc. With its wealth of statistics libraries, R is an excellent tool for doing quick-and-dirty standard statistical analyses. It is not so convenient to use for developing applications, however, as running those applications will usually require the user to know R programming.

### 13.7.6 Mixing Interpreted and Compiled Languages

Some developers write much of their code in an interpreted scripting language and only the portions that require speed in a compiled language. If you are using Unix shell scripts, there is no integration necessary. You simply write complete programs in the compiled language and run them from your scripts.

There are also tools available for integrating compiled subprograms into many interpreted languages. The MEX system allows you to write subprograms in C, C++, or Fortran and integrate them into a MATLAB script. Octave also supports MEX as well as its own system MKOCTFILE.

Cython is a Python-to-C translator and module generator. It allows you to write code in an extended Python syntax and compile it for better performance. The Cython compiler generates C or C++ source code that can be compiled to a Python module, which can be imported into Python programs. You may not get the same performance you would from hand-written C, but it will be orders of magnitude faster than interpreted Python.

It's important to consider whether you want to deal with the complexity of integrating multiple languages. Savvy programmers developing complex applications may prefer this approach while less savvy users or those writing simpler programs may find it easier write the entire program in C or Fortran.

### 13.7.7 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Can all computations be done efficiently in interpreted languages such as MATLAB, Perl, Python, and R? Explain.
2. Describe one advantage and one disadvantage of Octave vs MATLAB?
3. What is the primary niche of each of the following languages?
   - MATLAB
   - Perl
   - Python
   - R
4. What are the pros and cons of mixing interpreted and compiled languages?

## 13.8 Engineering Product Life Cycle

This section introduces the *engineering product life cycle*, which is used to assure a rational development process and quality results in all fields of engineering, including software engineering. In software engineering, we refer to it as the *software life cycle*. Although the software life cycle may not be the primary focus of this course, it should be practiced in all programming endeavors, including college courses, personal projects, and professional development.

It is, unfortunately, not usually stressed in early computer programming classes. If it were, students would avoid developing bad habits and struggle far less as they develop increasingly complex programs. We present it here before getting into C and Fortran to help you avoid these issues.

The product life cycle has been extensively studied and refined over time, and is the topic of entire semester courses in most engineering disciplines. Our coverage here is a very high level overview, using a 4-step process which is outlined in the following sections.

### 13.8.1 Specification

Specification is understanding the essence of the problem to be solved as clearly as possible. Specifications may evolve during design and implementation stages as new insights are gained from working on the solution. However, every effort should be made to write specifications that will require minimal change during later stages of development.

A clear specification makes the steps that follow an order of magnitude easier and more enjoyable.

### 13.8.2 Design

The design phase involves examining possible solutions to the problem with a completely open mind. The decision to write software or develop a non-software solution does not occur until *after* the design phase. Instead, the design phase focuses on the *abstract process* of solving the problem.

A design contains only algorithms, mathematical formulas, diagrams, etc. It does not mention any specific programming languages or other tools used to *implement* (build) the solution. One should avoid any thoughts about how the solution will be implemented during the design phase. Such thoughts lead to bias that will reduce the quality of the design.

A well-developed design makes implementation and testing an order of magnitude easier and more enjoyable.

### 13.8.3 Implementation and Testing

Implementation involves building something to test the process developed in the design stage. If you developed a good design, the implementation stage will be relatively uneventful and enjoyable. If you find yourself struggling during implementation, then you either need to develop a better programming process or go back and correct deficiencies in the design.

If the best solution found during the design stage is to use existing hardware or software, there is little to do in this stage. If it involves developing new hardware or software, then implementation involves the following:

1. Selecting the right tools and materials. For software, this means computer hardware, operating system, and programming language. For a hardware design, it means electronic or mechanical devices and fabrication techniques. A good choice here requires a solid understanding of the design, and knowledge of *many* available tools. Far too often, software developers choose an operating system or language because it's the only one they know, leading to a poor quality product that does not serve the customers' needs well.
2. Performing the implementation. For software, this means writing the code. For hardware, it could mean building prototypes of the hardware.

Testing is the heart of good engineering. Solid scientific theories and technology can help us design and build products faster and cheaper, but testing is the only way to ensure quality.

Testing is not a separate stage in the development time line, but occurs continuously starting at the beginning of the implementation stage, and continues indefinitely, long after implementation and product release.

Testing should occur incrementally, following *every* small change throughout the implementation process. Generally, you should add no more than about 10 lines of code before testing again. Add a caveman debug statement along with the new code to show that it is working correctly. Remove it or comment it out after the code is verified. If you find yourself struggling during the implementation and testing stage, it is probably because you are violating this principle.

```
#!/bin/sh -e

# Trim one file at a time
for file in *.fastq; do
    # Caveman debug statement to show that the loop is processing the
    # correct files.  Remove or comment out after verifying.
    # Then uncomment the actual trim command below and test that.
    printf "$file\n"

    # Trim reads
    # fastq-trim --3p-adapter1 AGATCGGAAGAG \
    #     --polya-min-length 3 $file $trimmed
done
```

Additional types of testing occur following completion of the product, such as *alpha testing*, which refers to formal in-house testing of the complete product before releasing it to customers, and *beta testing*, which refers to testing performed by a limited group of real customers before officially releasing the product for general use.

Incremental testing should catch 99% of the bugs in a program. Alpha testing should catch almost all of the few remaining bugs before the program is released for beta testing. Beta testers should not find any additional bugs.

Beta testing should be a formality just to make absolutely certain that the product is ready for release. Beta testers may also reveal room for improvement in the user interface. Developers are not usually in tune with typical users, so this kind of feedback is important.

**Note**
Every script or program should be tested on more than one platform (e.g. BSD, Cygwin, Linux, Mac OS X, etc.) immediately, in order to shake out bugs before they cause problems.
The fact that a program works fine on one operating system and CPU does not mean that it's free of bugs.
By testing it on other operating systems, other hardware types, and with other compilers or interpreters, you will usually expose bugs that will seem obvious in hindsight.
As a result, the software will be more likely to work properly when time is critical, such as when there is an imminent deadline approaching and no time to start over from the beginning after fixing bugs. Encountering software bugs at times like these is very stressful and usually easily avoided by testing the code on multiple platforms in advance.

### 13.8.4 Production

After the product is fully tested, it is ready to release to the general public. Unfortunately, many products are released without being properly tested. Some individuals and organizations simply lack the discipline to implement proper test procedures. Some have adopted the odd notion that they should adhere to a rigid release schedule in order to appease rigid customers who do not understand product development. This simply does not work. We cannot predict how long it will take to identify and fix all the major bugs in a product.

### 13.8.5 Support and Maintenance

No product is ever really finished. There will always be more flaws to be discovered and improvements to be made. This is especially true with software, which will likely need updates just to keep it working with newer dependent libraries and operating systems on which it runs. Writing software is exactly like adopting a puppy. It's fun and exciting at the beginning, but a lot of work. The software will grow over time, and become easier to manage. Most of all, it's a commitment of a decade or more.

Implementing code in a stable language (one that isn't changing rapidly) will reduce maintenance costs. C, Fortran, POSIX Bourne shell, and awk are examples of highly stable languages that are still heavily used and have not changed significantly in many years. Hence, even long-abandoned C, Fortran, Bourne shell, and awk code still works today and will continue to work for years to come.

In contrast, there is a great deal of code written to Python 2 standards that would still be useful today, if not for the fact that it does not run under a Python 3 interpreter and Python 2 is no longer maintained or secure. Python 2 was originally scheduled to be sunsetted in 2015, but not surprisingly, there were many Python 2 scripts that people still needed, but no one was willing or able to upgrade to Python 3. As a result, the Python project continued to maintain Python 2, at great expense, until 2020. (https://www.python.org/doc/sunset-python-2/) There are still today numerous useful Python 2 scripts that will not run under Python 3, and people running Python 2 interpreters with known bugs and security holes that will never be fixed.

A new C++ standard is published every few years, adding new features and deprecating old ones, so old C++ code often fails to build under new compilers. This can usually be solved by forcing the new compiler to use an older standard. Compiling new code with older compilers is often simply impossible. Users of Redhat Enterprise Linux, which is based on heavily patched older tools for stability and long-term compatibility, often need to install a second compiler in order to build newer programs.

If you decide to implement code in a rapidly evolving language, you must be prepared to make significant updates every few years in order to maintain its usefulness. If you abandon such code, it will quickly become a fossil.

### 13.8.6 Hardware Only: Disposal

Hardware engineers must also think about what will happen to the product when it reaches its end of life. Should it simply be thrown away? Does it contain valuable materials that should be recycled? Does it contain toxic materials that should not go in a landfill? All of these questions tie into the design and implementation stages. Implementing a product in a way that makes disposal easy is a wise move that will prevent many problems for the customer and the company.

### 13.8.7 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Briefly describe the six stages of the engineering product life cycle.
2. Describe the three major types of testing and when they occur.
3. What is the most likely reason if someone is having a hard time figuring out what code to write for a new section of a program?
4. What is the most likely reason someone is having a hard time locating a bug in some code they just wrote?

# Chapter 14

# Data Representation

## 14.1 What Everyone Should Know About Data Representation

Computer science students typically spend at least one or two semesters studying computer architecture (hardware design) and machine/assembly language. Most will never become hardware designers, or assembly language programmers, but understanding how data are represented in computer hardware makes all of us better programmers.

Knowing the limitations of computer number systems is necessary in order to write programs that produce correct and precise output and also in writing the most efficient programs possible.

This chapter provides a very brief overview of these limitations. We will cover the standard number systems used by modern computers alongside some historical and hypothetical systems to help put them in context.

### 14.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are two advantages of understanding computer data representation if we are not doing hardware design?

## 14.2 Representing Information

We'll start with some general number system theory to put things in context. Otherwise, using common computer number systems tends to become a mechanical process that students don't really understand. A little depth of understanding will make you better at your job and prevent costly problems down the road.

Any information can be represented using patterns of two or more symbols. The English alphabet uses 26. Our number system uses 10, the digits 0 through 9, because most of us have 10 fingers. True story: My former barber is an exception with 9, because he was a bit clumsy with sharp instruments. Our 10-digit number system is called decimal, and our 26-letter alphabet would be a *hexavigesimal* information system, which has nothing to do with witchcraft. A system with 8 symbols is called *octal*, and one with 16 is called *hexadecimal*.

Computers use just two symbols, each represented by a different voltage, because it is easier to design circuits with just two states. Any information system using only two symbols is called *binary*. In modern integrated circuits, these states are commonly 0 volts and 3.3 volts. On paper, we represent these as 0 and 1, because *sometimes* the patterns represent numbers. 0V can represent 0 while 3.3V represents 1 (positive logic) or the other way around (negative logic). It makes no difference to functionality, though it could affect power consumption if there tend to be more 0s or more 1s in the device.

In ASCII (American Standard Code for Information Interchange, pronounced "askee") and the ISO (International Standards Organization) extensions of ASCII, the letter 'A' is represented as 01000001. THIS IS NOT A NUMBER. However, we can treat it like a binary number and convert it to decimal 65 for convenience. Conversions like this one are introduced in Section 14.10.

Whether or not the 0s and 1s in a pattern represent a number, we often refer to them as *binary digits*, or *bits* for short.

### 14.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the minimum number of symbols needed to represent information?
2. Why does computer hardware use binary rather than decimal?
3. How are 0s and 1s of binary represented in digital circuits?
4. Do all values in a computer represent numbers? If not, what is an example of non-numeric data?

## 14.3 Numeric Limitations of Computers

Computers cannot represent infinite mathematical sets like the integers or real numbers we learned about in grade school math. Each digit in a number requires memory in which to store it. An infinitely large number has an infinite number of digits, and no computer has infinite memory for storing all these digits. Many finite numbers, such as PI, also have an infinite number of digits, so they cannot be represented precisely in computer hardware.

CPUs process information using a limited set of simple operations known as *machine instructions*. A single machine instruction might add two integers or copy data from one place to another. In addition to the limits on memory, computer CPUs have much smaller limits on how much data they can process at once (with a single machine instruction). This limit is known as the CPU's *word size*. At the time of this writing, most computers can process at most 64 bits at once, which corresponds to an integer value of no more than $2^{64}$-1, or 1.84467440737e+19. (More on this later.) Older computers can only process 32 bits (an integer of $2^{32}$-1, or 4,294,967,295). Still older computers and many modern micro-controllers are limited to 16 bits (an integer of 65,535).

It is possible for a CPU to process values larger than its word size. It simply can't do it all at once (in a single machine instruction). A 32-bit computer can add 64-bit numbers, but it has to do it 32-bits at a time, and hence it takes twice as long. This is known as *multiple precision arithmetic*. Multiple precision addition and subtraction are straightforward. Multiple precision multiplication and division are much more complex, so multiplying 64-bit numbers on a 32-bit computer will take more than two instructions.

Software exists for processing arbitrarily large (arbitrary precision) values, with any number of significant figures, regardless of the word size of the hardware. The **bc** calculator, which is a standard part of every Unix system, is an example. Just be aware that processing arbitrary precision numbers takes an arbitrarily long time and requires an arbitrary amount of memory.

Another limit imposed by a CPU's word size is how much memory it can address. Computer memory is essentially an array of values and the address is a subscript of limited size. The size of a memory address in most CPUs is the same as its word size. Hence, most 32-bit CPUs use a 32-bit memory address and can address no more than $2^{32}$ bytes (4 gibibytes) of RAM. Many computers today use much more than this. A 64-bit CPU can theoretically address up to $2^{64}$ bytes (16 exbibytes), which is far more than today's computers can have installed. In fact, this would require over a billion 16 gibibyte memory chips. A typical PC has 4 or 8 memory slots.

### 14.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the main difference between mathematical sets such as integers and real numbers, and computer number systems?
2. What is a computer's word size?
3. Can computers process numbers larger than their word size?
4. Describe two advantages of a 64-bit computer over a 32-bit computer.

## 14.4 Fixed Point Number Systems

A fixed point number system has a fixed number of total digits and a fixed number of fractional digits.

Such a system has a limited *range* (minimum and maximum values) and *precision* (maximum digits in any value, like significant figures in science). As with any number set, the results of operations on two of the numbers must also be within the set. This can sometimes yield different results than we would see for an infinite number system.

**Caution** Precision should not be confused with *accuracy*. Accuracy refers to how close a value is to reality, and reflects on how well it was measured. Precision is the number of significant figures that can be stored and retrieved, whether or not they are accurate. A high quality scale that reports weight to the microgram but is not properly calibrated (set to zero when empty) will be precise, but not accurate, since that measurement will be consistently wrong.

**Example 14.1** A fixed-point system

Suppose a system has 4 decimal digits, two of which are fractional. Then all numbers have the form ##.##.
The range is 00.00 to 99.99 and the precision is 4 decimal digits.

```
    94.01       10.00       03.33
+   12.58   /   03.00   *   03.00
---------   ---------   ---------
  1 06.59       03.33       09.99
```

The sum above is 06.59, not the same as the 106.59 we get using real numbers. We have to drop the '1' carried over from the addition of the leftmost digits to fit within our fixed-point system. This is known as *overflow*. Overflow can be detected either by checking for a carry out of the leftmost digit, or by noting that the sum is less than one of the terms being added.
The answer 10.00 / 03.00 = 03.33 is not as precise as we would get using real numbers. Using real numbers, the result would be 3.33333... with the digits going on forever. In our fixed-point system, we have to stop after 2 fractional digits and our result is imprecise by 0.0033333... This is commonly called *round-off* error, though the correct term here is *truncation* error. If a value is rounded to the nearest true value, such as 20.00 / 3.00 = 6.67, it is round-off error. If digits are simply lost, such as 20.00 / 3.00 = 6.66, it is truncation. Fixed point systems typically suffer from truncation error, not round-off.
Note also that reversing the division by multiplying the result by 03.00 does not produce the original value of 10.00. In fact, the error in the result 0.33 (0.033333...) is multiplied by 3 along with the value itself, so the error in the final result of 0.99 is 0.01. This reveals a potential problem in computer programs that perform many calculations using limited number systems. The error can grow out of control unless we employ strategies to control it.

### 14.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Define each of the following:
   (a) Range
   (b) Accuracy
   (c) Precision
2. What happens when the result of an arithmetic operation in a mathematical set such as integers or real numbers is outside the range of the fixed-point system we are using? Show an example using a 3-digit decimal system with 2 fractional digits.
3. What happens when the result of an arithmetic operation in a mathematical set such as integers or real numbers has more fractional digits than the fixed-point system we are using? Show an example using a 3-digit decimal system with 2 fractional digits.

4. Show the following computations in a fixed-point decimal system with 4 total digits and 1 fractional digit. Indicate whether overflow, round-off, or truncation occurs.

    (a) 145.0 + 993.1

    (b) 100.0 / 006.0

    (c) 100.0 / 006.0 * 006.0

## 14.5 Modular Number Systems

A modular integer system is a special case of fixed-point systems where there are no fractional digits. It operates within a subset of integers.

Modular systems are important because *all* computers use them to approximate integer operations. In order to represent true integers, which can have an infinite number of digits, a computer would require an infinite amount of memory to store those digits.

**Example 14.2** A modulo-100 System

To illustrate the concept, consider a subset of the non-negative integers limited to 2 decimal digits.
Such a set only includes value from 0 to 99, and is known as the modulo-100 set. The value 100 is known as the *modulus*.

```
    94          10          03
+   12      /   03      *   03
-------     -------     -------
  1 06          03 R1       09
```

Note that the same problems with overflow and round-off/truncation occur with any fixed-point system. The integer addition result of 106 is too large to fit in 2 digits, to the '1' is dropped and the result is 06.
Results of integer division may be truncated, leaving a remainder. We can correct the results of the multiplication above by adding the remainder from division: 03 * 03 + 1 = 10.

When performing operations such as addition or multiplication on modular systems, we follow the exact same procedure as we would for any other fixed point system.

Looking at it another way, when counting in a modular system, after reaching the largest allowable value, we "circle back" or "wrap" to zero. I.e., 99 + 1 = 0 in modulo-100 arithmetic.

Computers behave the same way when performing "integer" operations. We must be aware that what we call integers in computer programming are actually modular number systems, limited to a certain number of binary digits. They can and often do overflow, and this does not normally trigger an error condition in a program. Instead, the program continues unaware and produces incorrect results if it was assuming that it's working in the infinite integer set. The programmer usually must explicitly check for this following calculations that have the potential to overflow, or code them in such a way that overflow will never happen. Some programming languages insert checks automatically, but this severely reduces program performance, because it requires additional machine instructions following every integer arithmetic instruction.

**Note** There are 3 possible results from a computational error in a program:

1. The program produces a meaningful error message telling the user what went wrong. This is what happens when the programmer is competent.
2. The program crashes or produces a misleading or uninformative error message. We have all been annoyed by these issues.
3. The program continues and produces incorrect results. This is the worst possible outcome, the kind of thing that causes billion-dollar space probes to crash.

Programmers must be aware of the range of the modular integer sets used by any computer and take measures to ensure that overflows do not cause incorrect results.

### 14.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a modular number system?
2. What would we call a modular system with 3 decimal digits?
3. What is the range of a modulo-10 system?
4. What happens when we add 1 to the largest possible value in a modular number system?
5. What typically happens when an "integer" overflow occurs in a computer program?

## 14.6 Binary Data Representation

As mentioned earlier, computers store all data as patterns of 0s and 1s. Information systems using 0s and 1s are collectively known as *binary information systems*.

Each 0 or 1 in a binary value is called a *bit*, which is short for *binary digit*. This can sometimes be misleading, since not all bits actually represent digits in a number. A bit could be part of any type of information, including numbers, letters of the alphabet, hieroglyphics, Boolean (true/false) values, Chinese characters, encrypted passwords, etc.

A collection of 8 bits is called a *byte*. A byte is the most common unit of storage for electronic memory, i.e. each memory location holds one byte of data in most computers. It is also usually the smallest amount of data that a machine instruction can process, although it is possible to manipulate individual bits within a byte. Processing data smaller than a byte is generally not as easy as processing whole bytes.

A collection of 4 bits is called a *nybble*.

A *word* is the maximum amount of data a CPU can process at once, and is usually 1, 2, 4, or 8 bytes (8 to 64 bits).

Table 14.1 summarizes the simple binary data quantities.

| **Term** | **Meaning** |
|---|---|
| bit | Binary Digit (0 or 1) |
| byte | 8 bits |
| nybble | 4 bits |
| word | CPU-dependent, usually 8, 16, 32, or 64 bits |

Table 14.1: Basic Binary Units

Numeric data in computers are stored using several different binary number formats, all of which use a finite number of *binary digits (bits)*, and therefore are subject to overflow and round-off or truncation.

### 14.6.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a bit? Is this always an accurate name for the information it represents? Why or why not?
2. What is a byte? State two reasons why this is an important unit of information.
3. What is a word?

## 14.7 Metric and Binary Quantities

Working with computers requires knowing some terminology referring to large quantities of data. You are probably familiar with metric prefixes such as "kilo-" and "mega-" in terms like kilogram and megawatt from science classes. However, computers use base 2, not base 10, so the metric prefixes don't quite fit when referring to common computer quantities.

Table 14.2 shows a separate set of binary prefixes representing powers of 2 that are close to the powers of 10 used by the metric prefixes. They use the same first two letters as the metric prefix, followed by 'bi' (from binary).

Metric and binary prefixes are often confused and interchanged. This is not a problem where precision is not critical, such as in casual conversation, since they represent similar values. For example, it is often stated that a computer has 4 gigabytes (4 * $10^9$ bytes) of RAM, when in fact it has 4 gibibytes (4 * $2^{30}$ bytes).

In fact, using binary terms like gibibyte in casual conversation with non-technical people might make them think you've been drinking too much.

| Metric prefix | Abbreviated | Value | Binary prefix | Abbreviated | Value |
|---|---|---|---|---|---|
| kilo- | K- | 10^3 | kibi- | Ki- | 2^10 (1,024) |
| mega- | M- | 10^6 | mebi- | Mi- | 2^20 (1,048,576) |
| giga- | G- | 10^9 | gibi- | Gi- | 2^30 (1,073,741,824) |
| tera- | T- | 10^12 | tebi- | Ti- | 2^40 (1,099,511,627,776) |
| peta- | P- | 10^15 | pebi- | Pi- | 2^50 (1.12589990684e+15) |
| exa- | E- | 10^18 | exbi- | Ei- | 2^60 (1.15292150461e+18) |
| zetta- | Z- | 10^21 | zebi- | Zi- | 2^70 (1.18059162072e+21) |
| yotta- | Y- | 10^24 | yobi- | Yi- | 2^80 (1.20892581961e+24) |

Table 14.2: Prefixes for larger quantities

The abbreviated forms of the binary prefixes are the same as the metric, but with an 'i' inserted. For example, 2 GB means 2 gigabytes, while 2 GiB means 2 gibibytes.

### 14.7.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. You meet an attractive fashion designer at a party who seems impressed that you're a computer engineer, and proceeds to tell you that [s]he just bought a new laptop with "16 gigabytes" of memory. The correct response (assuming you want the conversation to continue) is:
   (a) "Very cool. Does it help much with your design modeling?"
   (b) "Since you're only a fashion designer, I'm not surprised that you would confuse gigabytes with gibibytes. RAM sizes are always a power of two, so you can never really have 16 gigabytes of RAM. Giga- is a metric unit. Don't feel bad, though, 'cuz smart people make the same mistake sometimes."
   (c) "I personally would never buy a computer with less than 32. Modern operating systems and applications are memory pigs because the programmers are mostly idiots, but I don't have time to rewrite everything for them so I just fork over a few bucks for more hardware to work around their incompetence."
   (d) Other (write your response here).

## 14.8 The Arabic Numeral System

As you probably know, we use a weighted-digit *positional notation* called the Arabic system. ( As opposed to the Roman numeral system. ) The Arabic system is much more convenient for performing mathematical manipulations. It was actually invented in India, but first put to general practical use in the Arab world somewhat later when its value to science and commerce was recognized.

A number represented in the Arabic system is a weighted sum of products. Each digit is multiplied by a power of the *radix*, or *base*, depending on its position. The power on the radix decreases from left to right, and the digit just left of the *radix point* (decimal point in decimal numbers) always has a power of 0.

$458.12 = 4 * 10^{2} + 5 * 10^{1} + 8 * 10^{0} + 1 * 10^{-1} + 2 * 10^{-2}$

We use a radix of 10 because most of us have 10 fingers, and in the early days of the Arabic system, most math was done by counting fingers. Normal people assume the base is 10 when looking at a number. But we, as computer scientists are not limited to one base, and in fact other bases work better for us in many situations.

In our world, the example above is written as $458.12_{10}$ to avoid ambiguity. It could also be a hexadecimal (base 16) number, written as $458.12_{16}$. It cannot be an octal (base 8) number, since it contains the digit 8, and octal digits range from 0 to 7.

In a non-graphical text environment, we can indicate bases using an underscore, such as 458.12_10.

Note that the '.' in a number is only called the decimal point if the number is decimal. It is called the octal point in a base 8 number, the binary point in a base 2 number, etc.

### 14.8.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a radix?
2. Break down the decimal number 34.23_10 into a weighted sum.
3. What do we call the '.' in 34.77_16?

## 14.9 Number Bases

### 14.9.1 Converting Other Bases to Decimal

We can represent a number in any base >= 2, since we need at least 2 different values to distinguish a digit. Names of the first 15 number bases are given in Table 14.3.

In computer science, the most useful bases are 2, 8, and 16. The reasons for this will become clear in the following sections.

The digits for any given number base range from 0 to base-1. As we know, base 10 uses digits 0 to 9. Base 8 (octal) uses 0 to 7 and base 2 (binary) uses 0 to 1. Using larger digits would make it possible to represent a value in more than one way. For example, if binary fixed point allowed the use of the digit '2', then the number two could be represented as either '2' or '10'.

$1001.11_{2} = 1 * 2^{3} + 1 * 2^{0} + 1 * 2^{-1} + 1 * 2^{-2} = 8 + 1 + 1/2 + 1/4 = 9.75_{10}$

$34.5_{8} = 3 * 8^{1} + 4 * 8^{0} + 5 * 8^{-1} = 24 + 4 + 5/8 = 28.625_{10}$

$12.0_{2}$ and $80.6_{8}$ are not valid numbers.

For bases larger than 10, we use letters for digits greater than 9: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

$3F.8_{16} = 3 * 16^{1} + 15 * 16^{0} + 8 * 16^{-1} = 48 + 15 + 8/16 = 63.5_{10}$

| Base | Name |
|---|---|
| 2 | Binary |
| 3 | Ternary |
| 4 | Quaternary |
| 5 | Quinary |
| 6 | Senary |
| 7 | Septenary |
| 8 | Octal |
| 9 | Nonary |
| 10 | Decimal |
| 11 | Undecimal |
| 12 | Duodecimal |
| 13 | Tridecimal |
| 14 | Tetradecimal |
| 15 | Pentadecimal |
| 16 | Hexadecimal |

Table 14.3:

#### The Double-and-Add Method

The easiest conversion is binary to decimal. In this case we are dealing with powers of 2, which are easy, and multiplying by either 0 or 1, since those are the only possible digits. So, if there's a 1, we add that power of 2 and if there's a 0, we don't.

One method is scanning left to right, doubling the sum so far for each new digit (whether 0 or 1) and adding the next digit to the sum.

```
1101_2

1)  1 (leftmost digit)
2)  2 + 1 = 3
3)  6 + 0 = 6
4)  12 + 1 = 13
```

This method is easy to program since it reads the string of 0s and 1s from left to right, which is how they would arrive from a file or a keyboard. We don't need to store the digits in an array.

```
sum = 0;
while ( (ch = getchar()) != '\n' )
    sum = sum * 2 + ch - '0';
```

When converting by hand, you may find it more intuitive to go from right to left, doubling the value we add (the power of 2) each at step and then only adding if the digit is 1.

```
1101_2 = 1 + 0 + 4 + 8 = 13
```

### 14.9.2 Converting Decimal to Other Bases

As converting to decimal is a process of adding and multiplying, converting from decimal to other bases is naturally a matter of dividing and subtracting. The basic left-to-right method is as follows:

1. Find highest power of the target base that fits in the number.
2. Divide by the power of the base.
3. Repeat using the remainder from the previous division and the next lower power of base. Note that we do not stop when the remainder is 0. We must continue at least until base$_0$, further if there are fractional digits.

**Example 14.3** Conversion from Decimal

$439.5_{10}$ to binary, octal, hexadecimal, and base 5.

| Base | Highest power |
|---|---|
| 2 | 256 |
| 8 | 64 |
| 16 | 256 |
| 5 | 125 |

Table 14.4: Highest power of each base <= 439

```
Binary              Octal           Hex             Base 5

      110110111.1         667.4           1B7.8           3224.222
    +------------       +----           +------         +---------
256 | 439.5         64  | 439.5     256 | 439.5     125 | 439.5
      256                 384             256             375
     +-----               +----          +-----           +----
 128 |183.5             8 |55.5       16 |183.5        25 |64.5
      128                  48             176             50
      +----                +---           +----           +----
   64 |55.5              1 |7.5        1 | 7.5          5 |14.5
       0                   7              7               10
      +----                +---          +---             +---
   32 |55.5            1/8|0.5       1/16|0.5           1 |4.5
       32                  0.5            0.5              4
      +----                ---            ---             +---
   16 |23.5                0              0            1/5|0.5
       16                                                 0.4
       +---                                               +---
    8 |7.5                                           1/25|0.1
       0                                                  0.08
      +---                                                +----
    4 |7.5                                         1/125|0.02
       4                                                  0.016
      +---                                                +-----
    2 |3.5                                                |0.004
       2                                                 (repeats)
      +---
    1 |1.5
       1
      +---
   1/2|0.5
       0.5
       ---
       0
```

#### The Remainder Method

We can also work in the other direction (from right to left) by repeatedly dividing by the base (using integer division) and taking the remainder as the next digit to the left. Here we stop as soon as we get a quotient of 0.

**Example 14.4** $37_{10}$ to binary

```
Decimal to binary

37_10   = 100101_2
```

```
37/2    = 18 r1     1
18/2    = 9 r0      01
9/2     = 4 r1      101
4/2     = 2 r0      0101
2/2     = 1 r0      00101
1/2     = 0 r1      100101_2
```

**Example 14.5** $135_{10}$ to octal

```
135/8   = 16 r7     7
16/8    = 2 r0      07
2/8     = 0 r2      207_8
```

**Example 14.6** $135_{10}$ to hex

```
135/16  = 8 r7      7
8/16    = 0 r8      87_16
```

### 14.9.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the names for base 2, base 8, and base 16?
2. Perform each of the following conversions using the method of your choice.
   (a) 100101_2 to decimal
   (b) 234_8 to decimal
   (c) FC1_16 to decimal
   (d) 32_10 to binary
   (e) 129_10 to octal
   (f) 129_10 to hexadecimal

## 14.10 Binary Fixed Point

A binary fixed point system is another example of an Arabic numeral system. The only distinction from decimal is the radix. Binary uses a radix of 2 instead of 10.

We specify the radix (base) using a subscript on the number. From now on, we will not assume a base of 10, but will always specify the base.

$1001.101_2 = 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3}$

$= 8 + 0 + 0 + 1 + .5 + 0 + .125$

$= 9.625_{10}$

An easy way to read binary is by working left and right from the binary point, doubling or halving the value of the digits at each step:

```
1001.101_2 = ?

1001 = 1*1 + 0*2 + 0*4 + 1*8 = 9_10
.101 = 1*.5 + 0*.25 + 1*.125 = 0.625_10

1001.101_2 = 9.625_10
```

### 14.10.1 Limitations of Binary

Try to convert 1/10 to binary. It actually cannot be done: It's like trying to represent 1/3 in decimal, in that requires an infinite number of digits. This is a problem for monetary systems using dollar amounts. One solution invented by computer engineers many decades ago is *Binary Coded Decimal*, or *BCD*, where each nybble stores one decimal digit. Then someone realized that storing money amounts in pennies rather than dollars makes BCD unnecessary, since everything is an integer.

### 14.10.2 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What distinguishes binary fixed point from decimal?
2. Besides range and precision, what is another limitation of binary fixed point?

## 14.11 Binary/Octal/Hexadecimal Conversions

A very convenient property of number bases is this: If A = $B^N$, then each digit in a base A number will be exactly N digits in the base B number.

Since 16 = $2^4$, each hexadecimal digit corresponds to exactly 4 bits.

Since 8 = $2^3$, each octal digit corresponds to exactly 3 bits.

Hence, when converting between binary and either octal or hex, we can convert one octal or hex digit at a time. There is no long multiplication or division necessary. We simply convert one octal or hex digit at a time using Table 14.5 or Table 14.6, or convert each group of 3 or 4 bits. With a little practice, you will have this table memorized and be able to "see" the bits in an octal or hex number at a glance.

| Binary | Octal |
|---:|---:|
| 000 | 0 |
| 001 | 1 |
| 010 | 2 |
| 011 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |

Table 14.5: Binary/Octal Conversion

For this reason, we use octal or hexadecimal to represent values where we want to know the values of individual bits. Binary numbers tend to be long, so we represent them in hex or octal to save space. The lower the base, the more digits it takes to represent a typical value.

$FFC3.2_{16}$ = 1111 1111 1100 0011 . $0010_2$

| Binary | Hex |
|---:|---:|
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 2 |
| 0011 | 3 |
| 0100 | 4 |
| 0101 | 5 |
| 0110 | 6 |
| 0111 | 7 |
| 1000 | 8 |
| 1001 | 9 |
| 1010 | A |
| 1011 | B |
| 1100 | C |
| 1101 | D |
| 1110 | E |
| 1111 | F |

Table 14.6: Binary/Hex Conversion

$374.4_8 = 011\ 111\ 100\ .\ 100_2$

When converting from binary to base $2^N$, we need to be careful. Always group bits beginning at the radix point and move outward. The number of bits may not be a multiple of N, so we may have to pad the leftmost group with leading 0s and the rightmost group with trailing 0s.

$11001.11_2 = 011\ 001\ .\ 110_2 = 31.6_8$

$111010.011_2 = 0011\ 1010\ .\ 0110_2 = 3A.6_{16}$

**Note** When converting large numbers from decimal to binary or from binary to decimal, smart people first convert to hexadecimal. This greatly reduces the number of steps in long multiplication or long division. People who are paid by the hour to do manual number conversions are not advised to use this shortcut.

### 14.11.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How many bits would each base 32 digit represent?
2. Perform the following conversions.
   (a) 11FC.2_16 to binary
   (b) 1100101111.01_2 to hex
   (c) 773550.5_8 to binary
   (d) 1001001010010.1_2 to octal

## 14.12 Unsigned Binary Integers

**Note** There are 10 kinds of people in the world: Those who know binary, and those who don't.

### 14.12.1 Introduction

An unsigned binary integer is simply a binary fixed-point system with no fractional digits.

Unsigned binary integers are modular number systems, where the modulus is usually a power of 2.

**Example 14.7** A 4-bit unsigned binary number system

A 4-bit unsigned binary number has values ranging from $0000_2$ ($0_{10}$) to $1111_2$ ($15_{10}$). Hence, it is a modulo-$10000_2$, or modulo-$16_{10}$.

We identify bits in a binary integer by the power of 2 at that position. Hence, the rightmost bit is called "bit 0", the next one to the left is "bit 1", and so on. The leftmost bit will have a power of N-1, where N is the number of bits, since we start at 0. In 1011, bit 0 is 1, bit 2 is 0, and bit 3 is 1. there is no bit 4.

Modern computers typically support binary integers of 8, 16, 32, and 64 bits. A 64-bit computer supports all of these integer sizes, a 32-bit computer supports all but 64-bit, etc.

### 14.12.2 Range

The largest value in any unsigned binary integer system is the one containing all 1s, just as the largest decimal number is the one containing all 9s.

Largest modulo-$1000_{10}$ = $999_{10}$.

Largest modulo-$1000_2$ = $111_2$.

If we add 1 to a 4-bit binary value of all 1s, note what happens:

```
    111
    1111_2
+       1
--------
   10000_2 = 2^4
```

This is the same thing that happens when we add 1 to $9999_{10}$.

```
    111
    9999_10
+       1
--------
   10000_10 = 10^4
```

If we add one to the largest value possible in 4 bits, we get $2^4$ (which requires 5 bits). If we add 1 to the largest value possible in N bits, we will get $2^N$. Hence, the largest value possible in N bits, is $2^N$ - 1, just as the largest possible value with N decimal digits is $10^N$ - 1. Table 14.7 shows the ranges of common unsigned integer sizes.

| Bits | Range |
|---|---|
| 8 | 0 to $2^8$-1 (255) |
| 16 | 0 to $2^{16}$-1 (65,535) |
| 32 | 0 to $2^{32}$-1 (4,294,967,295) |
| 64 | 0 to $2^{64}$-1 (18,446,744,073,709,551,615 = 1.844674407 * $10^{19}$) |

Table 14.7: Unsigned Binary Integer Ranges

Note that even a 64-bit integer cannot hold Avogadro's constant, 6 * $10^{23}$. In order to represent very large numbers with only 32 or 64 bits, we must use a system other than fixed-point. This is discussed in Section 14.19.

Note also that we cannot express fractional or irrational numbers or negative numbers, but only non-negative integers. Systems for representing signed integers and non-integers are covered in later sections.

How many bits are needed to represent a number? For this we use high-school algebra. If y = $b^x$, then x = $\log_b(y)$. If the result is not an integer, we need the next higher integer since we obviously cannot have portions of bits in digital hardware. Mathematically, we use the "ceiling" function, which rounds up to the nearest integer >= N. For an N-bit number:

```
    maximum-value = 2^N - 1
    maximum-value + 1 = 2^N
    N = ceiling(log_2(maximum-value + 1))
      = ceiling(ln(maximum-value + 1) / ln(2))
```

How many bits needed for 60,000?

```
    N = ceiling(log_2(60,000 + 1))
      = ceiling(ln(65,536) / ln(2))
      = ceiling(15.87)
      = 16
```

For convenience, you can also use Table 14.8 and Table 14.9. Just find the lowest power of 2 greater than the number you want to represent, and you'll immediately know how many bits it requires. The lowest power of 2 greater than 60,000 is 65,536, which is 2ˆ16, so we need 16 bits.

| **N** | **$2^N$** |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |
| 11 | 2048 |
| 12 | 4096 |
| 13 | 8192 |
| 14 | 16384 |
| 15 | 32768 |
| 16 | 65536 |
| 17 | 131072 |
| 18 | 262144 |
| 19 | 524288 |
| 20 | 1048576 |
| 21 | 2097152 |
| 22 | 4194304 |
| 23 | 8388608 |
| 24 | 16777216 |
| 25 | 33554432 |
| 26 | 67108864 |
| 27 | 134217728 |
| 28 | 268435456 |
| 29 | 536870912 |
| 30 | 1073741824 |
| 31 | 2147483648 |

Table 14.8: Powers of 2

| **N** | **$2^N$** |
|---|---|
| 32 | 4294967296 |
| 33 | 8589934592 |
| 34 | 17179869184 |
| 35 | 34359738368 |
| 36 | 68719476736 |
| 37 | 137438953472 |
| 38 | 274877906944 |
| 39 | 549755813888 |
| 40 | 1099511627776 |
| 41 | 2199023255552 |
| 42 | 4398046511104 |
| 43 | 8796093022208 |
| 44 | 17592186044416 |
| 45 | 35184372088832 |
| 46 | 70368744177664 |
| 47 | 140737488355328 |
| 48 | 281474976710656 |
| 49 | 562949953421312 |
| 50 | 1125899906842624 |
| 51 | 2251799813685248 |
| 52 | 4503599627370496 |
| 53 | 9007199254740992 |
| 54 | 18014398509481984 |
| 55 | 36028797018963968 |
| 56 | 72057594037927936 |
| 57 | 144115188075855872 |
| 58 | 288230376151711744 |
| 59 | 576460752303423488 |
| 60 | 1152921504606846976 |
| 61 | 2305843009213693952 |
| 62 | 4611686018427387904 |
| 63 | 9223372036854775808 |

Table 14.9: Powers of 2

### 14.12.3 Arithmetic

Addition in binary works the same way as in decimal. We need only remember that our largest possible digit is 1, so when adding 1 + 1, the result is not 2, but 10, and we carry the 1. 1 + 1 + 1 is 11, so the result is 1, carry a 1.

```
     11     111     Carry
    1001    0011
+   0011 +  0111
    ----    ----
    1100    1010
```

In fact, the process is the same for any base. We simply need to remember what our largest digit is and hence when to carry a 1.

```
    1
    27_8        19_16
+   21_8    +   21_16
--------    ---------
    50_8        3A_16
```

Subtraction, multiplication, and division are all basically the same as in decimal as well.

### 14.12.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the range of a 20-bit unsigned integer? Express your answer in binary and as an expression including a power of 2.
2. What are three limitations of unsigned binary?
3. What are bits 4 and 5 in the value 1000101010_2?
4. What is 0110 + 0011?

## 14.13 Signed Integers

Since there are only two values that can be represented in a digital circuit, we must somehow indicate negative numbers using only 0s and 1s.

There are numerous ways to accomplish this, some of which are more intuitive for humans and some of which make hardware design simpler and more efficient. The following sections provide an overview of common signed integer systems.

## 14.14 Sign-Magnitude

### 14.14.1 Format

The sign-magnitude binary format is the simplest conceptual format for expressing signed (positive and negative) integers. To represent a number in sign-magnitude, we simply use the leftmost bit to represent the sign, where 0 means positive and 1 means negative, and the remaining bits to represent the magnitude (absolute value).

A 8-bit sign-magnitude number would appear as follows:

| Sign | Magnitude |
|---|---|
| Bit 7 | Bits 6-0 |

Table 14.10: 8-bit sign-magnitude format

### 14.14.2 Negation

To negate sign-magnitude numbers, simply toggle the sign bit. Note that this leads to having two representations for the number zero, +0 = 00000000 and -0 = 10000000.

### 14.14.3 Conversions

What are the decimal values of the following 8-bit sign-magnitude numbers?

- $10000011_{SM}$ = -(2 + 1) = -3
- $00000101_{SM}$ = +(4 + 1) = 5

Represent the following in 8-bit sign-magnitude:

- -15 = -(8 + 4 + 2 + 1) = 10001111
- +7 = +(4 + 2 + 1) = 00000111

### 14.14.4 Addition and Subtraction

Addition and subtraction require attention to the sign bit. If the signs are the same, we simply add the magnitudes as unsigned numbers and watch for overflow. If the signs differ, we subtract the smaller magnitude from the larger, and keep the sign of the larger.

### 14.14.5 Range

Since the magnitude is an unsigned binary integer, range is computed as it is for unsigned binary, but with one bit fewer. Hence, for 8-bit sign-magnitude, the magnitude is a 7-bit unsigned integer and the range is therefore $+/-2^7-1$. In general, the range of an N-bit sign-magnitude number is $+/-2^{N-1}-1$.

### 14.14.6 Comparison

Comparison works as with unsigned binary if the signs are the same. If the signs differ, the positive value is larger.

### 14.14.7 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Perform the following conversions.
   (a) 11111111 8-bit SM to decimal
   (b) 01111111 8-bit SM to decimal
   (c) -7 decimal to 8-bit SM

## 14.15 One's complement

### 14.15.1 Format

Sign-magnitude is conceptually simple, but messy to implement in hardware. For this reason, other possibilities were explored, including one's complement.

Positive values in one's complement are the same as unsigned binary or sign-magnitude.

### 14.15.2 Negation

To negate a one's complement value, we invert *all* bits. Like sign-magnitude, one's complement has two representations for zero, e.g. +0 = 00000000 and -0 = 11111111.

**Caution**
The term "one's complement" can refer to the binary format (representation) or the negative of a one's complement binary value (the one's complement of 1011 is 0101, and negating is sometimes called "taking the one's complement"). Watch out for this and use the context of the sentence to understand which one we're talking about.
I've often wondered if people who create this kind of confusion are just evil, and intent on making students suffer. However, we should never ascribe to malice what can be explained by incompetence.
In this text, if we want you to negate a value, we will say negate the value. The term "one's complement" refers to the binary number system.

A more universal term for a value with all bits inverted is simply the *complement*. The complement of a binary value N is sometimes called "N prime" and is written as N'. For example, 00110101' = 11001010 and 11001010' = 00110101.

### 14.15.3 Conversions

What are the decimal values of the following 8-bit one's complement numbers? If the number is positive, which is indicated by a 0 in the leftmost bit, we just treat it like an unsigned integer. If the leftmost bit is 1, indicating a negative value, we invert it, convert the positive (unsigned) value, and add a '-' sign.

- 00001010 = +(8 + 2) = +10
- 10001010 = -(01110101) = -(1+4+16+32+64) = -117

### 14.15.4 Addition and Subtraction

Addition and subtraction work like unsigned for positive numbers. If negative numbers are involved, arithmetic is slightly complicated. One's complement is not generally used in hardware implementations for this reason.

### 14.15.5 Range

Range of an N-bit one's complement system is the same as N-bit sign-magnitude, +/-$2^{N-1}$-1.

### 14.15.6 Comparison

Comparison works like unsigned if both values have the same sign. If the signs differ, the positive number is greater.

### 14.15.7 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Perform the following conversions.

   (a) 11111111 8-bit one's comp to decimal

   (b) 01111111 8-bit one's comp to decimal

   (c) -7 decimal to 8-bit one's comp

## 14.16 Two's Complement

### 14.16.1 Format

The most common format used to represent signed integers in modern computers is called *two's complement*. This is a slight variation on one's complement that turns out to be remarkably convenient to implement in hardware.

A positive integer in two's complement is the same as unsigned, sign-magnitude, and one's complement. It always has a 0 in the leftmost bit (*sign bit*).

$+14_{10} = 01110_{\text{two's comp}}$

### 14.16.2 Negation

To negate a two's complement number, a process sometimes called "taking the two's complement", we invert all the bits and add one.

$-01110_{\text{two's comp}} = 01110' + 1 = 10001 + 1 = 10010_{\text{two's comp}}$

**Caution**

The term "two's complement" can refer to the binary format (representation) or the negative of a two's complement binary value (the two's complement of 1011 is 0101, and negating is sometimes called "taking the two's complement"). Watch out for this and use the context of the sentence to understand which one we're talking about.

I've often wondered if people who create this kind of confusion are just evil, and intent on making students suffer. However, we should never ascribe to malice what can be explained by incompetence.

In this text, if we want you to negate a value, we will say negate the value. The term "two's complement" refers to the binary number system.

One of the beauties of two's complement is that the same negation process works for both positive and negative numbers. We might think that to reverse a negation, we should subtract one and invert the bits, but doing that produces the exact same result as inverting and adding 1 again! Weird, huh?

```
10010 - 1 = 10001, 10001' = 01110
10010' = 01101, 01101 + 1 = 01110
```

More examples:

```
-(0001) = 1110 + 1 = 1111
-(1111) = 0000 + 1 = 0001
-(1110) = 0001 + 1 = 0010
-(0010) = 1101 + 1 = 1110
-(0000) = 1111 + 1 = 0000   Oops, that can't be right!
-(1000) = 0111 + 1 = 1000   Oops, that can't be right!
```

So what happened with the negation of 0000 and 1000? Well, by adding 1 after inverting, we trade in the redundant representations of 0 (0000 and 1111) in one's complement for one more negative value. 1111 becomes -1 instead of 0, 1110 becomes -2 instead of -1, and 1000 becomes -$2^{N-1}$. Negating 1000 actually causes an overflow, since there is no +$2^{N-1}$ possible. The largest positive value is +$2^{N-1}$.

Convert the following 4-bit two's comp values to decimal:

```
0111 = +(1 + 2 + 4) = +7
1000 = -(0111 + 1) = -(1000) = -8
0110 = +(2 + 4) = +6
1001 = -(0110 + 1) = -0111 = -(1 + 2 + 4) = -7
1110 = -(0001 + 1) = -0010 = -2
```

### 14.16.3 Another Way to Look at Two's Complement

We can also view two's complement as the same as unsigned binary, with one exception: The leftmost bit represents the negation of what it normally would. For example, in 3-bit unsigned binary, bit 2 (the leftmost bit) would be $2^2$ = 4. So, in two's complement, it's -4.

The remaining bits are treated as positive powers of 2, so we can actually convert negative values from two's complement to decimal without negating the whole number:

```
Binary  Unsigned          Two's comp
000     0 + 0 + 0 = 0      0 + 0 + 0 =  0
001     0 + 0 + 1 = 1      0 + 0 + 1 =  1
010     0 + 2 + 0 = 2      0 + 2 + 0 =  2
011     0 + 2 + 1 = 3      0 + 2 + 1 =  3
100     4 + 0 + 0 = 4     -4 + 0 + 0 = -4
101     4 + 0 + 1 = 5     -4 + 0 + 1 = -3
110     4 + 2 + 0 = 6     -4 + 2 + 0 = -2
111     4 + 2 + 1 = 7     -4 + 2 + 1 = -1
```

### 14.16.4 Addition and Subtraction

The other bizarre property of two's complement is that addition works exactly like unsigned addition, regardless of sign! This means a computer that uses two's complement to store signed integers can use the same adder circuit to do both signed and unsigned addition. Subtraction is done by simply negating and adding. Since a computer needs the ability to negate numbers anyway, no additional hardware is needed to support signed integers or subtraction. Two's complement is obviously part of a conspiracy among computer scientists to put electrical engineers out of work. And if you believe that, I can offer you a great deal on a bridge I own in Brooklyn, which was left to me by my late uncle.

```
Binary      Unsigned    two's comp
    0 1
    0101        5       +5
+   1001    +   9   +   -7
--------------------------------
    1110       14       -2
```

### 14.16.5 Range

Two's complement essentially takes one bit away from the value for use as a sign bit. Half of the bit patterns represent negative values and half represent non-negative values.

The largest positive value in N-bit two's complement is 0111...111, which is $2^{N-1}$-1.

The smallest negative value in N-bit two's complement is 1000...000, which is -$2^{N-1}$.

| Bits | Range |
|---|---|
| 8 | $-2^7$ (-128) to $+2^7$-1 (+127) |
| 16 | $-2^{15}$ (-32,768) to $+2^{15}$-1 (32,767) |
| 32 | $-2^{31}$ (-2,147,483,648) to $+2^{31}$-1 (+2,147,483,647) |
| 64 | $-2^{63}$ (-9,223,372,036,854,775,808) to $+2^{63}$-1 (9,223,372,036,854,775,807) |

Table 14.11: Two's Complement Integer Ranges

### 14.16.6 Comparison

Comparison of two's complement values is the same as unsigned if the numbers have the same sign. If the signs are different, then the positive number is larger. We cannot use unsigned comparison if the signs differ. This is the one thing that two's complement does not do conveniently.

```
A       B       Unsigned    Two's comp
0111    0110    >           >
1111    1000    >           >
0111    1111    <           >
```

This is due to the fact that two's complement rearranges the binary patterns on the number line, so that the latter half of the patterns (those beginning with 1), are actually less than those in the first half. With unsigned integers, a value with a 1 in the leftmost bit is obviously greater than anything with a 0 there. The opposite is true for two's complement.

```
0        +7   -8       -1       Two's comp
0000 ... 0111 1000 ... 1111
0        +7   +8       +15      Unsigned
```

As we can see above, 1000 is greater than 0111 in unsigned binary, but less than in two's complement. Hence, we will need more circuitry for two's complement comparison.

### 14.16.7 Overflow Detection

Overflow in two's complement is indicated by a result with the wrong sign. I.e., if you add two positives and get a negative result, or add two negatives and get a positive.

**Note** It is not possible to get an overflow when adding numbers of opposite signs. Think of it this way. When adding a positive and a negative, we always move toward and possibly across the center of the number line, and we can at most move half the length of the number line, so we will never reach either end.

```
|---------------------------------|---------------------------------|
                                   Start anywhere in the positives
                                   Add the maximum negative value
                                   and we still won't reach the
                                   minimum value.

 |---------------------------------|
 Finish                            Start
```

Examples:

```
    111
    0111    +7      0111    +7      1111    -1
+   0011    +3      1000    -8      1000    -8
-----------------------------------------------------------------
```

```
    1010    -6      1111    -1      0111    +7

    OV              No OV           OV
```

### 14.16.8 Extension and Reduction

Two's complement values are extended to larger formats by simply copying the sign bit to all new positions, a process called *sign extension*.

```
4-bit       8-bit       16-bit              Decimal
0111        00000111    0000000000000111    +7
1110        11111110    1111111111111110    -2
```

To reduce a value to fewer bits, we have no choice but to simply remove the leftmost bits. Note that this is likely to change the value (effectively overflowing the smaller bit size). This is an unavoidable risk when truncating values.

### 14.16.9 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Negate two's complement 01101111.
2. Negate two's complement 00000000.
3. Perform the following conversions.
   (a) 11111111 8-bit two's comp to decimal
   (b) 01111111 8-bit two's comp to decimal
   (c) -7 decimal to 8-bit two's comp
   (d) -4_10 to two's comp
   (e) +12_10 to two's comp
   (f) 8-bit two's complement 11111010 to decimal.
   (g) 8-bit two's complement 00010011 to decimal.
4. What is the range of a 10-bit two's complement value? Express the answer in binary and in decimal.
5. Indicate which of each pair of two's complement values below is greater.
   (a) 0111, 0101
   (b) 1110, 1101
   (c) 0110, 1111
6. What is the sum of two's complement values 0110 and 0011? Verify by converting the two terms and the sum to decimal. Was there an overflow? Explain in terms of the resulting bits.
7. Extend the following 8-bit two's complement values to 16 bits.
   (a) 01110010
   (b) 11110010

## 14.17 Biased Notation

### 14.17.1 Format

Biased notation stores a signed number N as an unsigned value N+B, where B is the bias. B is usually half the unsigned range, though we could in theory choose any bias we want.

For example, to store the value -3 in 4-bit bias-7 notation, we simply add -3 + 7 = 4 and store it as unsigned binary 0100.

To find out what 1110 represents in 4-bit bias-7 notation, we convert the binary representation to decimal and subtract the bias: $1110_2$ = 2 + 4 + 8 = 14 - 7 = +7.

```
Binary       0000 0111 1000 1111
Unsigned        0    7    8   15
Bias-7         -7    0    1    8
Two's comp      0    7   -8   -1
```

### 14.17.2 Negation

Subtract twice the value. E.g., to negate +6, subtract 12. To negate -6, subtract -12.

### 14.17.3 Addition and Subtraction

Complicated.

### 14.17.4 Range

The range is simply 0-bias to (maximum unsigned)-bias.

### 14.17.5 Comparison

Numbers stored in biased notation can be compared in the same way as unsigned numbers. Unlike two's complement, which uses what would be the higher unsigned values to represent negative numbers (0000 to 0111 are positive and 1000 to 1111 are negative), biased notation keeps the values in the same order, but simply shifts the number line.

### 14.17.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Perform the following conversions.
   (a) 11111111 8-bit bias-127 to decimal
   (b) 01111111 8-bit bias-127 to decimal
   (c) -7 decimal to 8-bit bias-127
2. What is an advantage of biased notation over two's complement?
3. What is an advantage of two's complement over biased notation?

## 14.18 Hex and Octal with Signed Numbers

When representing signed binary values in octal or hexadecimal, we pay no attention to the binary format and just convert the bits as we would with an unsigned binary number. Hence, we never see a '-' sign with octal or hexadecimal. The '-' sign is only used to indicate negative decimal values.

Hence, we can think of octal and hexadecimal as compressed representations of binary and nothing more. If given an octal or hexadecimal number, we must also be told what the binary format is in order to determine its value.

Likewise, to convert a signed decimal value to octal or hexadecimal, we must be told the binary format and number of bits. $-7_{10}$ is $11111000_{2, \text{one's complement}} = F8_{16, \text{one's complement}}$, but $11111001_{2, \text{two's complement}} = F9_{16, \text{two's complement}}$.

The main caveat is that we must be aware of how many bits the value contains in case it differs from the number of digits shown. Whatever octal or hex digits are shown are to be converted to binary. Any digits not shown are assumed to be 0. The examples below should clarify why this is important.

```
177_8 in 7-bit two's complement     = 1 111 111     = -1_10
177_8 in 8-bit two's complement     = 01 111 111    = +127_10
377_8 in 8-bit two's complement     = 11 111 111    = -1_10
377_8 in 16-bit two's complement    = 0 000 000 011 111 111 = +255_10

9F_16 in 8-bit two's complement     = 1001 1111     = -(01100000 + 1) = -97_10
9F_16 in 8-bit bias-127             = 1001 1111     = +32_10
9F_16 in 8-bit one's complement     = 1001 1111     = -01100000 = -96_10
9F_16 in 16-bit two's complement    = 0000 0000 1001 1111   = +159_10
9F_16 in 16-bit one's complement    = 0000 0000 1001 1111   = +159_10

1001000_2 7-bit two's complement    = 48_16 = 110_8
1001000_2 7-bit unsigned binary     = 48_16 = 110_8
1001000_2 7-bit bias-63             = 48_16 = 110_8
```

### 14.18.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Perform the following conversions:
   (a) 1001101 sign-magnitude to octal
   (b) 1001101 two's complement to octal
   (c) 1001101 bias-127 to octal
   (d) 73_16 to 8-bit two's comp
   (e) 73_16 to 8-bit bias-63
   (f) 73_16 7-bit two's comp to binary and decimal
   (g) 73_16 8-bit two's comp to binary and decimal

## 14.19 Floating Point

### 14.19.1 The Basics

Integer number formats cannot store fractional values. We can get around this using a fixed point format, but this severely limits both the range and the number of fractional digits. A 64-bit fixed point system with 32 whole number digits and 32 fractional digits has a maximum range of $2^{32}$ - 1 even when we don't need the fractional bits. A system that utilizes all the bits where they are needed would generally be a better option for approximating real numbers.

Floating point gets around the limitations of fixed point by using a format similar to scientific notation.

$3.52 * 10^3 = 3520$

A scientific notation number consists of a *mantissa* (3.52 in the example above) a *radix* (always 10), and an *exponent* (3 in the example above). Hence, the general format of a scientific notation value is:

$\text{mantissa} * \text{radix}^{\text{exponent}}$

The *normalized* form always has a mantissa greater than or equal to 1.0, and strictly less than 10.0. Written mathematically, the mantissa is in the set [1.0,10.0). As you may recall from algebra, [] encloses a set including the extreme values and () encloses a set excluding them. We can denormalize the value and express it in many other ways, such as $35.2 * 10^2$, or $0.00325 \times 10^6$. For each position we shift the digits of the mantissa relative to the decimal point, we are multiplying or dividing the mantissa by a factor of 10. To compensate for this, we simply increase or decrease the exponent. Adding 1 to the exponent multiplies $10^{\text{exponent}}$ by 10, for example. this compensates for shifting the mantissa digits one position to the right, which divides the mantissa by 10.

Denormalizing is necessary when adding scientific notation values with different exponents:

```
    3.52 * 10^3
+   1.97 * 10^5
-----------------------------------------


      1
    0.0352 * 10^5
+   1.97 * 10^5
-----------------------------------------
    2.0052 * 10^5
```

Adjusting the mantissa and exponent is also sometimes necessary to normalize results. For example, if we had chosen to denormalize the other value above, we would see the following:

```
    3.52 * 10^3
+   1.97 * 10^5
-----------------------------------------


      1
      3.52 * 10^3
+   197.0 * 10^3
-----------------------------------------
    200.52 * 10^3   Not normalized
    2.0052 * 10^5   Normalized
```

For this reason, it's best to denormalize the number with the smaller exponent. This way, the result of the addition is likely to be already normalized.

A binary floating system is basically the same, but stores a signed binary mantissa and a signed binary exponent, and usually uses a radix of 2. Using a radix of 2 (or any power of 2) allows us to normalize and denormalize by simply shifting the binary digits in the mantissa and adding or subtracting from the exponent, just as we do with scientific notation. Shifting binary digits in the mantissa n bits to the left or right multiplies or divides the mantissa by $2^n$, just as shifting decimal digits multiplies or divides by 10.

$00010_2 * 2^3 = 01000_2 * 2^1$.

Standard floating point formats are defined by the IEEE society. The IEEE formats are slightly more complex than necessary to understand floating point in general, so we will start with a simpler example here.

### 14.19.2 Floating Point and Abstraction

Many people get confused about floating point because of their inability to abstract it into a hierarchy of simple components. If we solve one piece of the puzzle at a time and then put them together, it's no more difficult than fixed point or integers. It just requires a simple extra step such as computing `mantissa * radix ^ exponent`. Don't think about the floating point format overall or the exponent while converting the mantissa. Just understand the mantissa and deal with it separately. This is a classic example of *divide and conquer*, which is the whole point of abstraction.

### 14.19.3 A Simple Floating Point Format

Suppose a 32-bit floating point format has a 24-bit two's complement mantissa, an 8-bit two's complement exponent, and a radix of 2. The general structure is:

mantissa * $2^{\text{exponent}}$

The binary format is as follows, labeling bits starting with 0 on the right as we would with an unsigned integer:

```
 31                    8  7         0
+--------------------------------+
|       mantissa       | exponent |
+--------------------------------+
```

We do not need to store the radix, since it never changes and can just be assumed by the hardware design.

1. What is the value of the following number?

   ```
   000000000000000000010010 11111100
   ```

   The mantissa is 000000000000000000010010, or +(2 + 16) = $+18_{10}$.

   The exponent is 11111100 = -(00000011 + 1) = -00000100 = $-4_{10}$.

   The value is therefore +18 * $2^{-4}$, or +1.125_10.

2. What is the largest positive value we can represent in this system?

   The largest positive value will consist of the largest positive mantissa and the largest positive exponent. Since we know the range of any two's complement system from Section 14.16, this is easy to figure out.

   The largest positive mantissa is 011111111111111111111111, which in two's complement is $+2^{23}$-1 = +8388607. The largest exponent is 01111111, which in two's complement is $+2^{7}$-1 (+127).

   Hence, the largest positive value is +8388607 * $2^{+127}$ = 1.42 * $10^{45}$.

   Note that this is a *much* larger number than we can store in 1 32-bit integer. But with 32 bits, there are exactly $2^{32}$ *different* patterns of 0s and 1s possible, so we cannot represent more than $2^{32}$ different numbers.

   The obvious catch, then, is that we cannot store every integer value up to 1.42 * $10^{45}$. So what is the second largest positive value? What is the difference between the largest and second largest? This would be the second largest mantissa and the largest exponent, so +8388606 * $2^{+127}$. This value is equal to the largest value minus $2^{127}$. As we can see, the larger the exponent, the more spread out are the consecutive values that our system can represent.

3. What is the smallest positive value? For an integer system, this is always 1. To find the smallest positive value in the form mantissa * $\text{radix}^{\text{exponent}}$, we choose the smallest positive mantissa, and the smallest *negative* exponent (the negative exponent with the largest magnitude).

   Since the mantissa is an integer, the smallest positive value possible is 1.

   Since the exponent is an 8-bit two's complement value, the smallest negative exponent is $10000000_2$, of $-2^{7}$ = -128.

   Hence the smallest positive value is 1 x $2^{-128}$, or 2.93873587706 x $10^{-39}$. The next smallest would be this value + $10^{-39}$. So we can see that consecutive values close to 0 are very close together.

4. What is the precision of the system?

   The precision of any system is the maximum number of significant figures, and is determined by the mantissa since the exponent only adds leading or trailing 0s to the value (i.e. shifts the digits in the mantissa left or right). The precision is therefore 23 bits, which is the number of bits that contribute to the magnitude of a two's complement integer.

   **Note** A 32-bit integer is significantly more precise than a 32-bit floating point value, since floating point sacrifices some bits of precision to increase range and allow for fractional values.

5. Represent -2.75 in this floating point system.

(a) Convert the number to fixed point binary using the methods described in previous sections:
$-2.75_{10} = -(10.11_2)$

(b) Multiply by $radix^{exponent}$ equal to 1 so we have all the components:
$-2.75_{10} = -(10.11_2) * 2^0$

(c) Shift the binary point to make the mantissa a whole number: $-(1011_2)$
By moving the binary point two places to the right, we multiplied the mantissa by $2^2$. We therefore must divide the other factor ($2^{exponent}$) by $2^2$ in order to preserve the overall value. This is is just a matter of subtracting 2 from the exponent.
$-10.11_2 * 2^0 = -(1011_2) * 2^{-2}$

(d) Convert the mantissa and exponent into the specified formats. The mantissa in this case is 24-bit two's complement and the exponent is 8-bit two's complement.
Mantissa: -(000000000000000000001011) = 111111111111111111110101
Exponent: $-2_{10}$ = 11111110

(e) Place the mantissa and exponent in their positions in the 32-bit floating point format:
Binary representation = 11111111111111111111010111111110

### 14.19.4 Overflow and Underflow

*Overflow* occurs when the result of a floating point operation is larger than the largest positive value, or smaller than the smallest negative value. In other words, the magnitude is too large to represent.

*Underflow* occurs when the result of a floating point operation is between the smallest possible positive value and the largest possible negative value. In other words, the magnitude is too small to represent.

The example 32-bit format above cannot represent values larger than $1.42 * 10^{45}$ or smaller than $2.93873587706 * 10^{-39}$.

One technique to avoid overflow and underflow is to alternate operations that increase and decrease intermediate results. Rather than do all the multiplications (by values greater than 1) first, which could cause overflow, or all the divisions first, which could cause underflow, we could alternate multiplications and divisions to moderate the results along the way. Techniques like these must often be used in scientific calculations.

Using 4-bit unsigned integers as a simpler example of the concept, if we try to compute 4 * 4 / 2, we will encounter an overflow because 4 * 4 is larger than the maximum value of 15 that we can represent in 4-bits. If, on the other hand, we compute it as 4 / 2 * 4, we get the correct result. The same principle applies to all systems with limited range.

### 14.19.5 Cost of Floating Point

Everything has a cost. The increased range and the ability to represent non-whole numbers is no exception.

#### Precision

There are only $2^{32}$ possible patterns of 32 0s and 1s. Hence, there are at most $2^{32}$ unique numbers that we can represent with 32 bits, regardless of the format.

So how is it that we can represent numbers up to $10^{45}$?

Obviously, we must be sacrificing something in between. What floating point does for us is spread out the limited number of binary patterns we have available to cover a larger range of numbers. The larger the exponent, the larger the gap between consecutive numbers that we can accurately represent.

The gap between consecutive numbers that we can represent gets larger as the exponent grows.

The precision of a 32-bit floating point value is less than the precision of a 32-bit integer. By using 8 bits for the exponent, we sacrifice those 8 bits of precision. Hence, our example format has the same precision as a 24-bit signed integer system.

#### Performance

Arithmetic on floating point is several times slower than on integers. This is an inherent property of the more complex format.

Consider the process of adding two scientific notation values:

1. Equalize the exponents
2. Add the mantissas
3. Normalize the result

Each of these operations take roughly the same amount of time in a computer as a single integer addition. Since floating point is stored like scientific notation, it uses a similar process and we can expect floating point addition to take about three times as long as integer addition. In reality, a typical PC takes about 2.5 times as long to run a loop doing mostly floating point arithmetic as it does to do the same loop with integers.

Note that this applies only to operations that can be carried out using either a single integer instruction or a single floating point instruction. For example, suppose a program is running on a 32-bit computer, and the data may be outside the range of a 32-bit integer. In this case, multiple integer instructions will be necessary to process integer values of more than 64 bits, and much of the speed advantage of integers is lost.

If hardware has floating point support built-in, then common operations like floating point addition, subtraction, etc. can each be handled by a single instruction. If hardware doesn't have a floating point unit (common in microcontrollers), floating point operations must be handled by software routines. Hence, adding two floating point values will require dozens of integer instructions to complete instead of just one floating point instruction.

Most algorithms can be implemented using integers instead of floating point with a little thought. Use of floating point is often the result of sheer laziness. Don't use floating point just because it's intuitive. Think about whether it is really necessary. The answer is usually no.

**Example 14.8** Improving Speed with Integers

The value of PI can be estimated using a Monte Carlo simulation by generating uniform random coordinates across a square "dart board" with an inscribed circle. Since the darts are distributed in a uniform random fashion, the probability of one landing in the circle is the area of the circle divided by the area of the square. This ratio is PI/4.

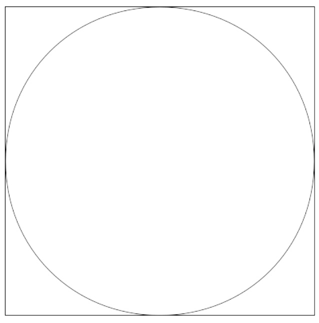


Figure 14.1: Monte Carlo Dart Board

After generating many random points, we simply divide the number within the circle by the total to get an estimate of PI/4. The intuitive approach is a use a 1 x 1 square and generate random x and y values between 0 and 1. However, this approach requires the use of floating point. if we instead make the dimensions of the square our maximum integer size (e.g. $2^{32}$-1) and generate integer x and y values instead, we can eliminate all floating point calculations from the program.
In fact, the integer implementation of this program is more than twice as fast as the floating point implementation. Full details can be found in the Research Computing User's Guide.

#### Power Consumption

Use of floating point also results in more power consumption. This is not usually noticeable on a desktop PC, but can become a problem on large grids consisting of hundreds of PCs, since the power grid they are attached to may not be designed to provide for their maximum draw. It can also be a problem when running a mobile device on battery while doing intensive computations. Battery life while doing intensive floating point computations could be a small fraction of what it is while reading email, browsing the web, or editing a document.

#### Difficult to Program

Because floating point suffers from round-off error, it is not safe to directly compare floating point values in a program. For example, the following program will fail due to the fact that 0.1 cannot be represented perfectly with a limited number of digits. Instead of stopping at 1.0, x misses the value by a tiny amount, and an infinite loop occurs.

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  x;

    for (x = 0.0; x != 1.0; x += 0.1)
        printf("%0.16f\n", x);

    return EX_OK;
}
```

Actual output:

```
0.0000000000000000
0.1000000000000000
0.2000000000000000
0.3000000000000000
0.4000000000000000
0.5000000000000000
0.6000000000000000
0.7000000000000000
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2000000000000000
1.3000000000000000
1.4000000000000001
1.5000000000000002
...
```

### 14.19.6 IEEE Floating Point Formats

The example floating point systems above are only for general understanding and context and not used in real systems. Use of integer mantissa formats suffers from problems such as multiple representations of the same value, e.g. $1 * 2^2 = 2 * 2^1$.

The IEEE floating point formats have become standard on most CPUs. They are highly optimized and contain special features, like representations for infinity and not-a-number (NaN).

S is the sign bit. Like integer formats, 1 indicates a negative number and 0 indicates positive.

F is the *fractional part* of the mantissa. The non-fractional part is not stored, and always 1, so the actual mantissa is $1.F_2$. This method provides an extra bit of precision that need not be stored, so we have a 24-bit fixed point mantissa in reality, even though

| S | E | F |
|---|---|---|
| 31 | 30-23 | 22-0 |

Table 14.12: IEEE 32-bit Format

we only store 23 bits. It also ensures that there is only 1 way to represent a given value, unlike the example formats above with integer mantissas.

The 8-bit exponent is stored in bias-127 notation, hence E is an unsigned integer with the value exponent + 127.

The absolute value of the number is 1.F x $2^{E-127}$, except for some special cases:

- If E = 255, and F is non-zero, then the value is NaN. (Not a Number)
- If E = 255, F = 0 and S = 1, then the value is -infinity.
- If E = 255, F = 0, and S = 0, then the value is +infinity.
- If E = 0, and F = 0, then the value is 0.

The floating-point hardware uses more than 23 bits for computation to avoid round-off error. This is particularly important when adding values with different exponents, since the mantissas must be offset by up to 128 bits before addition, thus producing a sum with up to 24 + 128 = 152 bits. The result is trimmed back to 24 bits after normalizing the result.

1. What is the largest positive value that can be represented in 32-bit IEEE floating point? Start by taking the largest positive mantissa and the largest positive exponent:

   $+1.11111111111111111111111_2 * 2^{+128}$

   However, any value where E=255 (exponent = 128) is a special case representing NaN or infinity, so we must take the second largest exponent:

   $+1.11111111111111111111111_2 * 2^{+127}$

   $0.11111111111111111111111_2$ is very close to 1.0, since all of the digits are at their maximum. We can therefore approximate it $0.9999_{10}$. So our value is $1.9999_{10} * 2^{+127} = 3.4028 * 10^{38}$.

   If we want the exact value of 0.11111111111111111111111, we can note that it is equal to 1.0 - the least significant bit:

   ```
        0.11111111111111111111111   2^-1 + ... + 2^-23
   +    0.00000000000000000000001   2^-23
   -----------------------------
        1.00000000000000000000000
   ```

2. What is the smallest positive value that can be represented in 32-bit IEEE floating point? Start by taking the smallest positive mantissa and the smallest negative exponent:

   $1.00000000000000000000000_2 * 2^{-127}$

   However, since this value is a special case in IEEE format used to represent 0, we take the second smallest mantissa:

   $1.00000000000000000000001_2 * 2^{-127} = 1.1754 * 10^{-38}$

3. Precision is 24 bits (23-bit F + the assumed 1), or around 6-7 decimal digits. The sign bit is separate from the mantissa, so we don't need to deduct it from the precision.

4. What is the decimal value of the following 32-bit IEEE number?

   ```
   1 10000001 10100000000000000000000
   ```

   ```
   S = 1
   E = 10000001_2 = 129 - 127 = 2
   F = 101_2
   M = 1.101_2 = 1.625_10

   Value = -1.625 * 2^2
   ```

5. How do we represent +4.25 in 32-bit IEEE?

```
                4.25_10 = 100.01_2
                        = 1.0001_2 * 2^2

                E = 2 + 127 = 129 = 10000001_2
                F = 0011
                S = 0

                0 10000001 00010000000000000000000
```

The 64-bit format is similar to the 32-bit format, but uses an 11-bit bias-1023 exponent, and a 53-bit fixed-point mantissa in the form 1.x, where only the fractional portion (52 bits) is stored.

| S | E | F |
|---|---|---|
| 63 | 62-52 | 51-0 |

Table 14.13: IEEE 64-bit Format

The range of the 64-bit format is +/- (2.2250 * $10^{-308}$ to 1.7976 * $10^{308}$)

### 14.19.7 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Define each of the following terms with respect to floating point:
   (a) Mantissa
   (b) Exponent
   (c) Radix
   (d) Normalized
   (e) Overflow
   (f) Underflow
2. What is the most common radix for floating point number systems? Why?
3. A 32-bit floating point system uses a 25-bit two's complement mantissa and a 7-bit bias-63 exponent, and a radix of 2.
   (a) What is the largest possible positive number?
   (b) What is the smallest possible positive number?
   (c) Show the value 15.25_10 in this floating point format.
   (d) What is the value of the following number in this floating point format?
   1111111111111111111111101 1000011
4. Describe two costs of using floating point in place of integers.
5. Show the decimal value of each of the following 32-bit IEEE floating point numbers:
   (a) 0 01111100 10000000000000000000000
   (b) 1 10000010 11000000000000000000000
6. Convert the following decimal values to 32-bit IEEE floating point.
   (a) -6.125_10
   (b) +0.75_10

## 14.20 Character Storage

### 14.20.1 ASCII

Characters are stored as a standardized set of binary patterns based on *ASCII (the American Standard Code for Information Interchange)*, pronounced "askee". The original ASCII set is a 7-bit binary code.

Although ASCII values are not numbers, we often use the decimal equivalent of their binary patterns to represent them for convenience rather than writing out the lengthy binary patterns.

Decimal codes 0-31 and 127 are the *control characters*. They were originally designated to control movement of the print head and scrolling on text-only printers and *teletypes* (basically typewriters that send keyboard input to a computer and print computer output on paper rather than a screen). ASCII terminals such as the vt100 and xterm replaced teletypes and use these codes to move the cursor and scroll the screen in the same way.

Below are some of the more commonly used control characters.

```
Binary      Dec Symbol  Keyboard    C       Meaning
00000000    0   NUL
00000100    4   EOT     (Ctrl+d)    \004    End of transmission
00000111    7   BEL     (Ctrl+g)    \007    Beep printer/terminal
00001000    8   BS      (Ctrl+h)    \b      Move head/cursor left
00001001    9   TAB     (Ctrl+i)    \t      Move head/cursor to next tab stop
00001010    10  LF      (Ctrl+j)    \n      Move cursor down, scroll paper up
00001100    12  FF      (Ctrl+l)    \f      Scroll to start of next page
00001101    13  CR      (Ctrl+m)    \r      Move head/cursor to left margin
01111111    127 DEL     -           \177    Delete char under cursor
```

CR+LF goes to the beginning of new line. Unix adds a CR to each LF by default since we usually don't want to go down a line without going to the beginning of the new line. Terminal-based programs need only send a newline, so a C program would write "Hello, world!\n" instead of "Hello, world!\n\r" or "Hello, world!\r\n".

Characters 32-126 are the printable characters, i.e. anything that has a font pattern (including a space character, which is just a printable character with all pixels off). Sending one of these to a terminal or teletype prints the character and moves the head/cursor to the right.

```
Binary      Decimal C
00100000    32      ' '
00100001    33      '!'
...
00110000    48      '0'
00110001    49      '1'
...
01000001    65      'A'
01000010    66      'B'
...
01100001    97      'a'     Note: Only 1 bit differs from 'A'
01100010    98      'b'
```

In C, we often use the int data type rather than char for scalar character variables to avoid promotions in algebraic expressions. C promotes char values to int in many situations, so storing as an int to begin with will make the program slightly faster.

We do not use int for character arrays (strings), since the wasted space would generally add up to more than the value of the speed gains.

```c
int     ch = 'A';
char    greeting[] = "Hello, world!\n";
```

The content of ch in the program above will be 01000001, sign-extended to the size of an int, which is usually 32 bits on modern computers, so 00000000000000000000000001000001.

**Note** The size of the C int type varies among different CPUs but always represents an integer that can be processed with a single instruction (i.e. is less than or equal to the CPU's word size). This allows the programmer to write portable code that runs at optimal speed on any CPU from a 16-bit microcontroller to a 64-bit server.

### 14.20.2 ISO

The *International Standards Organization (ISO)* extended the ASCII set to 8 bits to include non-English characters, and eventually non-Latin-based characters such as those found in many Asian languages. ISO-Latin1 is the most common character set used in the west.

In addition to European letters, the ISO character sets also add some graphic characters such as mathematical symbols, smooth border lines, etc.

### 14.20.3 Unicode

Unicode is a computing industry standard for representing the characters in most of the world's writing systems. It currently consists of over 100,000 characters from more than 90 writing scripts.

The Unicode Transformation Formats (UTF) provide ways to encode the Unicode character set using a stream of bytes. UTF-8 is the most commonly used format, and is backward-compatible with ASCII. UTF-8 encodes each of the Unicode characters using one to four bytes.

### 14.20.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is ASCII?
2. What is ISO Latin1?
3. What is unicode?

# Chapter 15

# Introduction to High-Performance Programming

## 15.1 Introduction

### 15.1.1 Two for the Price of One

You may be thinking: C *and* Fortran? At the same time? Are you insane??? I assure you, I am somewhat sane. After years of teaching computer programming and assuming that learning two languages at once would be a calamity, one day by chance I questioned this belief in a daydream.

It occurred to me that this is actually the solution to some major problems. People who only know one language don't really understand the separation between design and implementation, e.g. the concept of a loop and the syntax of a loop. They also fear learning a second language, not realizing that learning the second one is an order of magnitude easier than the first, because you already know the concepts, which are mostly the same across all languages.

Learning two languages syntaxes at once immediately clarifies the concepts underlying both of them by providing context, and alleviates many fears about future learning curves. So, I combined my notes from previous courses in C and Fortran, gave it a high-performance software spin, and here we are.

### 15.1.2 Why C and Fortran?

You may wonder why this book focuses on C and Fortran, given that your colleagues are mostly using Matlab, or Perl, or Python, or R. All of these languages are useful, but as we discuss in Section 13.5, they are generally an order of magnitude or two slower than C. They are also someone domain-specific, with Matlab being used mainly in engineering, R mainly in statistics, and Python in a variety of areas where specific tools provide a Python interface, such as machine learning.

The goal of this text is to empower the reader as broadly as possible. Given that C, C++, and Fortran are many times faster than interpreted languages, it is likely that you'll need to learn one of them anyway, when you cannot get the performance you need from other languages.

Knowing the simple combination of shell scripting and C, you can accomplish *anything*. If something cannot be done conveniently or efficiently enough in a shell script, you can create a C program and run it from your shell script. People often need to do the same when programming in Matlab, Perl, Python, R, or other interpreted languages.

Given the facts stated above, we focus on scripting and the most common and best performing compiled languages, and leave the many other programming languages to more domain-specific training.

### 15.1.3 C and C++

C and it's offshoot, C++ are the most popular compiled programming languages of recent decades.

C is a very simple, but high-level language that is renowned for giving the programmer unlimited power and flexibility, near optimal execution speed, and maximum *code density*, the amount of functionality per line of code. The creators of C deliberately made it a minimalist, though complete, high level language, by adhering to one rule: Don't add any features to the language that can be implemented by a function (subprogram). The result was a language with only about 30 keywords, no built-in input/output statements, and no mathematical functions.

Because the language is simple, it is easy for anyone to master the language itself. Most of the functionality in C programs comes from *libraries*, collections of subprograms that are not part of the C compiler, and are mostly written in C. Learning new library functions is easier than learning new language features, because the rules for calling any library function are the same. Once you know these simple rules, learning a library function is just a matter of knowing what it does, what to pass to it (the arguments) and what it sends back (the return value).

Another benefit of the simplicity of C is that the compiler can produce the fastest possible machine code. The programmers developing the C compiler itself don't have a lot of work do to making the compiler correct, and hence can spend more time on optimizing it. This also means that C programs compile very quickly compared to more complex languages. A typical C compiler can compile 10,000 lines of code in a few seconds.

Code written in C can be easily utilized by code written in other popular languages such as C++, Fortran, MATLAB, Perl, Python, and R, which makes C a good choice for writing general-purpose computational code, especially libraries. Hence, putting functionality into C libraries maximizes not only performance, but accessibility from virtually any language. Creating libraries is relatively simple and is covered in Section 20.16. Code added to a C library will never need to be duplicated, since anyone can use it from virtually any programming language. Typically, about 2/3 of all the C code I write ends up in libraries and only 1/3 is limited to a particular application.

C was first invented around 1970 and was improved in some important ways over the next few decades. It has changed very little since the 1990s, however. The fact that C is now stable also means that programs written in C will require minimal maintenance for years to come. Many other popular languages are still evolving rapidly. Some C++, MATLAB, Python, and R code written 10 years ago no longer works with the latest compilers or interpreters. Perl 6 is a drastically different language than perl 5. The changes needed are usually small, but a fair amount of expertise is required to make them. All changes to programs are time-consuming and require a new round of testing, as they almost invariably introduce new bugs.

Critics of C will often state that it is obsolete because it is not an object-oriented language. However, object-oriented programming is a *design* discipline, not a language feature. It is possible to implement object-oriented programs and any language, and in fact not at all difficult in C. The only features required for basic *OOP* (object-oriented programming) are structures and typedefs. This is discussed in more detail in Chapter 29. You can find plenty of information on the web about doing object-oriented programming in C (OOP-in-C). It will be discussed to some extent in later chapters.

Conversely, it is both possible and common to write non-object-oriented code in an object-oriented language. There exist many programmers who do not understand object-oriented design and believe that they are doing object-oriented programming simply because they are using an object-oriented language such as C++ or Java. This is a non-sequitur.

Many features of object-oriented languages, such as multiple inheritance, friend classes, and delegating copy constructors, are not essential to object-oriented design. They are more about convenience in implementation. Such features allow you to reduce code size in some situations, in exchange for a much higher learning curve and code complexity.

C++ is a superset of C, which means that C source code can be used directly in C++ programs. Unlike C, C++ is an extremely complex language which has drawn sharp criticism from some big names in computer science. (See https://en.wikipedia.org/wiki/C%2B
It contains many advanced features of questionable value, so many in fact, that very few C++ programmers fully understand the language. As a result, most C++ programmers use only a subset of its features. Different programmers use different subsets and often have a hard time understanding each others' code.

Porting C++ code from one platform or compiler to another often runs into trouble because of this complexity as well. Before deciding to implement a project in C++, ask yourself if the features of the language are really going to save you more time and effort than the complexity of the language will cost you. As most scientific software does not use complex data structures, you may be better off using plain C and a little self-discipline in cases where you want to maximize performance.

A little knowledge is a dangerous thing. This age-old adage is especially true in C++ development. In the words of Mr. Miyagi, "Karate do, or karate no do. Karate maybe so, and (sound of neck breaking)." I advise against using C++ unless you are prepared to devote the enormous amount of time required to learn it well. If you are not, then you will struggle and produce low-quality code, which is the last thing the scientific computing community needs more of. A good college student can master C in one semester. C++ will require three or four.

You may decide to learn C++ in order to utilize an existing library that is written in C++. Sometimes suitable alternatives are available to use from C or Fortran. BLAS and LAPACK can be used from any language, but Eigen, which has similar functionality requires C++. If you require a function of Eigen that is not available in any other library, then you either have to write an equivalent function yourself or learn C++.

Even if using C++ for a given project is preferable, this does not mean that you have to do all your coding in C++. If you develop your own libraries, you may choose to use C in order to maximize their accessibility from other languages, including C++, and not force others to tackle the complexities of C++ in order to use them.

This introduction will focus on C in order to help you become proficient in a compiled language as quickly as possible. Since C is a subset of C++, everything discussed here will directly benefit those who want to continue on and learn C++ eventually. Most readers will find that C is more than adequate for their needs and will be better off focusing on mastering general programming skills rather than spending that time learning the features of more complex languages.

### 15.1.4 Not Your Grandfather's Fortran

Fortran was the first widely available high level language, originally created in the 1950's. If you tell people, especially computer scientists, that you're learning Fortran, you may get some odd looks and snide remarks such as "I thought all the Fortran programmers were dead", or "Wow, I bet you had a pet mastodon when you were little".

What you're learning here is not your grandfather's Fortran, however. Fortran has gone through a number of major evolutionary steps, and is greatly enhanced since the early days. Fortran 90 brought some particularly important improvements, such as free-format (Fortran versions up to 77 requires a strict line structure) and more support for structured code. Fortran has always had intrinsic support for complex numbers, and newer versions support many matrix operations like those found in MATLAB and other higher-level tools.

Fortran is a compiled language, so well-written Fortran programs run about as fast as any program could, and nearly as fast a C in many cases.

Fortran is an open standard language, so there are many compilers available from multiple vendors. There are also free Fortran compilers for most common computer hardware and operating systems.

### 15.1.5 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are two benefits of learning two languages at once?
2. What are three benefits of the simplicity of C?
3. What is the benefit of a stable language, i.e. one that is not evolving rapidly?
4. Can we do object-oriented programming in C?
5. Is learning C a waste of time for those who need to use C++?
6. What kind of performance can you expect from well-written Fortran programs?

## 15.2 C Program Structure

The general layout of a simple C program includes the following components:

1. A block comment describing the program.
2. One or more #include directives to include *header files*, which have a filename ending in ".h". Header files add functionality that is not part of the C language itself by defining named constants such as `M_PI`, derived data types such as `FILE` and `size_t`, and the interfaces for all standard library functions.

   Header files should never contain executable C statements. Those belong in the C source files (".c" files).

3. Main program body (required):
   (a) `int main(int argc, char *argv[])`
   (b) `{`
   (c) Variable definitions/declarations + comments
   (d) Program statements + comments
   (e) A return statement
   (f) `}`

C and C++ are case-sensitive, so `PRINTF` is not a valid substitute for `printf`.

**Example 15.1** A Simple C Program

```
/***************************************************************************
 *  Description:
 *      Compute the area of a circle given the radius as input.
 *
 *  History:
 *  Date         Name        Modification
 *  2013-07-28   Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>      // Contains prototypes for printf() and scanf()
#include <math.h>       // Defines M_PI
#include <sysexits.h>   // Defines EX_OK

int     main(int argc,char *argv[])

{
    // Variable definitions for main program
    double  radius,
            area;

    // Main program statements
    printf("What is the radius of the circle? ");
    scanf("%lf", &radius);
    if ( radius >= 0 )
    {
        area = M_PI * radius * radius;
        printf("The area is %f.\n", area);
    }
    else
        fprintf(stderr,"The radius cannot be negative.\n");

    return EX_OK;
}
```

`#include` is an example of a *preprocessor directive*. `#include` inserts a *header file* into the program at the point where it appears.

Header files contain constant definitions, type definitions, and *function declarations*. Modern C function declarations are called *prototypes*, and they define the interface to the function completely. A prototype for the `printf()` function looks like this:

```
int     printf(const char *format, ...);
```

The example above contains the *function definition* for `main()`, the entry point into the program. A definition begins like a declaration/prototype, but also includes the function body (the statements that do the work of the function).

C compilers do their job in one pass through the source file, so *forward references* (references to objects that are declared or defined later) are not allowed. The compiler only needs to see a prototype for functions that is not defined before it is referenced.

The `printf()` statement in the example above is a *function call*. C programs generally contain many function calls, since they are used in lieu of language features that were deliberately left out.

C is a *free format* language, which means that the compiler treats the end of a line the same as a space or tab. The end of a variable definition or a statement is indicated by a semicolon (;).

### 15.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the major components of a C program?
2. What is a free-format language?

## 15.3 Fortran Program Structure

The general layout of a simple Fortran program includes the following components:

1. A block comment describing the program.
2. Main program body (required):
   (a) `program <program-name>`
   (b) Optional `use` statements to enable extended features (like C #includes, but more abstract)
   (c) Variable definitions/declarations + comments
   (d) Program statements + comments
   (e) `end program <program-name>`

Fortran is not case sensitive, so `end` is the same as `END` or `End`.

**Example 15.2** A Simple Fortran Program

```
!-----------------------------------------------------------------------
!   Description:
!       Compute the area of a circle given the radius as input.
!
!   Modification history:
!   Date        Name        Modification
!   2011-02-16  Jason Bacon Begin
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
! Main program body

program Circle_area
    use iso_fortran_env     ! Enable error_unit for error messages

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Constants
    real(8), parameter :: PI = 3.1415926535897932d0
    ! Variable definitions for main program
    real(8) :: radius, &
                        area

    ! Main program statements
    print *, 'What is the radius of the circle?'
    read *, radius
    if ( radius >= 0 ) then
        area = PI * radius * radius
        print *, 'The area is ', area
    else
        write(error_unit,*) 'The radius cannot be negative.'
    endif
end program
```

Fortran was originally designed as a line-oriented language, which means that the end of a line marks the end of a statement. Most newer languages, in contrast, are free-format, so that a single statement can span many lines, or multiple statements may be on the same line. Languages such as C, C++, and Java use a semicolon to mark the end of each statement, and line structure is completely ignored.

Fortran 90 introduced a more flexible source code format than previous versions, but still uses the end of a line to mark the end of a statement. If a particular statement is too long to fit on the screen, it can be continued on the next line by placing an ampersand (&) at the end of the line to be continued, the same way we use a backslash (\) in a shell script:

```
print *, 'This message is too long to fit on a single line, ', &
         'so we use the continuation character to break it up.'
```

### 15.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the components of a Fortran program?
2. What is a line-oriented language? Can a statement span more than one line in a line-oriented language?

## 15.4 The Compilation Process

As compiled languages, C, C++, and Fortran programs are translated entirely to machine language before being executed.

### 15.4.1 Compilation Stages

In all three languages, production of an executable file involves up to three steps, outlined below and in Figure 15.1.

1. Preprocessing: This step runs the source code through a stream editor called the preprocessor, which is designed specifically for editing source code. The preprocessor makes modifications such as inserting the contents of header files and replacing named constants with their values, and outputs modified source code.

   The preprocessor command is usually **cpp** ( short for C PreProcessor ).

   The preprocessor is described in detail in Section 15.7.

2. Compilation: This step translates the preprocessed source code to machine language (also known as *object code*), storing the resulting machine code in an *object file*. The object file is not a complete executable file, as certain components necessary to load and run a program have not been added yet. Object files on Unix systems have a file name extension of ".o".

3. Linking: This step combines the object files from the compilation step with other object files stored in *libraries* (precompiled collections of functions) and the machine code needed to start a program. The result is an *executable* file such as `/bin/ls` or any other Unix command.

   The linker program is usually called **ld**.

   An example of a library is `/usr/lib/libc.so`, the standard C library. It contains the object files for many standard functions used in the C language, such as `printf(), scanf(), qsort(), strcpy()`, etc.

You generally do not need to run these steps individually. They are executed automatically in sequence when you run a compiler such as **cc**, **clang**, **gcc**, or **gfortran**.

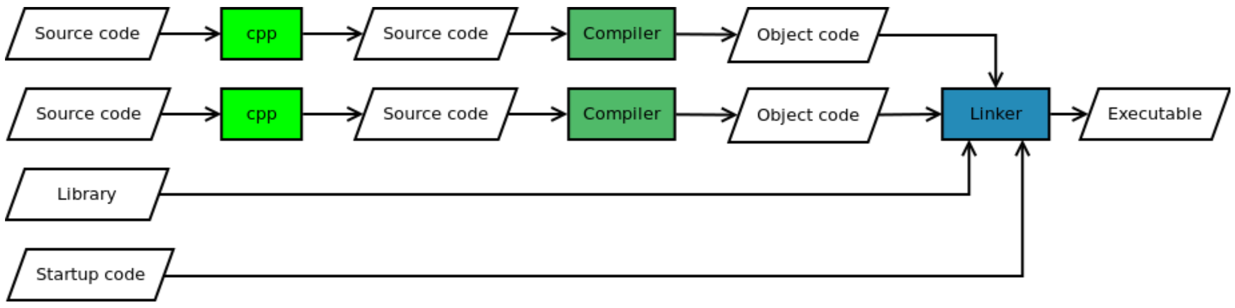


Figure 15.1: Compilation

### 15.4.2 Portability

Every Unix system with a C compiler has a **cc** command. On FreeBSD and OS X, **cc** is equivalent to **clang**. On Linux systems, **cc** is equivalent to **gcc**. On some commercial Unix systems, **cc** is a proprietary compiler developed by the vendor. Clang/LLVM and GCC are both open source compiler suites and are highly compatible with each other. They support most of the same command-line flags, such as `-Wall` to enable all possible warning messages.

Unfortunately, there is no standard compiler name for Fortran, since most Unix system don’t include a Fortran compiler. Fortran is usually added in the form of **f2c** (a Fortran 77 to C translator), **gfortran** (the open source GNU Fortran compiler), or **flang** (the open source Clang/LLVM Fortran compiler). There are also several commercial Fortran compilers available.

C source files have an extension of ".c". C++ files usually use ".cc", ".cpp", ".cxx", or ".c++".

Fortran files use ".f", ".F", ".for", or ".FOR" for Fortran 77, ".f90" or ".F90" for Fortran 90, ".f03" or ".F03" for Fortran 2003, and ".f08" or ".F08" for Fortran 2008.

Examples of building an executable file from a single source file:

```
shell-prompt: cc jumping-genes.c
shell-prompt: gfortran gauss.f90
```

The commands above will produce an *executable* file called `a.out`. This is the default for most Unix compilers. The executable file is also sometimes called a *binary* file. This is why program directories on Unix systems are named "bin" (/bin, /usr/bin, /usr/local/bin, etc.)

To run the binary program, we simply type it’s file name followed by any arguments that it requires.

```
shell-prompt: ./a.out
```

Most Unix compilers also support the `-o` flag to specify a different output file name.

```
shell-prompt: cc jumping-genes.c -o jumping-genes
shell-prompt: ./jumping-genes
shell-prompt: gfortran gauss.f90 -o gauss
shell-prompt: ./gauss
```

All Unix compilers support certain common flags. The `-g` tells the compiler to include debugging information in the binary file, so that a *debugger* (a program that helps you find problems) can determine the location of a problem in the source code while examining an executable. The `-O, -O2` and `-O3` flags tell the compiler to turn on standard levels of object code optimization.

```
shell-prompt: cc -O2 jumping-genes.c -o jumping-genes
shell-prompt: gfortran -O3 gauss.f90 -o gauss
```

You can keep your life simple by compiling with **cc**, rather than specifically using **clang** or **gcc**, and using portable flags such as `-O`. The `-Wall` flag is not entirely portable, but is supported by both **clang** and **gcc**, which are by far the most popular compilers. Compiling with `-Wall` is extremely helpful for catching potential program bugs.

Using `-O` will usually improve the speed of your executable file significantly and will often reduce its size as well. The `-O2` will usually offer only a marginal improvement over `-O` (and is actually the same with some compilers), and `-O3` will usually provide little or no benefit over `-O2`.

Higher optimization levels like `-O3` may also impede debugging, since they may reorganize the machine code in ways that make it impossible to determine which line of source code a given machine instruction came from. Generally, the higher the level of optimization, the more dangerous and less beneficial the optimizations will be. Using `-O` will include all optimizations considered to be very safe and will provide the vast majority of all the performance benefit that's possible.

You can also enable specific optimizations using other command line flags, but such flags may not work with all compilers and may produce executables that will not run on older CPUs of the same family. The `-O` flags all aim to generate portable executables that will run on any machine in the same family of processors that is likely to still be in use. For example, compiling with `-O2` on the latest AMD or Intel processor will generate an executable that should work on any recent Intel or AMD processor produced in the last several years.

In rare cases, you may see noticeably better performance by utilizing the latest processor features. Clang and GCC make this relatively easy with the `-march=native` flag:

```
shell-prompt: clang -O2 -march=native super-analyzer.c
```

For most programs, this will make very little difference in speed. In extremely rare cases, it may reduce run time by as much as 30%. The executable produced will not work on older processors, however. You also need to be using a compiler that is new enough to support all or your bleeding-edge processor's features.

Before committing to anything more sophisticated than `-O2`, compare the run time of your program when compiled with various options to see if it's really worth doing. This can be easily done using the **time** command, as discussed in Section 3.14.14.

### 15.4.3 Using Libraries

Libraries, as mentioned above, are collections of precompiled subprograms that we can use in our programs. Libraries are built with the same compilers as our programs (cc, c++, gfortran, flang, etc). We can create our own libraries as described in Chapter 20. More often, we will use libraries supplied with the compiler or installed via a package manager.

While all languages use libraries, the C language was intentionally designed to rely heavily on them. The C language designers decided not to give the language any features that could be implemented as a library function. This keeps the language very simple, fast, easy to learn, and easy to implement on new hardware. In some cases it makes the program a bit less elegant, but no harder to read in reality.

For example, to compare two strings in many languages, we might write something like the following:

```
if ( string1 == string2 )
```

The C language does not directly support string comparison, so for this we use a library function call:

```
if ( strcmp(string1, string2) == 0 )
```

Some libraries, such as the standard C library (usually `/usr/lib/libc.so`) are automatically searched by the linker.

For other libraries, such as the standard math library (usually `/usr/lib/libm.so`) we need to tell the linker to search it, by using the `-l` flag. This flag is immediately followed by the unique portion of the library's file name. For example, to use `libm.so`, we specify `-lm`. To use `liblzma.so`, we would specify `-llzma`.

```
cc -O gauss.c -o gauss -lm -llzma
```

All library file names begin with "lib" and end with common extensions like ".a", ".so", or ".dylib". We omit these parts when using the `-l` flag.

Add-on libraries, such as those installed by a package manager, may not be in the linker's default search path, so we also need to use `-L` to tell the linker where to find the library file. This flag is immediately followed by the absolute or relative path of the directory containing the library. For example, to use `/usr/local/lib/libblas.a`, we would use a compile command like the following:

```
cc -O gauss.c -o gauss -L/usr/local/lib -lblas -lm
gfortran -O gauss.f90 -o gauss -L/usr/local/lib -lblas -lm
```

**Note** Order may be important with `-l` flags. For example, if a function in the blas library calls a function in the standard math library, then `-lm` should come *after* `-lblas`.

### 15.4.4 C++ and Fortran Compilation

The compilation process for C++ or Fortran is largely the same as for C. C++ also uses a preprocessor stage. Preprocessing was not part of the original Fortran language, but it has been adopted from C. Fortran compilers can be told to use the C preprocessor by specifying command-line options such as `-cpp` or by choosing an appropriate filename extension such as `.fpp`.

C, C++ and Fortran object files are slightly different from each other, but can be linked together to form executables from multiple languages. It is actually quite common for C++ programs to use C libraries, and for C/C++ programs to use Fortran libraries such as BLAS.

C++ programs can be compiled on any system using the **c++** command, which is equivalent to **clang++** on FreeBSD and macOS, and to **g++** on GNU/Linux systems. There is rarely a reason to invoked **clang++** or **g++** explicitly.

```
# C++ program using Fortran BLAS library and C math library
shell-prompt: c++ super-analyzer.cxx -L/usr/local/lib -lblas -lm
```

The most stable Fortran compiler is **gfortran**. The **Flang** project aims to develop an open source Fortran compiler companion to **clang** and **clang++**, but is still a work-in-progress at the time of this writing. Commercial Fortran compilers also exist. At the time of this writing, Fortran compilers are not as compatible with each other as are **clang** and **gcc**, so invoking **gfortran** directly may be the best option.

### 15.4.5 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the three stages in C, C++, and Fortran compilation?
2. What is the portable way to invoke a C, C++, and Fortran compiler? Contrast to the non-portable ways.
3. What is the risk of using optimizations like `-march=native`?
4. Show a portable command that compiles the program `find-waves.c` to an executable called `find-waves`. The program uses functions from the C math library. Use the best safe and portable optimizations.
5. Show a compile command that uses GNU Fortran to build `find-waves` from `find-waves.f90`. The Fortran version of this program uses the library `/usr/local/lib/libblas.so`. Use the best safe and portable optimizations.

## 15.5 Comments

### 15.5.1 Why do we Need Comments?

Comments are critical for making computer programs readable by humans. Intelligent programmers do their best to make the code *self-documenting*, by using descriptive variable names and writing non-cryptic code instead of showing off their cleverness. Self-documenting code requires far fewer comments.

```
# Self-documenting Bourne shell code written by a wise programmer
printf "Please enter your name: "
read name
if [ $name = "Bob" ]; then
    printf "Hi, Bob."
fi

# Cryptic code intended to show off how clever I am, or just to avoid
# typing because I'm lazy and short-sighted
read n; test $n = "Bob" && printf "Hi, Bob."
```

Regardless how well-written a program is, some comments will be needed. Keep in mind that a computer program is a way to explain something to a computer, which requires a different approach and far more detail than human beings are used to dealing with. Hence, it's impossible to make the code itself completely self-documenting. Comments are a supplement to help humans understand a set of instructions aimed at a machine.

In ANSI C and C++, a comment is everything from a `//` to the end of the line, or everything between `/*` and `*/`, even if it spans lines.

In Fortran 90 and later, a *comment* is everything from a '!' character to the end of the line.

Well-written programs are typically around 1/3 comments, 2/3 code, but this varies widely depending on the complexity of the program. Simple programs may require fewer comments, and complex programs may require more comments than code. Blank lines are a form of comment as well. Separating logical blocks of code, each of which performs a different function, helps document the structure of the program.

Good comments build good karma. You will be happy with yourself later, after your short-term memory about the code you are writing has faded, if there are comments that help you remember how it works. Others who did not write the code, but have to work on it, will also be happy with you.

### 15.5.2 Line Comments

Line comments document one to a few lines of code, and may appear above the code or to the right of it. If above the code, they should not be separated from the code they document with a blank line, but should be separated from the code above. If next to the code, they should be indented consistently with other nearby line comments to make the code easy to read.

```
double  radius,     // Radius of the circle
        area;       // Area of the circle

// Input the radius
printf("What is the radius of the circle? ");
scanf("%lf", &amp;radius);
```

```
use ISO_FORTRAN_ENV     ! Enable INPUT_UNIT, OUTPUT_UNIT, ERROR_UNIT, etc.

! Disable implicit definitions (i-n rule)
implicit none
```

### 15.5.3 Block Comments

Block comments are multi-line comments that document a section of code, such as a whole subprogram, a loop, or just a block of code that does a particular task. They are formatted nicely and consistently so that they stand out, and are separated from the other code by blank lines above and below.

```
/*
 *  Description:
 *      Compute the area of a circle given the radius as input.
 */
```

```
!------------------------------------------------------------------------
!   Description:
!       Compute the area of a circle given the radius as input.
!------------------------------------------------------------------------
```

### 15.5.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the purpose of comments?
2. Can we eliminate the need for comments by writing good code?
3. How can we reduce the need for comments?
4. What will happen if we don't comment code as we write it?

## 15.6 Strings

A *string* is any sequence of characters such as letters, digits, spaces, punctuation symbols, and control characters such as tabs, carriage returns, and newlines. Most programming languages support the use of *string constants*, which consist of literal characters between quotes.

C uses double quotes to enclose a string:

```
"What is the radius of the circle?"
```

Fortran uses single quotes to enclose a string:

```
'What is the radius of the circle?'
```

In C, we can also enclose a single character between single quotes:

```
putchar('x');
```

A character between single quotes in C is not a string, however. It is a single character, whereas a string is an array of characters. They are different data types that are not interchangeable.

## 15.7 The C Preprocessor

The C preprocessor, also used with C++ and optionally with Fortran, is a stream editor specially designed to modify source code. It recognizes language *tokens*, such as variable names, numeric constants, strings, etc.

Preprocessing was added to Fortran by developers who found the features useful in C. The Fortran preprocessor is based on the C preprocessor and is actually the same program in some cases.

Preprocessor actions are designated by *directives*, which are indicated by a '#' as the first character on the line. The preprocessor is line-oriented, unlike the C and C++ compilers, which are completely free-format.

### 15.7.1 #define

The C preprocessor is used to define named constants. The names are simply replaced by their value with a simple text substitution.

The `#define` directive is followed by an identifier that must follow the same naming conventions as a C variable, i.e. it must begin with a letter or underscore, and subsequent characters can be letters, underscores, or digits.

**Note** Constant names defined with `#define` typically use all capital letters, so that they can be easily distinguished from variables where they are used in statements.

```
#define PI              3.14159265358979323846
#define RADIUS_PROMPT   "Please enter the radius: "
```

Constants are actually the simplest case of what `#define` can be used for. It can also be used to define *macros*, which are discussed in Section 20.14.

### 15.7.2 #include

The `#include` directive inserts the contents of another source file into the stream at the point where the `#include` appears.

The files included are known as *header files*. The use a file name extension of ".h" for C and ".hpp" for C++. They contain things like constant definitions, type definitions, and prototypes, which may be used by many other source files. *Factoring* out commonly used code in this way eliminates redundancy and greatly reduces the maintenance cost of the source code.

Suppose we have the following C program:

```
#include "constants.h"

int     main()

{
    printf("PI = %f and Avogadro's constant is %f.\n", PI, AVOGADRO);

    return 0;
}
```

The file "constants.h" contains the following:

```
#define PI          3.141592653589
#define AVOGADRO    6.02e23
```

The output of the C preprocessor, which is passed on to the compiler, will then be the following:

```
int     main()

{
    printf("PI = %f and Avogadro's constant is %f.\n", 3.141592653589, 6.02e23);

    return 0;
}
```

**Caution** It is widely regarded as a very bad practice to place any executable statements in header files. Doing so can lead to the same code being defined in multiple places in the same program. All statements should be in a ".c" file and headers should be used only to define constants, macros, and new data types.

Header files provided by the system or installed globally with libraries have their names enclosed in angle brackets. The preprocessor looks for these headers in `/usr/include` by default.

For example, to include `/usr/include/stdio.h` and `/usr/include/sys/types.h` in your program, simply use the following:

```
#include <stdio.h>
#include <sys/types.h>
```

```
cc myprog.c -o myprog
```

The `-I` flag indicates additional directories to check for headers. The path indicated in `#include` is relative to the prefix specified with `-I`.

For example, if you are using vector functions from the GNU Scientific Library (GSL) and the GSL headers are installed in `/usr/local/include/gsl`, then you could do the following:

```
#include <gsl/gsl_vector.h>
```

```
cc -I/usr/local/include myprog.c -o myprog
```

Alternatively, you could use the following:

```
#include <gsl_vector.h>
```

```
cc -I/usr/local/include/gsl myprog.c -o myprog
```

Header files that are part of the project and reside in the same directory as the code being compiled are enclosed in double quotes.

```
#include "constants.h"
```

C++ compilers use the exact same directives and compiler flags. The **gfortran** compiler will use them as well, as long as it is told to use the preprocessor by specifying `-cpp` or by using an appropriate filename extension such as ".fpp" or ".F90" instead of ".f90".

### 15.7.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How does the C preprocessor differ from other stream editors, such as **sed**?
2. What languages use the preprocessor?
3. Show a preprocessor directive that defines a constant named HOME_PLANET with the value "Earth".
4. What is the output of the preprocessor for the following code:

```
// planets.c
#include "planets.h"

int     main()

{
    printf("My home planet is %s.\n", HOME_PLANET);

    return 0;
}

// planets.h
#define HOME_PLANET "Earth"
```

5. What kind of code should and should not be placed in header files?
6. Show a command for compiling the following program, `prog1.c`, to `prog1`, if `stdio.h` is a standard header file in `/usr/include` and `xtend/math.h` represents `/usr/local/include/xtend/math.h`.

```
#include <stdio.h>
#include <xtend/math.h>

int     main()

{
    return 0;
}
```

## 15.8 The Basics of Variable Definitions

A *variable* in a computer program is a name for a memory location, where we can store information such as integers, real numbers, and characters. See Section 12.4 for an explanation of computer memory. This differs from variables in algebra, which are simply abstract representations of unknown values. Be sure not to confuse the two.

A *Variable definition* allocates one or more memory cells for storing data, and *binds* the data type and a variable name to that memory address. High-level language programmers generally don't know or care what the actual memory address is. We just use the variable names to refer to all data stored in memory.

**Note** In C, a *definition* and a *declaration* are not the same thing. The latter does not allocate space, but only alludes to something defined elsewhere, to provide information about data types. This will be clarified later when we discuss various types of definitions and declarations. For now, just keep the two terms separated in your head. To avoid confusion, we will adhere to these terms throughout this text when discussing any language.

Variable definitions in C:

```
// Variable definitions for the main program
double  radius, area;
```

Variable definitions in Fortran:

```
! Variable definitions for main program
double precision :: radius, area
```

**Caution**

The original Fortran language included a rule known as the *I through N rule*, which allowed programmers to omit variable definitions, and allow the compiler to define them implicitly. The rule states that if a variable is not defined explicitly, and the variable's name begins with any letter from I to N, then it is an integer. Otherwise, it is real number.

This rule may seem insane in today's world, but it was created in the days when programs were stored on paper punch cards and each line of code meant another punch card, so eliminating variable definitions saved a lot of work and paper. On today's computers, there is no reason to keep such a rule, but nevertheless, Fortran has maintained it so that old code will still compile.

If you forget to define a variable explicitly, the Fortran compiler will quietly define it for you using the I-N rule, and often not with the data type you need.

The I-N rule should *always* be disabled in new Fortran code by adding the line

```
implicit none
```

above the variable definitions in every subprogram, including the main program. This will prevent variables from being accidentally defined with the wrong type.

Variable definitions are discussed in greater detail in Section 16.4.

### 15.8.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a variable in a computer program?
2. What does a variable definition do?
3. How does Fortran's I-N rule work?
4. How can we prevent the I-N rule from causing problems?

## 15.9 Program Statements

A *statement* is anything in a program that actually performs some sort of work at run-time. It could compute a value or perform input or output. Variable definitions are not statements, not are preprocessor directives.

### 15.9.1 C Statements

In C, a statement is any expression followed by a semicolon. All of the following are valid statements as far as a C compiler is concerned:

```
area = M_PI * radius * radius;
printf("The area is %f.\n", area);
5;
area + 2;
```

The last two statements above don't accomplish anything, of course, but are nevertheless perfectly valid. The C compiler may issue a warning about a statement with no effect, but the program will compile and run.

Part of the design philosophy behind the C language was "trust the programmer". Hence, C enforces a minimal number of rules. A C compiler is not bogged down with code to forbid things in order to prevent you from shooting yourself in the foot. Trusting the programmer makes it easier for the programmer to do certain things that the compiler-writer might not have anticipated. Most languages that came before C, and some that came after, are considered somewhat obstructive by many people.

The absence of draconian rules makes the C compiler much simpler and faster. Programmers tend to learn from their mistakes anyway, so it isn't necessary for the compiler to contain extra code to police them.

Every expression in a C program has a value, including the call to the printf() function, which returns the number of characters printed, and the assignment statement above, which has a value of the number assigned. We are simply free to ignore the value if we choose. The fact that an assignment statement is an expression with a value allows us to string assignments together:

```
a = b = c = 10;
```

The expression `c = 10` has a value of 10, which is assigned to b, and so on. It is as if we had written the following:

```
a = (b = (c = 10));
```

Fortran does not allow this, since an assignment statement has a specific syntax of:

```
value = expression
```

### 15.9.2 Fortran Statements

Unlike C, where a statement is any expression followed by a semicolon, there are two distinct types of statements in Fortran, described below.

#### Subroutine Calls

A subroutine call statement jumps to another block of code, which could be part of the program, part of a *library* of subprograms stored in another file, or could be intrinsic to (built into) the Fortran language. Some commonly used intrinsic subroutines include:

- read: Inputs data from the standard input or any other input stream.
- write: Outputs data to the standard output, or any other specified output stream.
- print: Outputs data to the standard output stream. The print statement is a shorthand for write.

```
write (*, *) 'What is the radius of the circle?'
print *, 'What is the radius of the circle?'
```

#### Assignment Statements

An assignment statement assigns a new value to a variable, overwriting what was previously contained at that memory address.

```
area = PI * radius * radius
```

Both kinds of statements can contain expressions, but neither actually *is* an expression in Fortran.

### 15.9.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a statement in C?
2. What is a statement in Fortran?

## 15.10 Fortran Modules

Fortran modules are used to define things for use in more than one subprogram (including the main program). Each module has a name, and a subprogram can gain access to items defined in that module with a `use` statement.

```
module constants
    ! Define ONLY constants here, not variables!
    ! (use the 'parameter' keyword)
    double precision, parameter :: &
        PI = 3.1415926535897932d0, &
        E = 2.7182818284590452d0, &
        TOLERANCE = 0.00000000001d0, &  ! For numerical methods
        AVOGADRO = 6.0221415d23         ! Not known to more digits than this
end module constants

! Main program body
program example
    use constants           ! Constants defined above

    ...
end program
```

**Caution**
Modules should be used only to define constants, not variables. Hence, each definition should include the `parameter` modifier.
Defining variables that can be modified by more than one subprogram will cause *side effects*. A side effect occurs when one subprogram modifies a variable and that change impacts the behavior of another subprogram. Issues caused by side effects are extremely difficult to debug, because it is impossible to tell by looking at a subprogram call what side effects it may cause. Hence, subprograms should only modify the values of variables that are passed to it as arguments, since these are visible in the call.
Defining constants in a module this way provides an alternative to using `#define` and the C preprocessor, which is non-standard in Fortran programming.

### 15.10.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What do Fortran modules do?
2. What kinds of things should and should not be defined in a module? Explain.

## 15.11 C Standard Libraries and Fortran Intrinsic Functions

All programming languages provide access to a large collection of subprograms for common tasks such as computing square roots and other mathematical functions, performing input and output, sorting lists, etc. The only difference is in how this functionality is provided. An *intrinsic*, or built-in function is a function that is recognized by the compiler. Functions can also be provided by libraries that are separate from the compiler.

### 15.11.1 C

Unlike many languages, C has *no* intrinsic functions. All functions in C are provided by libraries, separate from the compiler. The interface (required arguments and return values) of common functions such as `printf()`, `sin()`, `cos()`, `exp()`, etc. are specified in header files such as `stdio.h` and `math.h` and the functions themselves are provided by precompiled libraries such as `libc.so` and `libm.so`.

The function interface comes in the form of a *prototype*, which simply states the return type and the type of each argument the function requires:

```
shell-prompt: fgrep 'sin(' /usr/include/math.h
double  asin(double);
double  sin(double);
```

To use `printf()` in a C program, we must add the line `#include <stdio.h>`. To use `sin()` and other math functions, we must add `#include <math.h>` and use `-lm` to link with the standard math library.

To find out which header files and/or libraries are required for a given C library function, and what the interface looks like, simply check the man page:

```
shell-prompt: man sin
SIN(3)                 FreeBSD Library Functions Manual                SIN(3)

NAME
     sin, sinf, sinl sine functions

LIBRARY
```

```
     Math Library (libm, -lm)

SYNOPSIS
     #include <math.h>

     double
     sin(double x);

     float
     sinf(float x);

     long double
     sinl(long double x);

DESCRIPTION
     The sin(), sinf(), and sinl() functions compute the sine of x (measured
     in radians).  A large magnitude argument may yield a result with little
     or no significance.

RETURN VALUES
     The sin(), sinf(), and sinl() functions return the sine value.

SEE ALSO
     acos(3), asin(3), atan(3), atan2(3), cos(3), cosh(3), csin(3), math(3),
     sinh(3), tan(3), tanh(3)

STANDARDS
     These functions conform to ISO/IEC 9899:1999
```

**Note**

One of the best ways to become a great C programmer is by browsing the header files. This is how you discover all the cool features that are available to make your life easier. As a new C programmer, there will be things in the header files that you don't understand. Ignore them if you don't have time to explore them right now, and just take in what comes easily for now.

```
shell-prompt: more /usr/include/math.h

[scroll past stuff that's Greek to you]

#define M_E             2.7182818284590452354   /* e */
#define M_LOG2E         1.4426950408889634074   /* log 2e */
#define M_LOG10E        0.43429448190325182765  /* log 10e */
#define M_LN2           0.69314718055994530942  /* log e2 */
#define M_LN10          2.30258509299404568402  /* log e10 */
#define M_PI            3.14159265358979323846  /* pi */
#define M_PI_2          1.57079632679489661923  /* pi/2 */
#define M_PI_4          0.78539816339744830962  /* pi/4 */
#define M_1_PI          0.31830988618379067154  /* 1/pi */
#define M_2_PI          0.63661977236758134308  /* 2/pi */
#define M_2_SQRTPI      1.12837916709551257390  /* 2/sqrt(pi) */
#define M_SQRT2         1.41421356237309504880  /* sqrt(2) */
#define M_SQRT1_2       0.70710678118654752440  /* 1/sqrt(2) */

#define MAXFLOAT        ((float)3.40282346638528860e+38)

[scroll down more]

int     isinf(double);
int     isnan(double);

double  acos(double);
double  asin(double);
double  atan(double);
double  atan2(double, double);
double  cos(double);
double  sin(double);
double  tan(double);

double  cosh(double);
double  sinh(double);
double  tanh(double);

double  exp(double);
double  frexp(double, int *);   /* fundamentally !__pure2 */
double  ldexp(double, int);
double  log(double);
double  log10(double);
double  modf(double, double *); /* fundamentally !__pure2 */

double  pow(double, double);
double  sqrt(double);

double  ceil(double);
double  fabs(double) __pure2;
double  floor(double);
double  fmod(double, double);

[and much more...]
```

Functions are subprograms which are called by using them within an expression, and for the most part, look like they would in any algebraic expression. The general form is

```
function-name(argument [, argument ...])
```

An *argument* is any value that we pass to the function, such as the angle required by the sine, cosine, and tangent functions. The function call itself is an expression that has the value returned by the function. For example, the value of the expression `sin(M_PI)` in a C program is 0.0, since the sine of any multiple of Pi is 0.

```
#include <stdio.h>  // For printf()
#include <math.h>   // For sin() and M_PI

int     main()

{
    printf("The sine of %f is %f\n", M_PI, sin(M_PI));

    return 0;
}
```

In the code above, `M_PI` is an argument to the `sin()` function, and `"The sine of %f is %f\n"` and `sin(M_PI)` are arguments to `printf()`.

To compile this code, we can use the following:

```
cc -O sine-pi.c -o sine-pi -lm
```

### 15.11.2 Fortran

Fortran provides a wealth of intrinsic mathematical functions for computing trigonometry functions, logarithms, etc. The complete list of functions is too extensive to list here, but it includes all the standard trigonometric functions, square root, etc.

```
module constants
    double precision, parameter :: PI = 3.1415926535897932d0
end module constants

program sine_pi

    print *, 'The sine of ', PI, ' is ', sin(PI), '.'

end program
```

Fortran can also utilize addition functions from external libraries, as C does.

### 15.11.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is an intrinsic function?
2. How many intrinsic (built-in) functions does the C language have? Elaborate.
3. What is a prototype in C?
4. Where do we find prototypes for C standard library functions such as sin()?
5. How would we find out which header files and libraries are required for the sqrt() function?
6. What is an argument?

7. What is a return value?
8. How do we call a function?
9. How many intrinsic functions for Fortran have?
10. Is Fortran limited to intrinsic functions only?

## 15.12 Code Quality

A program that works (produces correct output) is not necessarily a good program. In fact, most programs that appear to work fine are actually pretty crappy in most respects. There are several objective and subjective measures of good code. Most code quality measures and advice are specific to particular topics in programming and are therefore discussed throughout the text alongside the relevant topic. Below is a brief introduction to some of the basic concepts.

There are tools available to help you check code quality. The GCC and Clang compilers both provide a `-Wall` flag to issue as many warnings as possible about potential issues during compilation.

Most Unix systems provide the **lint** command for further checking C code for potential issues. ( The command is so named because it picks the "lint" off your source code. ) There are similar source code analysis tools for most other languages as well.

- Readability is a somewhat subjective measure of how easy it is for people to understand the code. Readability is determined by several factors such as comments that explain *why* the code is doing what it's doing, variable names that make it clear what a variable contains, etc.

  Good code format involves consistent indentation (which is actually required by Python), and spacing (blank lines) to separate logical sections of code that perform separate tasks.
- Speed is an objective measure that refers to how long the program takes to perform a task. Naturally, we want most programs to run as fast as possible. Tips for speeding up code are presented throughout the text.
- Resource requirements are another objective measure that refers to how much memory and disk space a program requires to run. Programs that use more than they should need to in theory are referred to as *bloatware*.
- Robustness is an objective measure that refers to the ability of a program to gracefully handle user errors and other bad input. The worst case with any program is incorrect output. This should simply never happen. The next worst is crashing. The best case is a program that detects all bad input and takes appropriate action such as telling the user exactly what they need to do to correct it. The minimum any program should do is indicate where the error occurred, and then terminate. This will at least save the user from having to waste time hunting down the problem.

Most existing programs could be made to run much faster and use far fewer resources. Most programmers are content if the program works and is fast enough for their purposes on their hardware. This is sometimes OK, but it can create problems when someone wants to run the program with bigger inputs or on a computer with less memory or a slower CPU. Hence, it's best to try to write the fastest code possible and minimize resources used every time you write new code.

### 15.12.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Is a program that produces correct output a high-quality program?
2. How can we request help locating problems in our code from compilers like **clang** and **gcc**?
3. What other tool might find problems that were missed by the compiler?
4. How do we make our code readable?
5. How fast should we try to make our programs? Why?
6. How much memory should our programs use? Why?
7. What does a robust program do?

## 15.13 Performance

At this point in the book, there isn't a whole lot to say about performance, except that you should try to maximize it at all times. While a program may be "fast enough" for your current needs on your current computer, you may later need to run it using much larger inputs or on a slower computer with less memory.

Maximizing performance is mainly a matter of choosing the best algorithms and the fastest compiled programming language. Generally, shorter code is faster code, so set out from the start to learn how to write the most concise and efficient code possible. There are also some additional tricks we can use if we understand how computer hardware works. Some are portable and some are not. These tricks will be covered in the appropriate context throughout the remaining chapters.

### 15.13.1 Polynomial Factoring

Multiplication is far more expensive than addition. We can sometimes alter code to reduce the number of multiplication operations, such as by factoring polynomials:

```
// 6 multiplications, some of which are redundant
y = 3.0 * x * x * x + 2.0 * x * x + 1.0 * x + 5.0;

// Only 4 multiplications after factoring out x
y = x * (3.0 * x * x + 2.0 * x + 1.0) + 5.0;

// Only 3 multiplications after factoring out another x
y = x * (x * (3.0 * x + 2.0) + 1.0) + 5.0;
```

### 15.13.2 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. How do we maximize performance of a program?

# Chapter 16

# Data Types

## 16.1 Choosing a Data Type

No matter what programming language you use, the data types chosen by the program will have a profound effect on performance, memory use, and correctness of the output. Many languages, such as Matlab, are *typeless*, which means we do not control that data types of our variables. Instead, they choose a type that is likely (but not guaranteed) to have enough range and precision. This often results in a performance penalty and/or greater memory use, as they end up playing it safe to minimize the chance of overflow and round-off error.

C, C++, and Fortran give us full control over the data types of our variables and constants. Choosing data types is important in order to maximize the speed of your programs, minimize memory use, and ensure correct results. The approach to choosing a data type is basically the same in any programming language. We simply need to understand the limitations of the data types offered by the language and how they are supported by typical hardware.

Choosing the best data type always involves understanding the needs of the specific code you are working on. There are, however a few general ideas that can be applied. It will often depend on whether the variable is a *scalar* (dimensionless, holds only one value) or a large *array* (at least one dimension such as a vector or matrix, holds multiple values). This determines whether we need to consider how much memory the variable uses.

Some general guidelines to apply to each specific situation:

1. Use integers instead of floating point whenever possible. They are faster and more precise. Recall from Section 14.19 that floating point numbers are stored in a format like scientific notation and hence floating point addition takes about three times as long as integer addition. Section 33.2.1 shows an example program that runs about 2.5 times as fast when implemented with integers than when using floating point.

   Integers are more precise because all the bits are used to represent digits, whereas a floating point value with the same number of bits uses some of them for the exponent, which does not contribute to precision. Only the bits used for the mantissa provide significant figures in our real numbers.

   We can often eliminate the need for floating point by simply choosing smaller units of measurement, so that we no longer have fractions in our data. For example, specifying monetary amounts in cents rather than dollars, or internally representing probabilities as values from 0 to 100 rather than 0 to 1, allows us to use integers instead of real numbers.

2. If you must use floating point, use a 64-bit floating point value unless you need to conserve memory use (i.e. your program uses large arrays or other in-memory data structures) *and* you do not need much precision. Modern computers do not take significantly longer to process 64-bit floating point values than they do to process 32-bit floating point values.

   Keep in mind that the accuracy of your results is usually less than the accuracy you start with, since round-off error accumulates during calculations. 32-bit floating point operations are only accurate to 6 or 7 decimal digits, so results will likely be less accurate than that. 64-bit floating point has up to 16 decimal digits of precision.

3. Make use of complex data types (imaginary numbers) where they can be helpful. Allowing computations to go into the complex plane may simplify your code. For example, there is no need to check for a negative discriminant when

computing roots using the quadratic formula using complex numbers, since it is possible to compute the square root of a negative number. Note, however, that calculations with complex numbers take longer than the same calculations with real numbers.

4. Among integer types, choose the fastest type that provides the necessary range for your computations. Larger integer types will provide greater range, but may require multiple precision arithmetic on some CPUS. Very small types like 8 bit integers may be promoted to larger integers during calculations, which will also slow down the code.

The data types offered by C and Fortran generally correspond to what is supported by typical hardware, usually 8, 16, 32, and 64-bit integers, and 32 and 64-bit floating point numbers. Most compilers support 64-bit integers on 32-bit CPUs, in which case operations like addition and subtraction will take twice as long, since the CPU has to use two 32-bit machine instructions. This is called *multiple precision* arithmetic. Likewise, some compilers support 128-bit integers and floating point values, which will require multiple machine instructions on 64-bit hardware. If you require even bigger numbers than this, you will need to use a multiple-precision library functions, which will be significantly slower than hardware-supported arithmetic operations.

Integers are usually represented internally in unsigned binary or two's complement signed integer format, since these formats are directly supported by most hardware.

Floating point types, which approximate real numbers, are represented using IEEE standard floating point format in modern CPUs. There are some CPUs that do not have floating point support at the hardware level, but such CPUs are only generally used in small embedded applications and not for scientific computing.

Character types are generally processed internally as 8-bit unsigned integers representing the binary ISO character code. Some systems support 16 and 32-bit ISO character sets as well, depending on the locality selected in the operating system configuration. The 16 and 32-bit codes are only needed for non-alphabetic languages such as Chinese.

### 16.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why is data type selection important?
2. What do we need to know in order to select the optimal data type for a variable?
3. Why should we use integers rather than floating point whenever possible?
4. What if the data require the use of fractions, as with probabilities, which are always between 0.0 and 1.0? We have to use floating point, right?
5. When should we use 32-bit floating point and when should we use 64-bit floating point. Why?
6. What are the pros and cons of using complex data types?
7. What is the general approach for selecting an integer type?

## 16.2 Standard C Types

C supports the data types available in most CPUs, as well as some extended types. Table 16.1 outlines the standard data types available in C.

C's `int`, `long`, and `long long` types vary in size from one platform to another. Because of this, we have to make pessimistic assumptions about the range of these types in order to write portable code. We must assume that an `int` has the range of a 16-bit integer and the memory requirements of a 32-bit integer, because it will on some systems. NEVER ASSUME THAT YOUR CODE WILL ONLY BE USED ON THE CPU AND OPERATING SYSTEM WHERE YOU WROTE IT. Likewise, we have to assume a `long` has the range of a 32-bit integer and the memory requirements of a 64-bit integer.

On 16-bit processors, `int` is typically 16 bits, while `long` is 32 bits, requiring multiple precision arithmetic. On 32-bit machines, both `int` and `long` are typically 32 bits. On 64-bit machines, `int` is usually 32 bits while `long` is usually 64 bits.

| **C Type** | **Description** | **Range** | **Precision** |
|---|---|---|---|
| char | 8-bit signed integer | -128 to +127 | Exact |
| short | 16-bit signed integer | -32,768 to +32,767 | Exact |
| int | 16 or 32-bit signed integer (usually 16 bits on 8 or 16-bit processors, 32-bits on 32 or 64-bit processors) | -32,768 to +32,767 or -2,147,483,648 to +2,147,483,647 | Exact |
| long | 32 or 64-bit signed integer (usually 32 bits on 16-bit and 32-bit processors, 64 bits on 64-bit processors) | -2,147,483,648 to +2,147,483,647 or +/- 9.22337203685e+18 | Exact |
| long long | 64 or 128-bit signed integer | +/- 9.22337203685e+18 or +/- 1.7014118346e+38 | Exact |
| unsigned char | 8-bit unsigned integer | 0 to 255 | Exact |
| unsigned short | 16-bit unsigned integer | 0 to 65,535 | Exact |
| unsigned int | 16 or 32-bit unsigned integer | 0 to 65,535 or 4,294,967,295 | Exact |
| unsigned long | 32 or 64-bit unsigned integer | 0 to 4,294,967,295 or 1.84467440737e+19 | Exact |
| unsigned long long | 64 or 128-bit unsigned integer | 0 to 1.84467440737e+19 or 3.40282366921e+38 | Exact |
| float | Almost always 32-bit floating point | +/- (1.1754 x $10^{-38}$ to 3.4028 x $10^{38}$) | 24 bits (6-7 decimal digits) |
| double | Almost always 64-bit floating point | +/- (2.2250 x $10^{-308}$ to 1.7976 x $10^{308}$) | 52 bits (15-16 decimal digits) |
| long double | 64, 80, 96, or 128-bit floating point | +/- 3.3621 x $10^{-4932}$ to 1.1897 x $10^{+4932}$) | 114 bits (64 decimal digits) |
| float complex | Two floats for real and imaginary parts | Same as float | Same as float |
| double complex | Two doubles for real and imaginary parts | Same as double | Same as double |
| long double complex | Two 128-bit floating point values | Same as long double | Same as long double |

Table 16.1: C Data Types

To summarize, an int could be either 16 or 32 bits, depending on where your code is compiled. A long could be wither 32 or 64 bits, depending where your code is compiled. Use int or long only if either possible size is acceptable in terms of range and memory use.

The size of int and long vary across CPUs for the sake of speed. The int type never requires multiple precision arithmetic, except perhaps on very small 8-bit microcontrollers, where an int may be 16 bits. Hence, if you want to maximize speed, and 16 bits provides enough range, and 32 bits isn't too much memory to use, then use int. Likewise, if 32 bits provides enough range, and 64 bits isn't too much memory to use, then use long.

If you need signed integers larger than +32,767 or an unsigned integer larger than 65,535, then int or unsigned int will not be big enough on 16-bit systems. Use at least a long. Always consider using unsigned before going to a larger data type. If you need values up to 50,000, but do not need negative values, then unsigned int will suffice and you do not need long, which uses more memory and may require multiple machine instructions to process.

If you want to ensure a specific size for an integer variable regardless of whether it means using multiple precision arithmetic, there are additional types defined in inttypes.h, such as int64_t and uint64_t.

```
#include <inttypes.h>

int     main(int argc, char*argv[])

{
    int32_t     myint;

    ...
    return 0;
}
```

If you don't need numbers larger than +32,767 and you want to limit the size of the variable to 2 bytes, then use short. This is typically only done on embedded systems with very little memory or when using very large arrays on a typical computer. Note that short values will be promoted to int in many situations, which will slow down the program.

Logical/Boolean values in C do not have a separate data type. Instead, C treats them as they are handled internally, as integers. A value of 0 represents false, and any non-zero value represents true. The standard header file stdbool.h defines a data type called bool and constants true and false. You can use these instead of integer variables and values to make programs that use Boolean variables more self-documenting.

For real numbers, use double unless saving memory is an issue. There is no significant difference in speed between float and double on modern CPUs, and double has much higher precision. Only use float for very large arrays to reduce memory use.

### 16.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why do the size of int and long types vary from one computer to another?
2. What would be the best C data type to use for each of the following values? Explain your reasoning in each case. Use the C types table in the text and the adjoining explanations to make optimal choices.
   (a) A single variable containing a person's age, in years.
   (b) A single variable holding the temperature of a star, up to 40,000 Kelvin. The value is only approximate within 1,000 degrees.
   (c) A huge array of people's ages in years.
   (d) The balance of Joe Sixpack's checking account, in pennies.
   (e) A single variable holding Avogadro's constant.
   (f) A large array holding values like Avogadro's constant.

## 16.3 Standard Fortran Types

Fortran provides all the typical data types supported directly by most CPUs, as well as a few abstract types. Table 16.2 outlines the standard data types available in Fortran 90.

| **Fortran 90 Type** | **Description** | **Range** | **Precision** |
|---|---|---|---|
| integer(1) | 8-bit signed integer | -128 to +127 | Exact |
| integer(2) | 16-bit signed integer | -32,768 to +32,767 | Exact |
| integer(4) [ integer ] | 32-bit signed integer | -2,147,483,648 to +2,147,483,647 | Exact |
| integer(8) | 64-bit signed integer | +/- 9.22 x $10^{18}$ | Exact |
| integer(16) | 128-bit signed integer | +/- 9.22 x $10^{18}$ or +/- 1.70 x $10^{38}$ | Exact |
| real(4) [ real ] | 32-bit floating point | +/- (1.1754 x $10^{-38}$ to 3.4028 x $10^{38}$) | 24 bits (6-7 decimal digits) |
| real(8) [ double precision ] | 64-bit floating point | +/- (2.2250 x $10^{-308}$ to 1.7976 x $10^{308}$) | 52 bits (15-16 decimal digits) |
| real(16) | 128-bit floating point | +/- 3.3621 x $10^{-4932}$ to 1.1897 x $10^{+4932}$) | 114 bits (64 decimal digits) |
| character | 8-bit ISO, 16-bit in non-Latin locales | ISO 0 (NUL) to 255 (y-umlaut in ISO-Latin1) | Exact |
| logical | .true. or .false. | false to true | Exact |
| complex(4) [ complex ] | Two 32-bit floating point values | Same as real(4) | Same as real(4) |
| complex(8) [ double complex ] | Two 64-bit floating point values | Same as real(8) | Same as real(8) |
| complex(16) | Two 128-bit floating point values | Same as real(16) | Same as real(16) |

Table 16.2: Fortran 90 Data Types

### 16.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What would be the best Fortran data type to use for each of the following values? Explain your reasoning in each case. Use the C types table in the text and the adjoining explanations to make optimal choices.
   (a) A single variable containing a person's age, in years.
   (b) A single variable holding the temperature of a star, up to 40,000 Kelvin. The value is only approximate within 1,000 degrees.
   (c) The balance of Joe Sixpack's checking account, in pennies.
   (d) A huge array of people's ages in years.
   (e) A single variable holding Avogadro's constant.
   (f) A large array holding values like Avogadro's constant.

## 16.4 Variable Definitions and Declarations

Variable definitions were introduced in Section 15.8. We will now examine them in greater detail. A variable definition allocates memory to store a value, and assigns the memory location a name and a data type. The compiler uses the data type to determine which machine instructions to use to process the data. For example, adding two integers is done with a different machine instruction than adding two floating point values. The integer add instruction is faster, so defining a variable as `int` rather than `double` will lead to a faster program.

A C variable definition consists of a data type followed by one or more variable names separated by commas, and ultimately a semicolon:

```
type name [, name ...];
```

```
double height, width, area;
```

A Fortran 90 variable definition consists of a type, followed by two colons, followed by a list of variable names separated by commas:

```
type :: name [, name ...]
```

```
real(8) :: height, width, area
```

Variable names must begin with a letter or an underscore, and may contain letters, underscores, and digits after that.

The words "definition" and "declaration" are often used interchangeably in programming. In the context of languages such as C and C++, definition and declaration have different meanings. A definition in C allocates memory, whereas a declaration merely alludes to a variable or function defined elsewhere. For example, all C programs contain a global variable called `errno`, which contains the error code from the most recent standard library function. We can *declare* it in a given function by using the `extern` modifier to tell the compiler that this variable is defined somewhere else and we want to access it from here:

```
    extern int  errno;  // Declare (allude to), not define, errno
```

Without the `extern` modifier, we would be defining a new local variable called `errno` rather than alluding to the one already defined globally. The local variable would take precedence over the preexisting global variable under the variable scope rules of C, which means it would be impossible to access the global variable where the local variable is defined.

Note that while we can declare `errno` using an allusion as shown above, the modern method is to include `errno.h`, which contains this allusion:

```
#include <errno.h>
```

### 16.4.1 Initializers

Both C and Fortran support assigning an initial value to a variable in the definition.

```
double sum = 0.0;
```

```
real(8) :: sum = 0.0d0
```

Using this feature reduces the length of a program slightly by eliminating an assignment statement further down. Some would argue that it is less cohesive, however. It is generally a good practice to group related statements together in one cohesive block. In this context, it would mean initializing variables immediately before the code that depends on that initial value. This way, we can see that the variable is initialized properly without having to scroll up or search through the source code. This will save time and distractions while debugging.

```
double  sum;

// Suppose there are 100 lines of additional code here

// A cohesive block of code
sum = 0.0;
for (c = 0; c < list_size; ++c)
    sum += value;
```

```
double  sum = 0.0;

// Suppose there are 100 lines of additional code here

// This is not cohesive since the initialization of sum is far away from
// the loop that requires it
for (c = 0; c < list_size; ++c)
    sum += value;
```

### 16.4.2 Memory Maps

Memory locations for variables are generally allocated together in a block. Suppose we have the following variable definitions:

```
double  height, width, area;
int     list_size, age;
```

or

```
real(8) :: height, width, area
integer :: list_size, age
```

The program will have 8 bytes (64 bits) of memory allocated for each double or real(8) variable, and 4 bytes (32 bits) for each integer variable. A possible map of the memory space, assuming the block begins at memory address 4000, is shown in Table 16.3. Recall that each memory location contains 1 byte (8 bits), so each of these variables will occupy multiple memory addresses.

| Address | Variable |
|---|---|
| 4000 to 4007 | height |
| 4008 to 4015 | width |
| 4016 to 4023 | area |
| 4024 to 4027 | list_size |
| 4028 to 4031 | age |

Table 16.3: Memory Map

### 16.4.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What does a variable definition do?
2. What are the rules for naming variables?
3. What does a variable declaration do?
4. State one argument for and one against using initializers in a variable definition.
5. Draw a possible memory map for the following variable definitions:

   ```
   int     age, year;
   float   gpa;
   ```

## 16.5 Constants

### 16.5.1 Literal Constants

Like variables, C and Fortran constants also have types, which are determined by how the constant is written. For example, constants containing a decimal point are floating point values. Whether they are single or double precision is determined by an optional suffix.

Table 16.4 and Table 16.6 illustrate the types that compilers assign to various constants.

| Constant | Type |
|---|---|
| 45 | int |
| 45u | unsigned int |
| 45l | long |
| 45ul | unsigned long |
| 45ll | long long |
| 45ull | unsigned long long |
| 45.0 | double |
| 4.5e1 | double |
| 45f | float |
| 45.0l | long double |
| (45.0, 0.0) | double complex |
| 'A' | int (not char!) |
| "45" | const char * (array of char) |

Table 16.4: C Constants and Types

Integer constants that begin with '0' are interpreted as octal values, so `045` is equivalent to 4 * 8 + 5, or 37. Constants beginning with '0x' are interpreted as hexadecimal, so 0x45 = 4 * 16 + 5 = 69.

Note that a character between single quotes in C is not a string constant, but a character constant. It represents the ISO code of that character, and its type is `int`, not `char`. For example, in C, 'A' is exactly the same as 65, '0' is the same as 48, and '!' is the same as 33.

**Note** In C++, the type of a character constant is `char`. This is one of very few places where C++ is not backward-compatible with C.

There are also special sequences known as escape sequences to represent non-graphic characters. The most common ones are listed in Table 16.5.

| Sequence | ISO code (decimal) | Description |
|---|---|---|
| '\n' | 10 | Newline / Line feed |
| '\r' | 13 | Carriage return |
| '\0' | 0 | Null byte |
| '\t' | 9 | Tab |

Table 16.5: C Escape Sequences

### 16.5.2 Named Constants

Virtually every constant in your program should be assigned a name. Using names for constants makes the code much easier to read and allows the program to be more easily modified.

| **Constant** | **Type** |
|---|---|
| 45 | integer |
| 45_8 | integer(8) |
| 45.0 | real |
| 4.5e1 = 4.5 * 10ˆ1 = 45.0 | real |
| 4.5d1 = 4.5 * 10ˆ1 = 45.0 | real(8) (double precision) |
| 45.0d0 = 4.5 * 10ˆ0 = 4.5 | real(8) (double precision) |
| (45.0, 0.0) | complex |
| (45.0d0, 0.0d0) | complex(8) (double complex) |
| '45' | character(2) (string of length 2) |
| .true. | logical |

Table 16.6: Fortran 90 Constants and Types

In C, we can define constants using the `#define` preprocessor directive, or using a variable definition with the `const` modifier. When using the `const` modifier, we *must* provide an initializer. Any attempts by a program statement to alter the value will result in a compiler error.

```
#define PI 3.1415926535897832
```

```
const double PI = 3.1415926535897832;
```

When using #define, the preprocessor simply replaces the name (PI) with the value (3.1415926535897832) wherever it is used as an identifier in the program. In the printf statement below, only the second instance of PI is replaced by the preprocessor. The PI embedded in the string is not replaced.

```
printf("The value of PI is ", PI);
```

In Fortran 90, constants are defined like variables, but with the `parameter` modifier, which informs the compiler that the variable is read-only. Any attempts by a statement to alter its value will result in a compiler error.

```
real(8), parameter :: PI = 3.1415926535897832d0
```

A constant without a name is known as a *hard-coded constant*. Using named constants such as PI and MAX_RADIUS throughout the program instead of hard-coded constants such as 3.1415926535897832 and 10.0 has the following advantages:

- It allows the compiler to catch typos. If we mistype PI or MAX_RADIUS, the compiler will complain about an undefined variable or constant name. If we mistype 10.5 as 1.05, the program will compile just fine, but produce incorrect output. This is the hardest type of bug to find, and could have disastrous results if it goes undetected. Bugs like this have been known to cause catastrophic failures in cars, planes, spacecraft, and consumer products, leading to heavy financial losses, injury, and even death.
- If the names of the constants are descriptive (as they should be), the reader has an easier time understanding the code. This reduces the need for comments. It will likely take a bit of effort to figure out what a hard-coded constant actually means in a given context, whereas a good name makes it obvious.
- If you need to change the value of a constant (not an issue for PI, but quite possible for MAX_RADIUS), then having a named constant means you only need to make one change to the program. If you have the hard-coded constant 10.5 sprinkled all of the program, you'll have to carefully hunt them all down and change them. You can't simply change every instance of 10.5, since some of them might have a different purpose, and coincidentally have the same value as the maximum radius.

Example of bad C code:

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])
```

```
{
    // Bad idea: Using hard-coded constants
    printf("Area = %f\n", 3.1415926535897932 * 2.0 * 2.0);
    return EX_OK;
}
```

Example of better C code:

```
#include <stdio.h>
#include <math.h>       // Use M_PI provided with standard libraries
#include <sysexits.h>

#define RADIUS  10.0

int     main(int argc,char *argv[])

{
    printf("Area = %f\n", M_PI * RADIUS * RADIUS);
    return EX_OK;
}
```

Example of bad Fortran code:

```
program circle_area
    use iso_fortran_env

    ! Bad idea: Using hard-coded constants
    print *, 'Area = ', 3.1415926535897932d0 * 2.0 * 2.0
end program
```

Example of better Fortran code:

```
module constants
    ! Define only constants in modules, not variables! (i.e. use 'parameter')
    real(8), parameter :: &
        PI = 3.1415926535897932d0, &
        RADIUS = 10.0d0
end module constants

program circle_area
    use iso_fortran_env
    use constants           ! Constants defined above

    print *, 'Area = ', PI * RADIUS * RADIUS
end program
```

### 16.5.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the data type of each of the following C constants?
   - 16

- 16.0
- 'x'
- "x"

2. What is the data type of each of the following Fortran constants?

   - 16
   - 16.0
   - 'x'

3. Show how to write the constant 32 so that it has each of the following data types:

   (a) C: int, float, unsigned long, long long, double, float complex

   (b) Fortran: integer, real(4), real(8), complex, integer(2)

4. Show how to define a constant names AVOGADRO with the value 6.02 * 10ˆ23, using:

   (a) The C preprocessor.

   (b) A C const definition.

   (c) A Fortran constant definition.

5. Why should we almost always used named constants instead of hard-coded constants?

6. What are some potential long-term consequences of using hard-coded constants?

## 16.6 Math Operators and Expressions

C and Fortran support mostly the same math operators, except that Fortran supports exponentiation, which is done in C using the `pow()` library function.

Precedence and associativity in C and Fortran are the same as in algebra. Grouping () has the highest precedence, followed by negation (unary minus), then exponentiation, then multiplication and division, then addition and subtraction.

The C math operators are listed in Table 16.7. C also has a number of additional operators for bitwise operations, since it is often used for systems programming, cryptography, etc. They are shown in Table 16.8.

| **Operator** | **Operation** | **Precedence** | **Associativity / Order of Operation** |
|---|---|---|---|
| () | Grouping | 1 (highest) | Inside to outside |
| - | Negation | 2 | Right to left |
| ++ | Increment | 2 | Nearest to farthest |
| -- | Decrement | 2 | Nearest to farthest |
| * | Multiplication | 3 | Left to right |
| / | Division | 3 | Left to right |
| % | Mod (remainder) | 3 | Left to right |
| + | Addition | 4 | Left to right |
| - | Subtraction | 4 | Left to right |
| [operator]= (=, +=, -=, etc.) | Assignment | 6 | Right to left |

Table 16.7: Basic Math Operators in C

The C shift operators can often be used in place of integer multiplication and division to greatly improve program speed. When we shift the digits of a decimal number one place to the left, it has the effect of multiplying the value by 10. Likewise, shifting to the right divides by 10.

| Operator | Operation | Precedence | Associativity / Order of Operation |
|---|---|---|---|
| ~ | Complement (invert all bits) | 2 (same as unary -) | Left to right |
| << | Shift left | 5 (below + and -) | Left to right |
| >> | Shift right | 5 (below + and -) | Left to right |
| & | Bitwise AND | 6 | Left to right |
| ˆ | Bitwise XOR | 6 | Left to right |
| \| | Bitwise OR | 6 | Left to right |
| [operator]= (=, +=, -=, etc.) | Assignment | 7 | Right to left |

Table 16.8: Bitwise Operators in C

**Caution** Right-shifting an odd negative number will produce a result that is off by 1 from the mathematical definition of integer division. Programs that might do this must manually adjust.

The same principal applies to any other number base, including base 2, which is used to store integers internally. This works with both unsigned binary and 2's complement values. Hence, the following two statements are the same:

```
c = c * 8;
c = c << 3;    // Multiplies by 2, three times
```

The second statement may be significantly faster. A shift of any number of bits can occur within a single clock cycle in most modern CPUs, while multiplication may require many clock cycles. Some optimizers are smart enough to recognize which of your multiplications can be done with a shift and will generate a shift instruction automatically. However, it's best not to rely on such features. Using a shift instruction explicitly will ensure that your code is optimal no matter where it is compiled and executed.

The C increment and decrement operators are a convenient way to add 1 to or subtract 1 from a variable. The following two statements have the same effect:

```
++c;
c++;
```

Like all C operators, increment and decrement operators can also be used within expressions, though. This often allows us to eliminate a separate statement for incrementing a counter variable, which improves code density.

When used this way, the value of the expression will depend on whether the ++ or -- comes before (pre-increment, pre-decrement) or after (post-increment, post-decrement) the variable. If it comes before, then the increment occurs before the value of the variable is used in the expression, and vice versa.

```
a = 1;
b = 5 + ++a;    // Pre-increment: b is now 7 and a is 2
```

```
a = 1;
b = 5 + a++;    // Post-increment: b is now 6 and a is 2
```

C also allows us to combine any binary (two-operand) operator with '=' to save typing in common situations. The following two statements are equivalent:

```
c = c + 7;
c += 7;
```

This feature can also be used inside an expression, but must be parenthesized:

```
a = 1;
b = 5 + (a += 10);  // b is now 16 and a is 11
```

| Operator | Operation | Precedence | Order of Operation |
|---|---|---|---|
| () | grouping | 1 (highest) | inside to outside |
| ** | exponentiation | 3 | right to left |
| - | negation | 3 | right to left |
| * | multiplication | 4 | left to right |
| / | division | 4 | left to right |
| + | addition | 5 | left to right |
| - | subtraction | 5 | left to right |

Table 16.9: Basic Math Operators in Fortran

Fortran supports most common algebraic operators, including exponentiation:

**Example 16.1** Precedence and Order of Evaluation

```
        a + b + c equals (a + b) + c equals a + (b + c)
        a - b - c equals (a - b) - c does not equal a - (b - c)
        a ** b ** c equals a ** (b ** c) does not equal (a ** b) ** c
```

Unary minus is the same as multiplication by -1 or subtracting from 0, so -x equals -1 * x equals 0 - x.

Math operators are sensitive to the data types of the operands and may produce difference results for different types, i.e. they are *polymorphic*.

Recall from grade school that integer division and real division are not the same. Integer division results in an integer quotient and a remainder, while real division results in a different quotient and no remainder. ( The real quotient is the integer quotient + remainder / divisor. )

The C and Fortran divide operator (/) performs integer division if both operands are integers, and floating point division if either operand is floating point. Pay attention to the types of the operands when dividing, whether they are variables or constants.

```
int     a = 5, b = 10;
double  x, y, z = 10;
x = a / 2;      // a and 2 are both integers, x = 2.0, remainder discarded
y = a / 2.0;    // 2.0 is floating point, y = 2.5
x = a / b;      // a and b are both integers, x = 0.0, remainder discarded
x = a / z;      // z is floating point, x = 0.5
```

**Caution**
When using the division operator (/), care must be taken not to divide by zero.
Likewise, the exponentiation operator (**) cannot be used with negative bases and certain fractional exponents, unless working with complex numbers. For example, (-1) ** (0.5) does not exist in the real numbers, and hence will cause an error in Fortran.

### 16.6.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are the precedence and associativity rules for basic math operators in C and Fortran, compared with algebra?
2. How does bit shifting relate to multiplication and division?
3. Show a C statement that subtracts 1 from the variable x and adds the old value of x to y.

4. What is the value of y after each of the following statements?

```
    int     a = 1, b = 2, c = 3;
    double  x = 4.0, y;

    y = a / b;
    y = c / b;
    y = c / x;
```

## 16.7 Mixing Data Types

C and Fortran allow you to mix numeric data types in expressions. This should be avoided as much as possible, however, for two reasons:

- Most computer hardware cannot perform operations on two different data types. There are usually machine instructions for adding two integers of the same size, and machine instructions for adding two floating point values of the same size, but usually no machine instructions for adding an integer to a floating point value, adding two integers of different sizes (e.g. a 32-bit integer and a 64-bit integer), or adding floating point numbers of different sizes.

  Before mixed data type operations occur, one value must be converted to the exact same type as the other. E.g., when adding an integer value to a floating point value, the two's complement value of the integer is converted to IEEE floating point format and stored in a temporary location. Some CPUs have instructions to convert between native data types. Other CPUs will require several instructions that manually manipulate the bits to perform a conversion. In either case, conversions cause a significant slow-down, as the compiler must insert additional machine instructions between the calculations to convert the values. Adding an integer to a double may take several times as long as adding two integers or adding two doubles.

- Mixing data types also makes it more difficult to predict the output of a program, since it is easy to get confused and mistake integer operations for floating operations.

### 16.7.1 Implicit Data Conversions

*Promotions* occur when two different data types are used as operands to a mathematical operator. In all cases, the value of the lower ranking type is converted to the higher ranking type.

The rules of promotion are as follows:

1. Complex types have the highest rank. Any other numeric type mixed with a complex value will be promoted to complex.
2. Floating point values have the next highest rank. Integers mixed with floating point values will be promoted to the floating point type.
3. Integers have the lowest rank and are always promoted to other types they are mixed with.
4. Within each general category of types (complex, floating point, and integer), a larger type ranks higher than a smaller one. E.g., a `char` will be promoted when mixed with any other integer, a `short` will be promoted when mixed with an `int`, and an `int` will be promoted when mixed with a `long`. A `float` is promoted when mixed with a double.
5. It is almost always a bad idea to mix signed and unsigned integers.
6. Integer types smaller than `int` (`char` and `short`) are promoted even when not being mixed with higher types, just to prevent overflow. Promotion occurs when performing arithmetic operations on the values and when passing them to functions, such as `printf()`. This means that adding two `char` or `short` values takes longer than adding two `int` values.

   The program below demonstrates. The maximum value we can store in an `unsigned short` is 65,535. 32,768 + 32,768 is 65536. When we add `a + b`, the result is 65,536, which requires 17 bits to represent. The values from `a` and `b` are promoted to `int` before the addition occurs, so the expression `a + b` has type `int` and the expected value of 65,536. However, when this value is assigned to `c`, the leftmost bit is lost and `c` ends up getting 0. The expression `a + b / 2` also has type `int`, but dividing by 2 reduces its value to something that fits in a `short`, so `d` gets 32,768, not 0.

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    unsigned short  a, b, c, d;

    a = 32768;
    b = 32768;
    c = a + b;          // a + b is 65,536, beyond the range of c
    d = (a + b) / 2;    // a + b / 2 is 32,768, within the range of d

    // Output is 32768 32768 0 32768
    printf("%d %d %d %d\n", a, b, c, d);
    return EX_OK;
}
```

Demotions occur only when assigning an expression to a variable of a lower ranking type:

```
integer :: area
real(8) :: height, width

height = 4.5
width = 1.3

! Oops, lost the fractional part!  area gets 5, rather than 5.83
area = height * width
```

Note that when assigning any type of real value to an integer variable, the value is *truncated*, not rounded.

The type of a variable or constant is never changed. The promoted value is stored in a temporary memory location, and discarded when the program finishes evaluating the expression.

The examples below show all the steps necessary to evaluate a mixed expression, including calculations and implicit conversions. Note how implicit conversions can account for a significant fraction of the total operations.

Note also that some conversions, such as integer to floating point, can be rather expensive.

```
int     a, b, d;
double  x;

a = 4;
b = 6;
x = 3.0;
d = b / a + x * 5.0 - 2.5;
```

```
Expression                    Steps completed              Notes
d = b / a + x * 5.0 - 2.5
d = 1 + x * 5.0 - 2.5         Integer division b / a       1, not 1.5!
d = 1 + 15.0 - 2.5            Double multiplication
d = (double)1 + 15.0 - 2.5    Promote 1 to double          No work completed
d = 16.0 - 2.5                Double addition
d = 13.5                      Double subtraction
d = (int)13.5                 Demote 13.5d0 to integer     No work completed
d = 13                        Assign result to d
```

```
integer :: a, b, d
real(8) :: x

```

```
a = 4
b = 6
x = 3.0d0
d = b / a + x * 5.0 - 2.5
```

```
Expression                   Steps completed              Notes
d = b / a + x * 5.0 - 2.5
d = 1 + x * 5.0 - 2.5        Integer division             1, not 1.5!
d = 1 + x * dble(5.0) - 2.5  Promote 5.0 to real(8)       No work completed
d = 1 + 15.0d0 - 2.5         Double multiplication
d = dble(1) + 15.0d0 - 2.5   Promote 1 to real(8)         No work completed
d = 16.0d0 - 2.5             Double addition
d = 16.0d0 - dble(2.5)       Promote 2.5 to real(8)       No work completed
d = 13.5d0                   Double subtraction
d = int(13.5d0)              Demote 13.5d0 to integer     No work completed
d = 13                       Assign result to d
```

Even assuming that the CPU running this program can convert from one data type to another with a single instruction, this code would run significantly faster if all the variables and constants were the same data type to begin with. If the CPU cannot convert data types with a single instruction, this statement will be many times slower than necessary.

The compiler's optimizer may be able to eliminate some promotions at run-time by performing them at compile-time. Nevertheless, mixing data types will usually slow down your code to some extent.

### 16.7.2 Explicit Data Conversions

In C, we can convert from one data type to another by simply prefixing the expression with the desired data type in parenthesis. This is known as *casting* the expression.

```
int a = 1, b = 2;
double x;

// Convert value in a to double before division. This will cause an
// implicit promotion of the value of b when division occurs and
// produce a result of 0.5 instead of 0.
x = (double)a / b;

// We can explicitly convert both a and b, but the result is exactly the same
x = (double)a / (double)b
```

Explicit data conversions can be performed in Fortran 90 using intrinsic functions. For example, to force a real division of two integer variables, we could do the following:

```
integer :: a = 1, b = 2
real(8) :: x

// Convert value in a to double before division. This will cause an
// implicit promotion of the value of b when division occurs and
// produce a result of 0.5 instead of 0.
x = dble(a) / b
```

### 16.7.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are two reasons to avoid mixing data types and mathematical expressions?

| Function | Description | Returns |
|---|---|---|
| int(A) | Truncated integer | integer |
| int2(A) | Truncated 16-bit integer | integer(2) |
| int8(A) | Truncated 64-bit integer | integer(8) |
| nint(A) | Nearest integer | integer |
| real(A) | Nearest real | real |
| dble(A) | Nearest real(8) | real(8) [ double precision ] |
| cmplx(R [,I]) | Nearest complex | complex |
| dcmplx(R [,I]) | Nearest double complex | double complex [ complex(8) ] |

Table 16.10: Explicit Conversion Functions

2. When do promotions occur in C and in Fortran?
3. When do demotions occur?
4. How do we explicitly convert a value to a different data type in C? In Fortran?
5. What is the output of the following code segment? ( First try to determine it by reading the code, then type it in and run it to verify. )

```
int      a, b, c;
double   x, y;

a = 1;
b = 2;
c = 3;
x = 4.0;
y = 5.0;

printf("%d\n", a / b);
printf("%d\n", 3 / 4);
printf("%d\n", c / 2);
printf("%f\n", x / y);
printf("%f\n", c / x);
printf("%f\n", sqrt(x));
```

```
integer :: a, b, c
real(8) :: x, y

a = 1
b = 2
c = 3
x = 4.0d0
y = 5.0d0

print *, a / b
print *, 3 / 4
print *, c / 2
print *, x / y
print *, c / x
print *, x ** (1/2)
```

## 16.8 Code Quality

Code quality issues related to data types can be summarized as follows:

- Write literal constants so that they match the data type of other constants and variables used in the same expressions. This makes the code more self-documenting.
- Name all of your constants to make your code readable and modifiable.
- Avoid mixing data types in expressions as much as possible. Mixed may cause bugs that are difficult to track down.
- Don't abbreviate variable and constant names. This is an irrational type of laziness that will save you about 1 second of typing and cost much more in debugging time later.

## 16.9 Performance

- Use all integers if possible. Integer operations are about three times as fast as floating point.
- Choose the fastest integer type that has sufficient range and precision for your calculations.
- Avoid multiple-precision integer types. Operations on them take at least twice as long as the same operation on an integer type native to the underlying CPU.
- Avoid mixing data types in expressions. This causes promotions to be inserted between the useful mathematical operations. Defining everything as floating point types may lead to a faster program than mixing integers and floating point.

# Chapter 17

# Basic Terminal Input/Output (I/O)

## 17.1 C Terminal Input/Output (I/O)

### 17.1.1 C Language Support

Odd as it may sound, the C language actually has no built-in I/O statements. The designers of the C language were careful not to include any features that could be implemented as subprograms, and I/O fell into this category.

C I/O is performed by functions in the C *standard libraries*, collections of subprograms that are written in C. The standard libraries include a large number of functions that perform I/O, manipulate character strings, perform common math functions, etc.

### 17.1.2 C Standard Streams

The C interfaces to the low level kernel I/O routines are provided by functions like `open()`, `close()`, `read()` and `write()`. The read() and write() functions simply read and write blocks of bytes to or from an I/O device. This is the lowest level and most direct and efficient way to perform input and output in bulk. It is the best approach for programs like **cp**, which do not need to inspect every byte being transferred.

*Stream I/O* is another layer of software on top of the low level functions that allows us to conveniently and efficiently read or write one character at a time. When we write a character to a stream, it is simply placed into a memory buffer (array), which is much faster than actually writing it to an I/O device. When that buffer is full, the whole buffer is dumped to the I/O device using a low-level `write()`. Writing large blocks of data to a device like a disk is much more efficient than writing one character at a time, so this type of buffering greatly increases efficiency. Likewise, a low-level `read()` is used to read a block of data from a device into a memory buffer, from which stream I/O functions can quickly fetch one character at a time.

Recall from Table 3.9 that all Unix processes have three standard I/O streams from the moment they are born. The C standard libraries provide a set of convenience functions specifically for reading from the standard input stream and writing to the standard output stream. The standard error stream is accessed using the more general functions used for any other stream. Names for the standard streams are defined in the header file `stdio.h`, which we can incorporate into the program using:

```
#include <stdio.h>
```

| C Stream Name | Stream |
|---|---|
| stdin | Standard Input |
| stdout | Standard Output |
| stderr | Standard Error |

Table 17.1: Standard Stream Names

### 17.1.3 C Character I/O

The most basic C stream I/O functions are `getc()` and `putc()`, which read and write a single character.

```
    int     ch;

    ch = getc(stdin);
    putc(ch, stdout);
```

**Note** The C language treats characters the same as integers. Hence, the `getc()` function returns an integer and the `putc()` function takes an integer as the first argument. More on this in Chapter 16.

Other stream I/O functions that input or output strings and numbers are built on top of `getc()` and `putc()`. You can also write additional stream I/O functions of your own using `getc()` and `putc()`.

The `getchar()` and `putchar()` functions are provided for convenience. They read and write a character from the standard input and standard output.

```
    int     ch;

    ch = getchar();     // Same as getc(stdin);
    putchar(ch);        // Same as putc(ch, stdout);
```

### 17.1.4 C String I/O

The `puts()` and `gets()` functions read and write simple strings, which in C are arrays of characters, with a null byte (ISO character code 0, '\0') marking the end of the content.

Arrays are discussed in Chapter 23. For now, we will only use simple examples necessary for understanding basic I/O.

**Caution**
The `gets()` function is dangerous, since it may input a string longer than the array provided as an argument. This could lead to corruption of other variables in the program. Hence, other functions such as the more general `fgets()` should be used instead, even when reading from stdin.

```
    fgets(string, MAX_STRING_LEN, stdin);
```

Note that `fgets()` retains the newline ('\n') character at the end of the string. If this is not desired, it must be removed manually after calling `fgets()`, or some other function should be used for input. A more sophisticated interface is offered by the POSIX standard `getline()` function, which requires knowledge of dynamic memory allocation (covered later).

```
#include <stdio.h>
#include <sysexits.h>

#define MAX_NAME_LEN    100

int     main()

{
    // Add 1 for null byte so that MAX_NAME_LEN means what it says
    char    name[MAX_NAME_LEN + 1];

    fputs("Please input your name: ", stdout);
    fgets(name, MAX_NAME_LEN + 1, stdin);
    fputs(name, stdout);

    return EX_OK;
}
```

The `puts()` function appends a newline character to the string, so the next output will be on a new line. If this is not desired, one can use `fputs()` instead. This would be the case for strings read by `fgets()`, which already contain the newline that was read from input.

```
    fputs(string, stdout);
```

### 17.1.5 C Formatted I/O

The `printf()` and `scanf()` functions can be used for more complex input and output. Both `printf()` and `scanf()` require the first argument to be a *format string*, optionally followed by additional arguments that must match the format string. The format string is similar to that used by the **printf** command, but the C function has many more options for handling various data types.

For each argument after the format string, the format string must contain a *format specifier* that matches the type of the argument.

```
    int     fahrenheit = 76;

    printf("The temperature is %d (%d Celsius)\n",
           fahrenheit, (fahrenheit - 32) * 5 / 9);
```

Some of the most common format specifiers for printf are outlined in Table 17.2. For more information, see the `printf(3)` man page, i.e. run **man 3 printf** to get the section 3 man page about the function instead of the section 1 man page about the Unix command.

| Type | Number Format | Format specifier |
|---|---|---|
| char, short, int | printable character | %c |
| string (character array) | printable characters | %s |
| char, short, int | decimal | %d |
| char, short, int | octal | %o |
| char, short, int | hexadecimal | %x |
| unsigned char, unsigned short, unsigned int | decimal | %u |
| size_t (used for array subscripts) | decimal | %zu |
| float, double | decimal | %f |
| float, double | scientific notation | %e |
| float, double | double or scientific notation | %g |
| long double | decimal | %Lf |

Table 17.2: Format specifiers for printf()

Prefixing any numeric format specifier with a lower case L ('l') corresponds to prefixing the type with 'long'. For example, "%ld" is for `long int`, "%lu" for "unsigned long int", "%lld" for `long long int`, "%lf" for `long double`.

**Note**

It may appear that the printf format specifiers don't always match the argument type, but this is because `char`, `short`, and `float` arguments are promoted to `int` or `double` when passed to a function. Because of this, "%f" is used for both `float` and `double`, and "%d" and "%c" can be used for `char`, `short`, and `int`.

The `scanf()` function reads formatted text input and converts the values to the proper binary format based on the format specifier matching each argument.

```
    int     temperature;
    double  pressure;

    scanf("%d %lf", &temperature, &pressure);
```

The "%d" format specifier tells scanf to read the sequence of characters as a decimal integer and convert it to the binary format of an `int`. The "%lf" tells scanf to read the next sequence of characters as a decimal real number and convert it to the binary format of a `double` (not a `long double` as it would mean to `printf`. The binary values are then stored in the variables (memory locations) called temperature and pressure.

Note that each argument after the format string is prefixed with an ampersand (&). This is because in C, all arguments in function calls are passed by value, meaning that the function gets a copy of the value, rather than accessing the caller's copy. This is explained in detail in Chapter 20. For example, consider the following printf() call:

```
printf("The area is %f.\n", area);
```

The `printf()` function only gets a copy of the value of `area`. It does not have access to the variable `area`, because it does not know what memory address `area` represents. Hence, it is impossible for `printf()` to modify the value of `area`.

However, an input function like `scanf()` needs to modify the variables passed to it. We allow it to do so by passing it the *memory address* of each variable, rather than just a copy of the value.

Recall that a variable is simply a name for a memory location where some value is stored. An ampersand (&) preceding a variable name represents the *address of the variable*, whereas the variable name alone would represent the value contained at that address.

By passing the address of a variable to `scanf()`, we give it the ability to put something in that memory location.

Some of the most common format specifiers for `scanf()` are outlined in Table 17.3.

| **Type** | **Number Format** | **Format specifier** |
|---|---|---|
| char | printable character | %c |
| char | decimal | %hhd |
| short | decimal | %hd |
| int | decimal | %d |
| int | octal | %o |
| int | hexadecimal | %x |
| int | hexadecimal if input begins with "0x", octal if it begins with "0", otherwise decimal | %i |
| long int | decimal | %ld |
| long long int | decimal | %lld |
| float | decimal | %f |
| double | decimal | %lf |
| long double | decimal | %Lf |

Table 17.3: Format specifiers for scanf()

**Note** Unlike `printf()`, `scanf()` format specifiers do not match multiple types. This is because the arguments to `scanf()` are addresses, not values, and are never promoted. Hence, the format specifiers must match the type of the variable exactly.

**Caution** Like `gets()`, `scanf()` may input a string too large for the character array provided, so it is considered a dangerous input function for strings. It should only be used to input numeric data or single characters.

**Caution**

Don't use printf() or scanf() where a simpler function such as putchar(), getchar(), puts(), or fgets() will do. Using a function that scans the format string for format specifiers that are not there is a waste of CPU time.

```
    printf("This is silly, just use puts() or fputs().\n");
```

The printf() and scanf() functions do not directly support complex data types. Instead, it is left to the programmer to decide how complex numbers are represented in input and output as two separate float or double values. Functions such as creal() and cimag() can be used to extract the real and imaginary parts of a complex number for output in printf() as shown in Section 17.1.6.

### 17.1.6 Example of C input and output

```
#include <stdio.h>
#include <sysexits.h>
#include <complex.h>

int     main(int argc,char *argv[])

{
    double a, b, c;
    complex double  root1, root2, two_a, discriminant_sqrt;

    printf("Please enter the coefficients a, b, and c: ");
    scanf("%lf %lf %lf", &a, &b, &c);

    /*
     * Precompute terms that will be used more than once.
     * Cast RESULT of all-double expression to complex so that only one
     * promotion occurs.
     * Convert 2a to complex now to avoid multiple promotions when
     * computing the roots.
     */
    discriminant_sqrt = csqrt((complex double)(b * b - 4.0 * a * c));
    two_a = 2.0 * a;

    root1 = (-b + discriminant_sqrt) / two_a;
    root2 = (-b - discriminant_sqrt) / two_a;

    printf("The roots of %fx^2 + %fx + %f are:\n", a, b, c);
    printf("%g + %gi\n", creal(root1), cimag(root1));
    printf("%g + %gi\n", creal(root2), cimag(root2));
    return EX_OK;
}
```

Output from the program above:

```
Please enter the coefficients a, b, and c: 1 2 3
The roots of 1.000000x^2 + 2.000000x + 3.000000 are:
-1 + 1.41421i
-1 + -1.41421i
```

### 17.1.7 Using stderr with C

The puts(), and printf() functions are special cases that implicitly use stdout.

To print to stderr, we simply use the more general functions `putc()` and `fputs()`. These are the same functions we would use to print to any other file stream that our program opened, as discussed in Chapter 25.

```
    fputs("Hello, world!\n", stdout);    // Same as puts("Hello, world!");
    fputs("Sorry, radius must be non-negative.\n", stderr);
    fprintf(stderr, "Sorry, radius %f is invalid.  It must be non-negative.\n", radius);
```

### 17.1.8 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is low-level I/O in C?
2. What is stream I/O in C? Explain.
3. What are the names of the standard input, standard output, and standard error streams in C?
4. Show a variable definition for a variable called `ch`, a C statement that reads a single character into it from the standard input, and another statement that prints the character to the standard output.
5. Write a C program that asks the user their name, reads a line of text no longer than MAX_NAME_LEN from the standard input, and prints "Hello, " followed by the name to the standard output.
6. Write a C program that asks the user for the radius of a circle and the prints the area of the circle.

   ```
   What is the radius? 10
   The area is 314.159265.
   ```

## 17.2 Fortran Terminal Input/Output (I/O)

### 17.2.1 Write and Print

The generic Fortran output statement is `write`, which is an intrinsic subroutine. A write statement consists of the keyword `write` followed by (unit, format) and finally a list of variables and constants to be written, separated by commas.

```
    write (unit, format) value [, value ...]
```

The *unit number* is an integer value that tells the write statement where to send the output. Each unit number represents a *file stream* such as the standard input, standard output, or standard error. Opening files and creating additional file streams is covered in a later chapter. If a * is given for the unit number, the write statement writes to the *standard output* stream, which is normally connected to the terminal screen.

The *format string* is a template for how the output should look. It specifies how many characters and how many fractional digits should be printed for real numbers, for example. If a * is used in place of the format string, the write statement will use default formatting for all items printed, which is somewhat ugly, but functional.

After the (unit, format) come the items to be written. These may be variables of any type, string constants (characters between quotes), numeric constants, or algebraic expressions.

```
    real(8) :: height, width

    write (*,*) 'Please enter height and width on the same line:'
    read (*,*) height, width
    write (*,*) 'The area is ', height * width
```

The `print` statement is short-hand for writing to the standard output.

```
    print format, value [, value ...]
```

is equivalent to

```
    write (*, format) value [, value ...]
```

Example:

```
    real(8) :: height, width

    print *, 'Please enter height and width on the same line:'
    read (*,*) height, width
    print *, 'The area is ', height * width
```

### 17.2.2 Read

The read statement inputs values from a stream such as the standard input and places the values into variables. The components of a read statement are the same as for a write statement.

Providing a * as the unit number indicates that the read statement should read from the *standard input* stream, which is normally attached to the keyboard.

```
    real(8) :: height, width, area

    print *, 'Please enter the height and width on one line:'
    read (*, *) height, width
    area = height * width
    print *, 'The area is ', area
```

The read statement can also be written in short-hand form like the print statement:

```
    read *, height, width
```

### 17.2.3 The One Statement = One Line Rule

In Fortran, every read, write or print statement reads or writes one line of input or output. Unlike C, we cannot print part of a line with one statement and add to it with another statement. Fortran `print` and `write` statements always appends a newline character and read always reads to the end of an input line. Hence, the read statement above expects to find both height and width on the same line of input. That is, the user cannot press enter between the numbers.

Likewise, all of the output from a write or print statement will appear on the same line of output. The write or print statement will output all values, and send a newline character after the end of all output, so the next write or print will begin on a new line.

What is the output of the following?

```
    area = 4.0d0

    print *, 'The area is '
    print *, area
    print *, '.'
```

```
The area is
  10.000000000000000
.
```

To get everything on one line, we must do it in one print/write statement:

```
    print *, 'The area is ', area, '.'
```

With default formatting, it's not too pretty, but it is on one line as we desired.

```
The area is    10.000000000000000      .
```

### 17.2.4 Standard Units

As mentioned earlier, using a * in place of the unit number tells the read statement to use the standard input stream and the write or print statement to use the standard output.

If you include the ISO_FORTRAN_ENV module in your program with:

```
    use ISO_FORTRAN_ENV
```

you can then use the actual unit numbers for the standard streams. The ISO_FORTRAN_ENV module provides the following named constants:

| Unit | Stream |
|---|---|
| INPUT_UNIT | Standard Input |
| OUTPUT_UNIT | Standard Output |
| ERROR_UNIT | Standard Error |

Table 17.4: Standard Stream Names

For the standard input and standard output, it's easier to type * than INPUT_UNIT or OUTPUT_UNIT. However, to write to the standard error unit, we must use ERROR_UNIT explicitly.

### 17.2.5 Example of Fortran input and output

```
!-----------------------------------------------------------------------
!   Description:
!       Find roots of a quadratic equation
!
!   Usage:
!       quadratic
!
!   Returns:
!       Nothing
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-09  Jason Bacon Begin
!-----------------------------------------------------------------------

program quadratic
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Local variables
    real(8) :: a, b, c
    ! Use complex discriminant so we can get the square root
    ! even if it's negative.
    double complex :: discriminant_sqrt, root1, root2, two_a

    ! Input equation
    print *, 'Please enter the coefficients a, b, and c on one line:'
    read *, a, b, c
    ! Precompute terms that will be used more than once.
    ! Cast RESULT of all-double expression to complex so that only one
    ! promotion occurs.
    ! Convert 2a to complex now to avoid multiple promotions when
    ! computing the roots.
    discriminant_sqrt = sqrt(dcmplx(b * b - 4.0d0 * a * c))
    two_a = 2.0d0 * a

    ! Compute roots
    root1 = (-b + discriminant_sqrt) / two_a
    root2 = (-b - discriminant_sqrt) / two_a

    ! Output roots
    print *, 'The roots of ', a, 'x^2 + ', b, 'x + ', c, ' are:'
    print *, root1
    print *, root2
end program
```

Output from the program above:

```
 Please enter the coefficients a, b, and c on one line:
1 2 3
 The roots of    1.0000000000000000       x^2 +    2.0000000000000000
 x +    3.0000000000000000       are:
 ( -1.0000000000000000     ,  1.4142135623730951     )
 ( -1.0000000000000000     , -1.4142135623730951     )
```

### 17.2.6 Fortran Formatted I/O

As mentioned earlier, one of the asterisks in the write, print, and read statements tells the statement to use default formatting.

The default format is designed to use a consistent number of columns and present accurate information. This often leads to excessive white space and decimal places in the output:

```
    integer :: i = 5
    real(8) :: x = 5.0d0

    ! Statements
    print *, i, x
```

```
Output:             5   5.0000000000000000
Columns: 123456789012345678901234567890123
```

As you can see from the "ruler" below the output, the integer is printed across 12 columns, and the real(8) value across 21.

If we want to control the appearance of output or validate the format of input, we can replace the asterisk with a *format specifier*. The format specifier contains *descriptors* for each item to be printed, in the same order as the arguments to the read, write or print statement. The first descriptor describes the format of the first argument, and so on.

The most common descriptors are described in Table 17.5.

```
    integer :: i = 5, j = -6
    real(8) :: x = 5.0d0, y = -6.0d0

    ! Statements
    print '(i3, f10.2)', i, x

Output:   5      5.00
Ruler:  1234567890123
```

| Descriptor | Type | Format |
|---|---|---|
| Iw | Any integer type | w columns total |
| Fw.d | Any real type | w columns total, d decimal places |
| Ew.d | Any real type | w columns total, d decimal places, scientific notation |
| Aw | String | w columns total |
| Gw.d | Any real type | Program chooses the best format |

Table 17.5: Common Format Descriptors

```
    write (*, '(i3, f10.2)'), i, x

Output:   5      5.00
Ruler:  1234567890123

    print '(i3, e10.2)', i, x

Output:   5  0.50E+01
Ruler:  1234567890123

    print '(i3, g10.2)', i, x

Output:   5   5.0
Ruler:  1234567890123

    print '(i3, i3, f10.2, f10.2)', i, j, x, y

Output:   5 -6      5.00     -6.00
Ruler:  12345678901234567890123456
```

If the width is non-zero and the value to be printed doesn't fit, the program will print a '*' instead of the number:

```
    integer :: i = 5, j = -6
    real(8) :: x = 5.0d0, y = -6.0d0

    ! Statements
    print '(i1, i1, f4.2, f4.2)', i, j, x, y

Output: 5*5.00****
Ruler:  1234567890
```

If the width is 0, the program will print the minimum number of digits needed to convey the correct value:

```
    integer :: i = 5, j = -6
    real(8) :: x = 5.0d0, y = -6.0d0

    ! Statements
    print '(i0, i0, f0.2, f0.2)', i, j, x, y

Output: 5-65.00-6.00
Ruler:  123456789012
```

The width and decimal places specifiers can be omitted for string values, in which case the program will print the string to its full length. This feature can be used with the zero-width descriptor for embedding numbers in text:

```
    integer :: i = 5, j = -6
    real(8) :: x = 5.0d0, y = -6.0d0

    ! Statements
    print '(a, i0, a, i0, a, f0.2, a, f0.2)', &
        'i = ', i, '  j = ', j, '  x = ', x, '  y = ', y

Output: i = 5  j = -6  x = 5.00  y = -6.00
Ruler:  1234567890123456789012345678901234
```

A format descriptor, or group of format descriptors in () can be preceded by an integer repeat count to shorten the format specifier:

```
    print '(i3, i3, f10.2, f10.2)', i, j, x, y

! Same as

    print '(2i3, 2f10.2)', i, j, x, y

    print '(a, i0, a, i0, a, f0.2, a, f0.2)', &
        'i = ', i, '  j = ', j, '  x = ', x, '  y = ', y

! Same as

    print '(2(a, i0), 2(a, f0.2))', &
        'i = ', i, '  j = ', j, '  x = ', x, '  y = ', y
```

Format specifiers can also be used in read statements. If a format specifier is used, then the input must match the format exactly, or an error will be generated. This is done to perform strict checking on input read from a file, if slight deviations in the format might indicate a problem with the data.

If we want to use the same format specifier in multiple print, read, or write statements, we can separate it out to a labeled format statement, and use the label in its place:

```
    print 10, 'i = ', i, '  j = ', j, '  x = ', x, '  y = ', y
    print 10, 'k = ', k, '  l = ', l, '  v = ', v, '  w = ', w

10  format (2(a, i0), 2(a, f0.2))
```

### 17.2.7 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the "One statement = one line" rule?
2. What are the names of the standard input, standard output, and standard error streams in Fortran when using ISO_FORTRAN_ENV
3. Write a Fortran program that asks the user their name, reads a line of text no longer than MAX_NAME_LEN from the standard input, and prints "Hello, " followed by the name to the standard output.
4. Write a Fortran program that asks the user for the radius of a circle and the prints the area of the circle.

```
 What is the radius?
10
 The area is    314.15926535897933      .
```

## 17.3 Using Standard Error

The standard error stream is, by default, attached to the terminal screen along with the standard output. However, programs should not regard them as equivalent. Normal results from a program should be sent to the standard output stream. Errors, warnings, and other information that is meant to inform the user about the *condition* of the program rather than present *results* from the computations, should be sent to the standard error.

Having two separate output streams for results and messages allows the user of the program to separate error and warning messages from normal output using redirection:

```
shell> a.out < input.txt > output.txt 2> errors.txt
```

This will be important to users who want to store screen output in a file for further processing. In this case, they will not want the output contaminated with error and warning messages.

Users may also want to see errors and warnings on the screen while redirecting results to a file. The following command reads keyboard input from the file `input.txt` and send results meant for the screen to `output.txt`. Anything the program prints to the standard error stream (ERROR_UNIT), however, will be written to the screen.

```
shell> a.out < input.txt > output.txt
```

### 17.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Why is it important to separate error messages from normal output in all of our programs?

# Chapter 18

# Conditional Execution

## 18.1 Motivation

Computers are much more useful when they can make decisions. Most calculations cannot proceed very far without reaching some sort of crossroad, where a decision must be made about what to do next. Very few programs perform a sequence of calculations with no branches. In this chapter, we will explore ways to design and implement decision-making in our algorithms and code.

## 18.2 Design vs. Implementation

Most decisions involved in a process should be discovered during the design phase, before implementation (coding) begins. The implementation phase then remains primarily a process of translation.

Conditionals (decisions) can be represented as flow charts, as shown in Figure 18.1. This is often helpful for grasping the high-level structure of the algorithm.

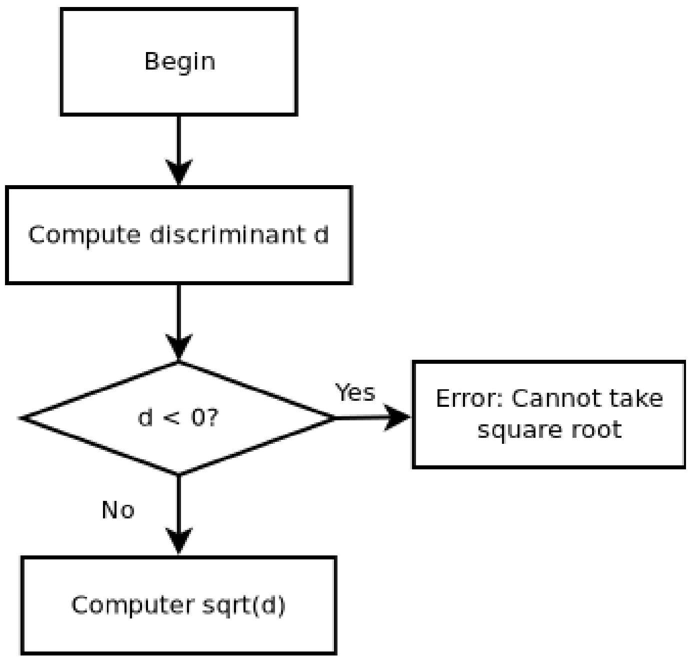


Figure 18.1: Flowchart for a simple conditional

## 18.3 Boolean Expressions

Decisions are based on *Boolean expressions*, algebra-like expressions with a value of true or false. The term Boolean is named after George Boole, a well-known mathematician, philosopher, and logician, who developed a systematic approach to logic now known as *Boolean algebra*. The basis of Boolean algebra was published in his text "The Laws of Thought", and provides a significant part of the foundation of computer science.

The C language does not have a separate Boolean data type. Recall that the design philosophy of C is minimalist and explicit rather than bloated and abstract. Boolean values in C are represented as they are stored internally, as integer values. A value of 0 is interpreted as `false` and any non-zero value is interpreted as `true`. A derived Boolean type called `bool` is provided in the standard header file `stdbool.h`, along with `true` and `false` constants. We can use this to make our C code more self-documenting, rather than defining Boolean variables as `int`.

### 18.3.1 Boolean Constants

Since the C language does not have a separate Boolean data type, there are no built-in constants. Traditionally, C programmers would define a Boolean variable as an integer.

```
#include <sysexits.h>

int     main()
{
    int     raining;

    raining = 0;    // 0 means false, non-zero means true

    ...
    return EX_OK;
}
```

However, the `bool` data type and the values `true` and `false` are now defined in the standard header file `stdbool.h`. While not part of the C language, it is included with all modern C compilers.

```
#include <stdbool.h>
#include <sysexits.h>

int     main()
{
    bool    raining;

    raining = false;

    ...
    return EX_OK;
}
```

Fortran provides logical constants `.true.` and `.false.`. The periods are actually part of the constants, and are a vestige of early Fortran, which used periods to delineate many program elements (tokens) in order to make parsing easier.

```
program example
    implicit none

    logical raining

    raining = .false.
end program
```

### 18.3.2 Relational Expressions

A *relational expression* is a Boolean expression (an expression with a value of true or false) that compares (tests the relation between) two values of any type, such as integers, real numbers, or strings.

| Expression | Value |
|---|---|
| 1 > 2 | false |
| 5 = 10/2 | true |
| "Bart" = "Bert" | false |

Table 18.1: Relational Expressions

Relational expressions in C are actually integers, with a value of 0 for false and 1 for true, which can be represented as `false` and `true` if `stdbool.h` is included in the program. C constructs such as `if` statements and loops will interpret a value of zero as false and any non-zero value as true.

Relational expressions in Fortran have the data type `logical` and a value of `.false.` or `.true.`.

#### Relational Operators

Relational expressions are formed using *Relational operators* and two values. C and Fortran provide the usual arithmetic relations such as "equals", "less than",etc.

**Caution**
Floating point values should NEVER be compared for equality. Due to round-off error, it is possible that the actual value might be slightly different than the expected value and therefore not equal to the value we're looking for. For example, the following condition is likely to fail, because 0.1 cannot be accurately represented in binary floating point. Even if it could be represented accurately, the value of thickness might be off due to round off during computations.

```
if ( thickness == 0.1 )
```

Fortran 90 and later use more intuitive symbols for operators, while Fortran 77 and earlier use old style operators. Fortran 90 is backward-compatible with Fortran 77, so the old style operators can still be used. This allows old Fortran 77 code to be compiled with the newer compilers, but newly written code should use the new operators.

| Relational Operator | C | Fortran 90 and Later | Fortran 77 and Earlier |
|---|---|---|---|
| Not equal | != | /= | .ne. |
| Equals | == | == | .eq. |
| Less than | < | < | .lt. |
| Less or equal | <= | <= | .le. |
| Greater than | > | > | .gt. |
| Greater or equal | >= | >= | .ge. |

Table 18.2: Relational Operators

Fortran 90 adopted the operators from C to replace the arcane .ne., etc., except for !=, because '!' marks the beginning of a comment in Fortran.

#### Boolean Operators

Some Boolean expressions entail more than one relation. For example, an airline may want to offer free peanuts to both kids and seniors, in which case we would want to know if age < 18 or age >= 65.

| Operation | C Operator | Fortran Operator |
|---|---|---|
| and | && | .and. |
| or | \|\| | .or. |
| not | ! | .not. |

Table 18.3: Boolean Operators

### 18.3.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the data type of a Boolean expression in C? In Fortran?
2. What values represents true and false in C? In Fortran?
3. Are there named Boolean constants in C?
4. What is a relational expression?

## 18.4 If Statements

### 18.4.1 C If Statements

In C, the `if-else` construct is used to test conditions such as relations and optionally execute one or more statements depending on whether the condition is true. The `else` clause is optional, so if we merely want to do something if a condition is true and do nothing if the condition is false, then the code would look something like the following:

```
    if ( condition )
        statement;      // Do this only if condition is true
```

If we want to do one thing if the condition is true and a different thing if it is false, then we add the else clause:

```
    if ( condition )
        statement1;     // Do this only if condition is true
    else
        statement2;     // Do this only if condition is false
```

In C, including multiple statements under an `if` or any other construct uses the same syntax. A C *block statement* is any group of statements enclosed in curly braces (`{}`). Block statements can be used anywhere in a C program, such as in `if` statements, loops, or even alone.

Since C is a free-format language, it makes no difference to the compiler whether the curly braces are on the same line or different lines. However, indentation is important for readability. Adding four columns to each indent level is common, since this is half a standard TAB indent. Typically, the braces are aligned in the same column and the statements between the braces are indented one more level. Some programmers prefer to place the first brace on the same line as the `if` in order to fit more code onto the screen.

```
    if ( (age < 18) || (age >= 65) )
    {
        puts("You get free peanuts!");
        --peanut_inventory;
    }

    if ( (age < 18) || (age >= 65) ) {
        puts("You get free peanuts!");
        --peanut_inventory;
    }
    if ( (age < 18) || (age >= 65) )
    {
        puts("You get free peanuts!");
        --peanut_inventory;
    }
    else
        puts("No free peanuts for you!");
```

Free programs such as **indent** can automatically reformat C source files, changing the indent levels and several other aspects of code format.

### 18.4.2 Fortran If Statements

The Fortran `if-then-else-endif` construct is used to test Boolean expressions and conditionally execute one or more statements.

The minimal version of Fortran conditional execution consists of a condition and a single statement. The statement must be on the same line as the if ( condition ), unless a continuation character & is used. If the statement is below the if, it should be indented one more level than the `if` for readability.

```
if ( condition ) statement

if ( condition ) &
    statement
```

```
if ( (age < 18) .or. (age >= 65) ) &
    print *, 'You get free peanuts.'
```

If more than one statement are to be conditionally executed, then Fortran requires the longer form, with the word `then` on the same line as the if, the statements on subsequent lines, and finally `endif` on a line below the last statement. The statements between the `if` and `endif` are indented one more level than the if and endif.

```
    if ( (age < 18) .or. (age >= 65) ) then
        print *, 'You get free peanuts!'
        peanut_inventory = peanut_inventory - 1
    endif
```

```
    if ( (age < 18) .or. (age >= 65) ) then
        print *, 'You get free peanuts!'
        peanut_inventory = peanut_inventory - 1
    else
        print *, 'No free peanuts for you!'
    endif
```

### 18.4.3 Additional Use Cases

In some cases, we might be tempted to use an if statement to set a Boolean variable for later use:

```
    bool    free_peanuts;

    if ( (age < 18) || (age >= 65) )
        free_peanuts = true;
    else
        free_peanuts = false;
    if ( (age < 18) .or. (age >= 65) ) then
        free_peanuts = .true.
    else
        free_peanuts = .false.
    endif
```

However, note that we are simply setting `free_peanuts` to the same value as the condition in the if statement. In cases like this, we can replace the entire if-else statement with a simple assignment:

```
bool    free_peanuts;

free_peanuts = (age < 18) || (age >= 65);
```

```
logical free_peanuts;

free_peanuts = (age < 18) .or. (age >= 65)
```

We can take it a step further, and check for a series of conditions:

```
    if ( age < 3 )
        puts("No peanuts for you! You might choke on them.");
    else if ( age < 10 )
        puts("Ask your mom if you can have some peanuts.");
    else if ( (age < 18) || (age >= 65) )
    {
        puts("You get free peanuts!");
        --peanut_inventory;
    }
    else
        puts("No free peanuts for you!");
```

```
    if ( age < 3 ) then
        print *, 'No peanuts for you! You might choke on them.'
    else if ( age < 10 ) then
        print *, 'Ask your mom if you can have some peanuts.'
    else if ( (age < 18) .or. (age >= 65) ) then
        print *, 'You get free peanuts!'
        peanut_inventory = peanut_inventory - 1
    else
        print *, 'No free peanuts for you!'
    endif
```

### 18.4.4 Nesting If Statements

The code inside an if or else clause can contain anything, including other if statements. Suppose the economy is tight, and we decide to cancel free peanuts for seniors, but keep them for kids. We might then restructure the code as follows:

```
    if ( age < 18 )
        if ( age < 3 )
            puts("No peanuts for you! You might choke on them.");
        else if ( age < 10 )
            puts("Ask your mom if you can have some peanuts.");
        else
        {
            puts("You get free peanuts!");
            --peanut_inventory;
        }
    else
        puts("No free peanuts for you!");
    if ( age < 18 ) then
        if ( age < 3 ) then
            print *, 'No peanuts for you! You might choke on them.'
        else if ( age < 10 ) then
            print *, 'Ask your mom if you can have some peanuts.'
        else
            print *, 'You get free peanuts!'
            peanut_inventory = peanut_inventory - 1
        endif
    else
        print *, 'No free peanuts for you!'
    endif
```

### 18.4.5 Practice Break

With the class, develop a program that asks the user for the radius of a sphere and then computes the surface area and volume. The program should print an error message to stderr and exit with status EX_DATAERR if a valid number is not entered, i.e. `scanf()` does not return 1, or the number entered is less than or equal to zero. Be sure to use meaningful variable names, input prompts, and error messages.

### 18.4.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a block statement in C? Where can it be used?
2. How should if statements be indented?
3. What is wrong with the following code?

```
    bool    free_peanuts;

    if ( (age < 18) || (age >= 65) )
        free_peanuts = true;
    else
        free_peanuts = false;
```

4. Write a C or Fortran program that inputs the coefficients of a quadratic equation and computes the roots, if real roots exist. The program should return EX_OK if real roots exist and EX_DATAERR if they do not. These constants are defined in `sysexits.h`.

```
Please enter the coefficients a, b, and c: 1 2 3
There are no real roots for 1.000000x^2 + 2.000000x + 3.000000 = 0.

Please enter the coefficients a, b, and c: 1 4 2
The roots of 1.000000x^2 + 4.000000x + 2.000000 = 0 are -0.585786, -3.41421.
```

## 18.5 Switch, Select-Case

If we need to compare a single expression to a number of different constant values, we can do a series of `else-ifs` as we did above to check multiple age ranges. A cleaner, more readable, and probably faster option is the C `switch` statement or the Fortran `select case` statement. The expression being compared, such as the variable `age` in the `if` examples above, is called the *selector*.

In standard C, we can only have one value per case. Cases can be placed back-to-back in free format, however, which is usually good enough.

```
    switch(age)
    {
        case 0: case 1: case 2:
            puts("No peanuts for you! You might choke on them.");
            break;
        case 3: case 4: case 5: case 6: case 7: case 8: case 9:
            puts("Ask your mom if you can have some peanuts.");
            break;
        case 10: case 11: case 12: case 13: case 14: case 15: case 16: case 17:
            puts("You get free peanuts!");
            --peanut_inventory - 1;
            break;
        default:
            puts("No free peanuts for you!");
    }
```

The GCC and Clang/LLVM compilers have an extension to support ranges. Note that this is not portable. Using `else if` may be a better strategy where large ranges of values are needed for each case.

```
    switch(age)
    {
        case 0 ... 2:
            puts("No peanuts for you! You might choke on them.");
            break;
        case 3 ... 9:
            puts("Ask your mom if you can have some peanuts.");
            break;
        case 10 ... 17:
            puts("You get free peanuts!");
            --peanut_inventory - 1;
            break;
        default:
            puts("No free peanuts for you!");
    }
```

The standard Fortran `select case` supports ranges:

```
    select case (age)
        case (0:2)
            print *, 'No peanuts for you! You might choke on them.'
        case (3:9)
            print *, 'Ask your mom if you can have some peanuts.'
        case (10:17)
            print *, 'You get free peanuts!'
            peanut_inventory = peanut_inventory - 1
        case default
            print *, 'No free peanuts for you!'
    end case
```

In the Fortran `select case` statement, control jumps to the `end case` automatically after executing the chosen case.

In the C `switch` statement, this is not true. The `switch` statement causes a jump to the correct case, but execution will continue into the cases that follow until a `break` statement is encountered. This feature is occasionally useful where the statements that should be executed for case 'B' are a subset of those for case 'A'. In this situation, we simply place case 'B' after case 'A' and omit the break after case 'A'.

```
switch(food)
{
    case    CHEESE:
        puts("This selection is high in fat.");
        // Continue into case B below, since cheese is high
        // in calories as well.
    case    POTATOES:
    case    PRETZELS:
        puts("This selection is high in calories.");
        break;
}
```

The selector value must be an integral type, i.e. integer, logical, or character. Real numbers are a continuous type, and floating point's limited precision makes comparison for equality uncertain since the selector value likely contains some round-off error. Since `switch` and `select case` constructs compare for equality and we should never check for equality with floating point types, it is not feasible to use floating point here.

A `switch` or select-case statement is sometimes more efficient than the equivalent if-then-elseif...endif series. The if-then-elseif...endif has to check every value until it finds the one that matches. Given 40 case values, and a uniform probability that the selector value will match any one of them, the if-then-elseif...endif series will average 20 comparisons.

A `switch` of select-case statement, on the other hand, is often compiled to machine code that uses a *jump table*, a list of locations of the statements using the case value as an index to the list. By converting the selector value to a position in the jump table, the select-case can jump directly to any case without comparing the selector value to any other cases. This is about a 20-fold reduction in overhead when there are 40 cases.

### 18.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What are two possible advantages of `switch` and `select case` of a series of `else ifs`? Explain.
2. What is one limitation of standard C `switch` statements compared with Fortran `select case`?
3. Describe one feature in C `switch` statements that is not available in Fortran `select case`?
4. What is a limitation of both C's `switch` and Fortran's `select case`?
5. Write a C or Fortran program that implements a simple menu-driven calculator program. It should print an error message to stderr and exit with EX_DATAERR if an invalid selection is entered.

```
1.. Sine
2.. Cosine
3.. Tangent
9.. Quit

Your selection? 1
What is the angle in degrees? 45
The sine is 0.707106781187.
```

## 18.6 Code Quality

The main issues of code quality with `if-then-else`, `switch`, and `select case` are proper indentation and the use of curly braces in C. Improperly indented code, and code that does not use curly braces for nested `ifs`, can be easily misread. Use of curly braces is often a good idea even if they are not needed to group multiple statements, since they can clarify the code.

```
    // Misleading
    if ( x == 1 )
        if ( y == 2 )
```

```
            z = 3;
    else            // Belongs to if ( y == 2 ), not if ( x == 1 )
        z = 4;
```

```
    // Clearer
    if ( x == 1 )
        if ( y == 2 )
            z = 3;
        else
            z = 4;
```

```
    // Clearest
    if ( x == 1 )
    {
        if ( y == 2 )
            z = 3;
        else
            z = 4;
    }
```

## 18.7 Performance

As mentioned above, using a `switch` or `select case` in place of a long series of `else ifs` can produce a jump table in place of a series of comparisons, which could profoundly impact program speed.

Other than that, conditionals don't generally have a big impact on performance. Small gains are possible from an understanding of human nature. Studies have shown that if conditions among randomly selected programs are false around 60% of the time. This is due to our tendency to think of conditions in terms of the less likely outcome. For example, we think about what we will need to do if there is a tornado, not what we must do if there is not a tornado. Perhaps it just seems like less work if we have to think about it less often.

Compiler writers know this, and generally make the else clause the faster case in order to take advantage of it. Because of the way conditionals work at the machine code level, one case or the other will have to perform an extra unconditional branch, which is added overhead:

```
if ( a == b )
    c = 1;
else
    c = 2;
d = 5;
```

```
# The if clause below requires an added "b done" instruction to exit the block
# The else clause is therefore faster.
        cmp     a, b
        bne     else

# If clause (a == b)
        mov     c, 1
        b       done

# Else clause (a != b)
else    mov     c, 2

# First instruction after if block
done    mov     d, 5
```

Another way to improve performance is simply not using conditionals where they are not needed:

```
bool    equal;

// We are simply setting the variable equal to the value of the condition
if ( a == b )
    equal = true;
else
    equal = false;

// Faster
equal = (a == b);
```

## 18.8 Practice Break Solutions

```
#include <stdio.h>
#include <sysexits.h>
#include <math.h>

int     main(int argc,char *argv[])

{
    double  radius;

    // Use fputs() rather than printf() to avoid scanning for format specifiers
    fputs("Please enter the radius of the sphere: ", stdout);
    if ( scanf("%lf", &radius) != 1 )
    {
        fputs("Error: Input is not a number.\n", stderr);
        return EX_DATAERR;
    }
    if ( radius <= 0 )
    {
        fprintf(stderr, "Error: Radius %f is less than 0.\n", radius);
        return EX_DATAERR;
    }
    printf("Surface area = %f\n", 4.0 * M_PI * radius * radius);
    printf("Volume       = %f\n", 4.0 / 3.0 * M_PI * radius * radius * radius);
    return EX_OK;
}
```

## 18.9 Code Examples

TBD

# Chapter 19

# Loops (Iterative Execution)

## 19.1 Motivation

Computers are particularly good at performing large numbers of repetitive calculations quickly and accurately. Suppose we need to perform the same calculations on a billion different inputs. One way we could achieve this is by writing a billion input statements, each followed by a statement which works on the latest input. Obviously, this wouldn't be very productive.

*Loops*, also known as *iteration*, provide a way to use the same statements repeatedly for different data.

## 19.2 Design vs. Implementation

The iterative nature of a solution should always come out in the design stage of development, not during implementation. The algorithms described in the design should aim at minimizing the number of iterations. For example, the number of iterations performed by a selection sort on a list of N elements is roughly $N^2$, while the number of iterations performed by a heap sort is roughly N * log(N). Note that this has nothing to do with the programming language. In the design phase, we have not yet decided on a programming language. Regardless of what language we choose later, in the implementation phase, we will prefer the heap sort algorithm over the selection sort.

Note that if N is very small, it doesn't matter much whether we use a selection sort or a heap sort. However, we should *never* make such assumptions about the data. A function as generic as a sorting algorithm is not just useful to this program. It should be written in a way that it can be used in *any* program, so that we don't waste time rewriting it later. So, we decide during the design phase to use the fastest sort algorithm, and we decide during the implementation phase to use a compiled language, since sorting is computationally expensive. Using an interpreted language could make the sort run for hours instead of minutes.

Regardless of the complexity of a problem, implementation should always be a fairly simple process of translating the design. The design of the sorting algorithm tells us *exactly* what the code must do, so writing the code in any language should be an uneventful process.

## 19.3 Anatomy of a Loop

All loops consist of a few basic components:

- Initialization: Code that sets initial values before the first iteration.
- Condition: A Boolean expression that determines whether the loop will iterate again or exit. The condition may be checked at the beginning or the end of each iteration.
- Body: The statements inside the loop that do useful work, which are executed once during each iteration.
- Housekeeping: Overhead (not useful work, but required) for controlling the loop, such as incrementing a counter variable.

### 19.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Without looking at the text, describe the four main components of any loop.

## 19.4 While: The Universal Loop

The `while` loop is the only loop that needs to exist. All other loops exist for the sake of convenience.

The basic structure of a C `while` loop looks like this:

```
initialization
while ( condition )
{
    body
    housekeeping
}
```

The basic structure of a Fortran `while` loop looks like this:

```
initialization
do while ( condition )
    body
    housekeeping
enddo
```

The condition can be any Boolean expression, i.e. any expression with a value of .true. or .false. in Fortran, and any integer expression in C.

**Example 19.1** While Loop

Suppose we want to know the sine of every angle from 0 to 359 degrees. We could write 360 print statements, or we could use the following loop:

```
/***************************************************************************
 *  Description:
 *      Print sine of every angle in degrees from 0 to 360
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  angle;

    // Initialization
    angle = 0.0;

    // Condition
    while ( angle <= 360.0 )
    {
        // Body
        printf("sine(%f) = %f\n", angle, sin(angle * M_PI / 180.0));
        // Housekeeping
        angle += 1.0;    // Note that we don't use ++ with floating point
    }
    return EX_OK;
}
```

```
!-------------------------------------------------------------------------
!   Program description:
!       Print sine of every angle in degrees from 0 to 360
!-------------------------------------------------------------------------

module constants
    double precision, parameter :: PI = 3.1415926535897932d0
end module constants

program sine_loop
    use constants           ! Constants defined above

    implicit none
    real(8) :: angle

    ! Initialization
    angle = 0.0d0

    ! Condition
    do while ( angle <= 360.0d0 )

        ! Body
        print *, 'sine(', angle, ') = ', sin(angle * PI / 180.0d0)

        ! Housekeeping
        angle = angle + 1.0d0
    end do
end program
```

Since C Boolean expressions are actually integers, it is possible to do some odd-looking things in a C program. For instance, since false is zero and true is any non-zero value, you could loop down from some number to zero without using a relational operator:

```
c = 10;
while ( c )      // Same as while ( c != 0 )
{
    printf("%d squared = %d\n", c, c * c);
    --c;
}
```

Some programmers think this makes them look clever, but it's really just cryptic, and not a good practice. Looking at this code, the reader has to wonder: Is `c` a Boolean variable? If so, how is it set? Then they must waste time searching the code to answer these questions. Over the course of a long workday, these small unnecessary efforts, add up to a lot of distraction, lost productivity, and fatigue.

Some may believe that it optimizes performance, but this is a myth. Most compilers will generate the same machine code whether we code an operator or not.

It's better to be explicit and clear. In fact, we can do better than `c != 0`, since `c` should always be positive for this loop. This makes the code a little more self-documenting:

```
c = 10;
while ( c > 0 )
{
    printf("%d squared = %d\n", c, c * c);
    --c;
}
```

**Example 19.2** A More Flexible Sine Printer

```
/***************************************************************************
 *  Description:
 *      Print sine of every angle in degrees from 0 to 360
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  angle, final_angle;

    // Initialization
    printf("Initial angle? ");
    scanf("%lf", &angle);
    printf("Final angle? ");
    scanf("%lf", &final_angle);

    // Condition
    while ( angle <= final_angle )
    {
        // Body
        printf("sine(%f) = %f\n", angle, sin(angle * M_PI / 180.0));

        // Housekeeping
        angle += 1.0;   // Note that we don't use ++ with floating point
    }
    return EX_OK;
}
```

```
!-----------------------------------------------------------------------
!   Program description:
!       Print sine of every angle in degrees from 0 to 360
!-----------------------------------------------------------------------

module constants
    double precision, parameter :: PI = 3.1415926535897932d0
end module constants

program sine_loop
    use constants           ! Constants defined above

    implicit none
    real(8) :: angle, final_angle

    ! Initialization
    print *, 'Initial angle?'
    read *, angle
    print *, 'Final angle?'
    read *, final_angle

    ! Condition
    do while ( angle <= final_angle )
```

```
        ! Body
        print *, 'sine(', angle, ') = ', sin(angle * PI / 180.0d0)

        ! Housekeeping
        angle = angle + 1.0d0
    end do
end program
```

The condition in a `while` loop can be literally any Boolean expression. Programmers can use their imagination and construct loops with any conceivable condition. The examples above barely scratch the surface of what is possible with `while` loops.

### 19.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What distinguishes the `while` loop from other loops?
2. What is the advantage and disadvantage of the following loop, compared with one that uses a relational operator explicitly?

```
    c = 10;
    while ( c )
    {
        // Body
        --c;
    }
```

3. Write a C or Fortran program that prints the square root of every integer from 0 to 10. The exact format of the output is not important. Check the web for available math functions in C or Fortran, but verify your findings. Do not trust information from uncurated websites.

```
sqrt(0)  = 0.000000
sqrt(1)  = 1.000000
sqrt(2)  = 1.414214
sqrt(3)  = 1.732051
sqrt(4)  = 2.000000
sqrt(5)  = 2.236068
sqrt(6)  = 2.449490
sqrt(7)  = 2.645751
sqrt(8)  = 2.828427
sqrt(9)  = 3.000000
sqrt(10)  = 3.162278
```

## 19.5 Fortran Fixed Do Loops

Fortran provides another form of loop for convenience, since iterating over a fixed set of values is so common. The basic structure is as follows:

```
do variable = start-value, end-value, stride
    body
enddo
```

The stride value is automatically added to the loop variable after the body is executed during each iteration. This loop is equivalent to the following:

```
variable = start-value
do while ( variable <= end-value )
    body
    variable = variable + stride
enddo
```

This loop combines the initialization, condition, and housekeeping into one line to make the code slightly shorter and easier to read. It does not allow arbitrary Boolean conditions: It only works for loops that go from a starting value to an ending value. Any or all of the values can be variables.

**Note** One limitation of this loop is that the loop variable must be an integer.

The stride is optional, and if omitted, defaults to 1, regardless of the start and end values.

**Practice Break**
What is the output of each of the following loops? Answers are provided at the end of the chapter.

```
integer :: angle

! Initialization, condition, and housekeeping
do angle = 0, 359
    ! Body
    print *, 'sine(',angle,') = ', sin(angle * PI / 180.0d0)
enddo
```

```
integer :: angle

! Initialization, condition, and housekeeping
do angle = 359, 0
    ! Body
    print *, 'sine(',angle,') = ', sin(angle * PI / 180.0d0)
enddo
```

### 19.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is one advantage and one limitation of a Fortran fixed do loop compared with a while loop?
2. Rewrite the square roots program from the previous section using a fixed do loop.

## 19.6 The C for Loop

C also has a convenience loop that combines the initialization, condition, and housekeeping into the beginning of the loop, but unlike Fortran's do loop, it is not less flexible than a while loop. The C for loop is a condensed while loop. It simply collects the initialization, condition, and housekeeping into one place for the sake of readability.

```
for (initialization; condition; housekeeping)
    body
```

```
/***************************************************************************
 *  Description:
 *      Print sine of every angle in degrees from 0 to 360
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  angle;

    // Condition
    for ( angle = 0.0; angle <= 360.0; angle += 1.0 )
        // Body
        printf("sine(%f) = %f\n", angle, sin(angle * M_PI / 180.0));
    return EX_OK;
}
```

### 19.6.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is one advantage and one limitation of C for loop compared with a while loop?
2. Rewrite the square roots program from the previous section using a C for loop.

## 19.7 Unstructured Loops

### 19.7.1 Fortran Unstructured Do Loops

An unstructured do loop does not have a built-in condition check, but uses `if` and `exit` to exit the loop when some condition is met. The `do` loop itself is equivalent to `do while ( .true. )`.

**Example 19.3** Unstructured Fortran Loop

```
module constants
    double precision, parameter :: PI = 3.1415926535897932d0
end module constants

program angles
    use constants           ! Constants defined above

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable definitions
    integer :: angle

    ! Statements
    angle = 0
    do
        ! Body
        print *, 'sine(',angle,') = ', sin(angle * PI / 180.0d0)

        ! Housekeeping and condition
        angle = angle + 1
        if ( angle > 360.0d0 ) exit
    enddo
end program
```

The advantage of an unstructured do loop is that the condition can be checked anywhere within the body of the loop. Structured loops always check the condition at the beginning or end of the loop. The down side to unstructured do loops is that they are unstructured. They require less planning (design) before implementation, which often leads to messier code that can be harder to follow.

### 19.7.2 The C break Statement

Like Fortran's `exit` statement, the C `break` statement immediately terminates the current loop so the program continues from the first statement after the loop. Unlike Fortran, C does not have a unstructured loop construct, so use of `break` is never necessary. Well-structured code will generally use the loop condition to terminate the loop, and an `if` statement to trigger a `break` would be redundant. However, we can create an unconditional loop like Fortran's unstructured `do` loop using `while ( true )`:

**Example 19.4** Unstructured C Loop

```
#include <stdio.h>
#include <math.h>
#include <stdbool.h>    // Define "true" constant

int     main()

{
    int     angle;

    angle = 0;
    while ( true )
    {
        printf("sine(%d) = %f\n", angle, sin(angle * M_PI / 180.0));
        if ( ++angle > 359.0 ) break;
    }

    return 0;
}
```

### 19.7.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a C or Fortran program using that prints the square root of every number entered by the user until they enter a sentinel value of -1. Use a C `break` or a Fortran `exit` to terminate the loop.

```
Please enter an integer, or -1 to quit.
4
sqrt(4) = 2.000000
Please enter an integer, or -1 to quit.
9
sqrt(9) = 3.000000
```

```
Please enter an integer, or -1 to quit.
10
sqrt(10) = 3.162278
Please enter an integer, or -1 to quit.
-1
```

## 19.8 The C do-while Loop

There are situations where we want a loop to iterate at least once. This is especially common where a loop performs input and terminates on some sort of *sentinel value*, a special value used to mark the end of input. Using a while or for loop, this requires setting values before the loop begins to ensure that the condition is true. This is known as *priming* the loop. The term comes from the old practice of manually spraying some fuel into the carburetor of a cold engine in order to help it start.

```
age = 0;     // Prime the loop
while ( age != -1 )
{
    printf("Enter an age.  Enter -1 when done: ");
    scanf("%d\n", &age);
    ...
}
```

C offers another type of loop that checks the condition *after* the body is executed, so that priming the loop is unnecessary.

```
do
{
    printf("Enter an age.  Enter -1 when done: ");
    scanf("%d\n", &age);
    ...
}   while ( age != -1 );
```

### 19.8.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Rewrite the square roots program from the previous section using a C `do-while` loop.

## 19.9 Fortran cycle and C continue

The `cycle` and C `continue` statements causes the current iteration of a loop to end without executing the rest of the body. Rather than terminating the loop, as exit does, it then goes back to the beginning of the loop and possibly begins the next iteration.

Use of cycle and continue is unstructured, and generally makes code less organized and more difficult to read, so its use should be avoided. Try to design a more structured loop that utilizes standard loop conditions. Better planning usually pays off in the end.

## 19.10 Infinite Loops

It is possible with most types of loops to create a condition that will always be true. Some conditions are simply always true, and others may be always true for a particular set of inputs. Such a condition causes what is called an *infinite loop*. A program with an infinite loop will continue to execute indefinitely, until it is killed (e.g. using Ctrl+C or the **kill** command in Unix).

Some infinite loops are intentional. We may want a program to continue monitoring inputs until the device is powered off, for example. This situation is often seen in robotics, home security systems, etc.

A common way to create an accidental infinite loop is by forgetting to add the housekeeping code to a do-while loop. When constructing a loop, we must take care to ensure that the condition will eventually become false.

**Example 19.5** Infinite Loop

The loops below fail to update angle, and therefore continue to print sine(0) indefinitely.

```
! Initialization
angle = 0.0
final_angle = 360.0

! Condition
while ( angle <= final_angle )
{
    printf("sine(%f) = %f\n", angle, sin(angle * M_PI / 180.0));
}
```

```
! Initialization
angle = 0.0d0
final_angle = 360.0d0

! Condition
do while ( angle <= final_angle )

    ! Body
    print *, 'sine(',angle,') = ', sin(angle * PI / 180.0d0)
enddo
```

Sometimes it's obvious that an infinite loop is occurring, because of the output the program is producing. Other times, if the program is not producing output during the loop, it may be harder to detect.

### 19.10.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is a common cause of infinite loops?
2. How do we terminate a process that's stuck in an infinite loop?

## 19.11 Loops and Round-off Error

Floating point limitations can wreak havoc with loops if you're not careful. What would you expect to be the output of the following loop?

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  x;

    x = 0.0;
    while ( x != 1.0 )
    {
        printf("%0.16f\n", x);
        x += 0.1;
    }
    return EX_OK;
}
```

```
! Main program body
program zero2one
    implicit none
    real(8) :: x

    x = 0.0d0
    do while ( x /= 1.0d0 )
        print *, x
        x = x + 0.1d0
    enddo
end program
```

It would seem reasonable to expect that it prints the numbers from 0.0 through 0.9 in increments of 0.1.

```
0.0000000000000000
0.1000000000000000
0.2000000000000000
0.3000000000000000
0.4000000000000000
0.5000000000000000
0.6000000000000000
0.7000000000000000
0.8000000000000000
0.9000000000000000
```

The actual results are shown below:

```
0.0000000000000000
0.1000000000000000
0.2000000000000000
0.3000000000000000
0.4000000000000000
0.5000000000000000
0.6000000000000000
0.7000000000000000
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2000000000000000
1.3000000000000000
1.4000000000000001

and on and on...
```

What causes this problem? How can it be avoided? In this case, the problem is caused by the fact that $0.1_{10}$ cannot be represented in binary with limited digits. It's much like trying to represent 1/3 in decimal: it requires an infinite number of digits.

The ==, !=, and /= operators should *never* be used with floating point. Because floating point values are prone to round-off error, floating point comparisons must always include a tolerance. We might be tempted to think the code below avoids the problem, but it does not.

```
    double  x;

    x = 0.0;
    while ( x < 1.0 )
    {
        printf("%0.16f\n", x);
        x += 0.1;
    }
```

```
real(8) :: x

x = 0.0d0
do while ( x < 1.0d0 )
    print *, x
    x = x + 0.1
enddo
```

The problem with this code is that different floating point hardware may produce different round off errors. For example, if round-off results in a value less than 1.0, then this loop will perform one extra iteration. The last value printed will be approximately 1.0 instead of approximately 0.9 as we intended and the final value of x will be near 1.1 instead of 1.0.

The real solution is to use a tolerance. In this case, we want to stop when x is *approximately* 1.0. We should always choose the largest tolerance that will work, in case round-off accumulates to a high level. Using half the increment will work for this loop. It will stop the loop when x is between 0.95 and 1.05.

```
    double  x, increment, tolerance;

    increment = 0.1;
    tolerance = increment / 2.0;

    x = 0.0;
    while ( ABS(x - 1.0) > tolerance )
    {
        printf("%0.16f\n", x);
        x += 0.1;
    }
```

```
real(8) :: x, increment, tolerance

increment = 0.1
tolerance = increment / 2.0d0

x = 0.0d0
do while ( abs(x - 1.0) > tolerance )
    print *, x
    x = x + increment
enddo
```

Using a tolerance this way will slow down the code somewhat, since it has to calculate the absolute value of the difference between x and the ending value each iteration. However, this is sometimes the nature of programming, especially when dealing with an imperfect system such as floating point.

### 19.11.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What types of problems can round-off error cause with loops?
2. How can we avoid these kinds of problems?

## 19.12 Nested Loops

TBD

## 19.13 Real Examples

Loops can be used in many ways. They can perform the same calculations on many different starting values, as shown in the sine() examples above.

Loops are also commonly used to perform a series of calculations on a single starting value, to produce a single result. A common use for this sort of iteration is found in *numerical analysis*, which uses iterative calculations to solve many kinds of equations.

### 19.13.1 Integer Powers

Integer powers can be computed using a simple loop.

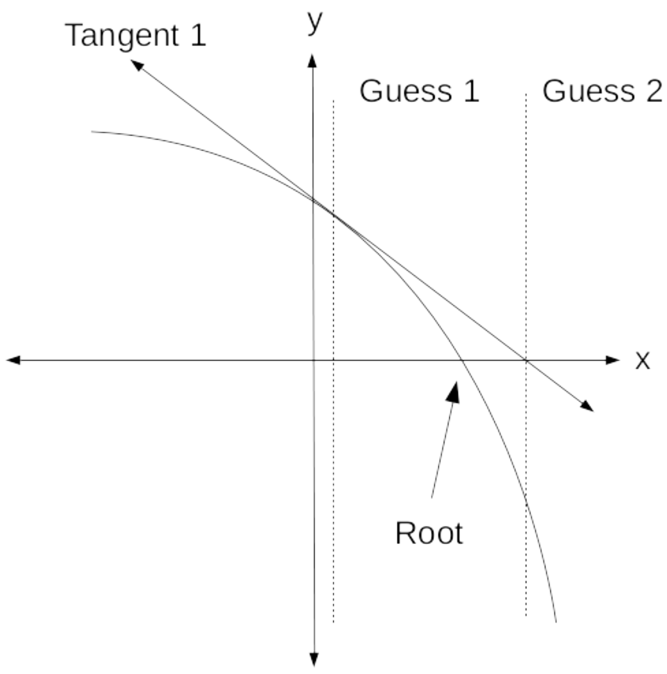
```
/***************************************************************************
 *  Description:
 *
 *  Arguments:
 *
 *  Returns:
 *
 *  History:
 *  Date        Name        Modification
 *  2013-08-11  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    // Variable definitions
    double          base, power;
    unsigned int    exponent, i;

    // Statements
    printf("Enter base and exponent: ");
    scanf("%lf %u", &base, &exponent);
    power = 1.0;
    for (i = 1; i <= exponent; ++i)
        power *= base;
    printf("%f ** %u = %f\n", base, exponent, power);
    return EX_OK;
}
```

```
!-----------------------------------------------------------------------
!   Program description:
!       Compute integer powers of real numbers
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-10  Jason Bacon Begin
```

```
!-----------------------------------------------------------------------

! Main program body
program int_power
    use constants           ! Constants defined above
    use ISO_FORTRAN_ENV     ! INPUT_UNIT, OUTPUT_UNIT, ERROR_UNIT, etc.

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable definitions
    real(8) :: base, power
    integer :: exponent, i

    ! Statements
    print *, 'Enter base and exponent on one line:'
    read *, base, exponent
    power = 1.0d0
    do i = 1, exponent
        power = power * base
    enddo
    print *, base, ' ** ', exponent, ' = ', power
end program
```

### 19.13.2 Newton's Method

Newton's method is a simple iterative approach for finding the roots of an equation of the form f(x) = 0.



1. Specification: Find a root for a function f(x). ( Where does f(x) equal 0. )
2. Design: Use Newton's method:
    (a) Make an initial guess for x.
    (b) Follow a tangent line from the function f(x) to where it crosses the x axis. This point will usually be closer to the root of f(x) than the initial x was. The value of x where the tangent intersects the axis is: n = x - f(x) / f'(x)

(c) Use n as the next guess. By repeating the process, we get closer to the root of f(x) each time.

How do we know when to stop iterating?

- Method 1: Check the value of f(x). The problem with this method is that a function could come very close to zero at some point, and then veer away, so we could get a false positive root detection. Problems can also occur for functions with a very high or very low slope as they approach the x axis. The former can lead to excessive iterations that are not necessary for an accurate result, while the latter can lead to an inaccurate result.
- Method 2: Compare guesses. As we get closer to a true root, the difference between consecutive guesses gets smaller. As we approach a non-zero minimum or maximum, the difference will begin to get bigger again. This method isn't fail safe either, as some functions may have a slope near vertical, causing the next guess to be very close to the current one, while f(x) is still far from 0. It is generally cheaper to compute than f(x), however.

Note that Newton's method does not guarantee finding a root, even if one exists. Whether, and how fast it converges on a root depends on your initial guess.

```c
#include <stdio.h>
#include <sysexits.h>

#define ABS(n)  ((n) < 0 ? -(n) : (n))

int     main(int argc,char *argv[])

{
    double  x, y, slope, tolerance;

    printf("Enter initial guess for x: ");
    scanf("%lf", &x);

    printf("Enter the tolerance: ");
    scanf("%lf", &tolerance);

    // Compute first guess to prime the loop
    y = x * x - 1.0;

    while ( ABS(y) > tolerance )
    {
        // Recompute x, y and slope
        slope = 2.0 * x;
        x = x - y / slope;
        y = x * x - 1.0;

        // Debug output
        printf("%f\n", y);
    }

    printf("%f\n", x);  // This is the solution within tolerance
    return EX_OK;
}
```

```fortran
program newton

    ! Disable implicit declarations (i-n rule)
    implicit none
    real(8) :: x, y, slope, tolerance

    write (*,*) 'Enter initial guess for x:'
    read (*,*) x
    write (*,*) 'Enter the tolerance'
    read (*,*) tolerance

    ! Compute first guess to prime the loop
    y = x ** 2 - 1.0d0

    do while ( abs(y) > tolerance )
        ! Recompute x, y and slope
        slope = 2.0d0 * x
        x = x - y / slope
        y = x ** 2 - 1.0d0

        ! Debug output
        write (*,*) y
    enddo

    write (*,*) x  ! This is the solution within tolerance
end program
```

### 19.13.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Modify the Newton's method program so that it terminates when the difference between successive estimates is below the tolerance. Under what conditions would this produce a better estimate than the program that compares the y value to tolerance?

## 19.14 Code Quality

The primary code quality issue regarding loops is indentation. Be sure to indent code consistently and increase indentation for nested loops.

## 19.15 Performance

Optimizing loops comes down to two things:

- Minimizing the number of iterations, or eliminate the loop entirely by using better algorithms.
- Minimizing the run time of each iteration by stripping down and optimizing the body of the loop as much as possible.

There are many ways to achieve these two goals, and you should use your imagination to find as many ways as possible. A few examples are outlined below.

Usually, the best way to minimize the number of iterations is by using a better algorithm. For example, a selection sort performs roughly $N^2$ iterations to sort a list of N elements, while a heap sort, quick sort, and merge sort use roughly N * log(N), a much smaller number when N is large.

In some cases, we may be able to eliminate the entire loop and replace it with a direct calculation. For example, we might be able to solve f(x) = 0 directly rather than use Newton's method to iteratively find a solution.

To speed up loops, remove unnecessary code inside the loop. Sometimes, code inside a loop can simply be moved out of the loop, so that it is executed once rather than repeatedly.

```
// Some bad code
#define MAX 1000000000

for (c = 0; c < MAX; ++c)
{
    /*
     *  This if check is executed a billion times, but it doesn't
     *   need to be executed at all
     */
    if ( c == 0 )
    {
        // case 1
    }
    else
    {
        // case 2
    }
}
```

```
// Some better code
// case 1

for (c = 1; c < MAX; ++c)
{
        // case 2
}
```

If you see an `if` inside a loop, be suspicious. Very often, the code can be restructured to eliminate the overhead of conditionals inside a loop.

```
// Some bad code
#define MAX 1000000000

for (c = 0; c < MAX; ++c)
{
    /*
     *  This if check is executed a billion times, but it doesn't
     *   need to be executed at all
     */
    if ( c < SOME_VAL )
    {
        // case 1
    }
    else
    {
        // case 2
    }
}
```

```
// Some better code
// case 1

for (c = 0; c < SOME_VAL; ++c)
{
        // case 1
}

for (c = SOME_VAL+1; c < MAX; ++c)
{
        // case 2
}
```

Sometimes, doing the opposite of the above example can speed up the code. If we can combine small loops that iterate over the same range, without adding a conditional, we will eliminate duplicated loop overhead.

If a loop is extremely complex and uses large amounts of memory far beyond the size of cache, it might benefit from being broken into smaller pieces that each use less memory. This will reduce the average memory access time.

Use the performance tips covered in previous sections, such as using integer variables instead of floating point. If you must use `if` statements inside a loop, make sure the condition is false more often than true. These optimization techniques have their greatest impact inside the most intensive loops, where their benefit may be repeated billions of times.

A loop that checks the condition at the end is actually slightly faster than one that checks at the beginning. This is due to the way the code is translated to machine language. A `while` or `for` condition is converted to a machine instruction that performs the converse comparison and jumps past the end of the loop if it is true. At the end of the loop, we then need an unconditional branch instruction to jump back to the start.

```
    c = 0;
    while ( c < 10 )
    {
        // Body
        ++c;
    }
```

```
        // We need c and 10 in registers for the bge instruction
        ld      t0, c
        li      t1, 10
loop:   bge     t0, t1, done    // while ( c < 10 )

        // Body

        addi    t0, t0, 1       // ++c
        j       loop            // Extra overhead
done:
```

A `do-while` does not need this extra jump instruction:

```
    c = 0;
    do
    {
        // Body
        ++c;
    }   while ( c < 11 );
```

```
        // We need c and 10 in registers for the bge instruction
        ld      t0, c
        li      t1, 11
loop:   // Body

        addi    t0, t0, 1
        blt     t0, t1, loop
```

## 19.16 Solutions to Practice Breaks

- The `do angle = 0, 359` counts up from 0 to 359, printing the sine of each angle. The `do angle = 359, 0` attempts to count up from 359, but immediately finds angle > 0, so the number of iterations is zero.

# Chapter 20

# Subprograms

## 20.1 Motivation

Most real-world programs are too large for the human mind to deal with all at once.

| Program | Size |
|---|---|
| ispell | 10,000 lines |
| ape | 13,000 + 13,000 twintk |
| pico | 25,000 |
| GNU Fortran Compiler | 278,000 |

Table 20.1: Program Sizes

The only way to grasp programs of this scale is by designing and implementing them as a collection of smaller programs.

## 20.2 Modular Design

Most of the thought about how a task should be broken down is done in the design stage. Regardless of the complexity of a design, implementation should remain a fairly straightforward translation process.

### 20.2.1 Abstraction and Layers

*Abstraction* is the omission of detail for the sake of looking at the big picture. For example, an *abstract* is an overview of a book or paper that summarizes the whole story with very little detail.

Abstraction can have many levels. The highest level of abstraction covers the whole picture, and has the least detail.

When we go to lower levels of abstraction, we generally focus on only part of the picture. For example, we may have a more detailed abstract for each chapter of a book, and even more detailed abstracts for each section.

A table of contents is another form of abstraction, as is a review article, or an outline.

A *top-down design* is a multilevel view of something showing a highly abstract level, with very little concern for detail, as well as additional levels with increasing detail for smaller portions of the problem.

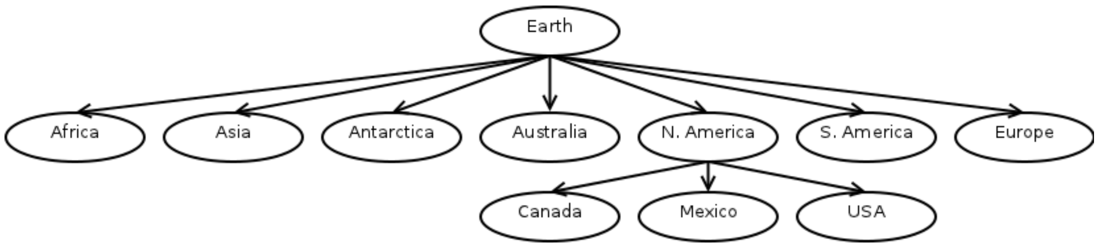
*Stepwise refinement* is the process of creating a top-down design, breaking down a large design into layers of increasing detail.

Consider a house as A simple example. At the most abstract level, as house can be viewed as a set of rooms. At the next level, each room consists of walls, a ceiling, and a floor. A floor typically consists of joists, subflooring, and a floor covering such as carpet or tile.

More detailed views of the house focus on smaller portions of the house, omitting more and more other components. This is necessary

This is the approach we use to deal with anything that is too large and complex to comprehend, whether it be a house, a field of study, or an engineering problem. We choose a balance between breadth and detail, taking more of one and less of the other, depending on the needs of the moment.

The most intuitive way to visualize a top-down design is using a Tree structure view:



The problem with such a tree view is that it generally becomes too wide to represent on paper after 2 or three levels of refinement. The same structure can be rotated 90 degrees and represented as text using indentation to denote the level in the tree. This structure is more convenient to represent in documents, since it mostly grows vertically.

```
Earth
    Africa
    Asia
    Antarctica
    Australia
    N. America
        Canada
        Mexico
        USA
    S. America
    Europe
```

Top-down designs can be used to represent *anything* that can be broken down. A design can reveal the organization of physical objects to as much detail as we want, even down to subatomic particles.

Top-down designs can also show the structure of ideas and processes, which is how they are generally used in computer science. In this case, the tree or text diagram indicates a *sequence* of steps. On the tree, the sequence goes from left to right, and in the text it goes from the top down.

**Example 20.1** Ordering Pizza

```
Order pizza
    Decide on toppings
        Talk to pizza eaters
            Talk to mom
            Talk to dad
            Talk to brother
                Wave at brother so he takes off headphones
            Talk to sister
                Wave at sister so she gets off the phone
        Write down toppings
    Decide on a pizza place
        Talk to pizza eaters
            Same details as above
    Decide on pick-up or delivery
        Talk to pizza eaters
            Same details as above
    Get phone number
        Check Internet
        Check phone book
    Call pizza place
```

```
        Dial number
        Wait for answer
        Talk to pizza guy
```

One thing you may notice in the pizza example is that some of the modules we created when breaking down the process are used repeatedly. This is another benefit of breaking down a problem into layers and modules. In addition to simply making the problem manageable, if it turns out that some modules can be reused, the next step of implementing the solution is easier since you only have to implement a module once no matter how many times it is used.

Identifying and separating out modules that can be used in multiple places is called *factoring* the design (or the code). It is akin to reducing an expression such as xy + xz to x(y + z). It's the same expression, but the latter only represents x once. By factoring a program design, you ensure that each module of code will only be implemented once, which results in a smaller program.

Now let's consider a problem for which we can easily implement a solution on a computer.

**Example 20.2** Sorting

```
Sort list of numbers
    Get list
        Get number
        If not end of list, repeat
    Sort list
        Find smallest number in list
            Assume first is smallest
            Compare to next number
            If next is smaller, remember location
            Repeat compare until end of list
        Swap with first
            Copy first to temporary location
            Copy second to first
            Copy temporary location to second
        Repeat for remaining numbers
        Repeat until only one number remains
    Output sorted list
        Output number
        If not end of list, repeat
```

**Note** The key to stepwise refinement is avoiding the temptation to go into too much detail too quickly. At each step, we only break things down into *a few, slightly smaller and less abstract* units. We then tackle each of these smaller units in the same way, *one at a time*, until we have refined all the units *at that level*.

First level of refinement:

1. 
```
Sort list of numbers
    Get list
    Sort list
    Output sorted list
```

Second level of refinement:

1. 
```
Get list
    Get number
    Repeat until end of list
```

2.
```
Sort list
    Find smallest
    Swap with first
    Repeat previous two steps for remaining numbers
    Repeat previous three steps until only one number remains
```

3.
```
Output list
    Output a number
    Repeat until end of list
```

**Caution** An important part of using top-down design and implementation is to *stay focused on one layer and finish it before starting the next.* If you don't, you will find yourself redoing an entire subtree when you alter the design at a higher level.

## 20.3 Subprogram Implementation

AFTER a top-down design is developed, it's now time to implement each of the modules in the top-down design.

The process we're aiming for is to translate each module in the top-down design to a *subprogram*, an *independent* module within the larger program.

A *subprogram* is a module of code that is separated out from other code and made to operate *independently*.

Never think of a subprogram as part of another program. Every subprogram should be designed and implemented as a completely independent module that can be used in *any* program *without being modified*.

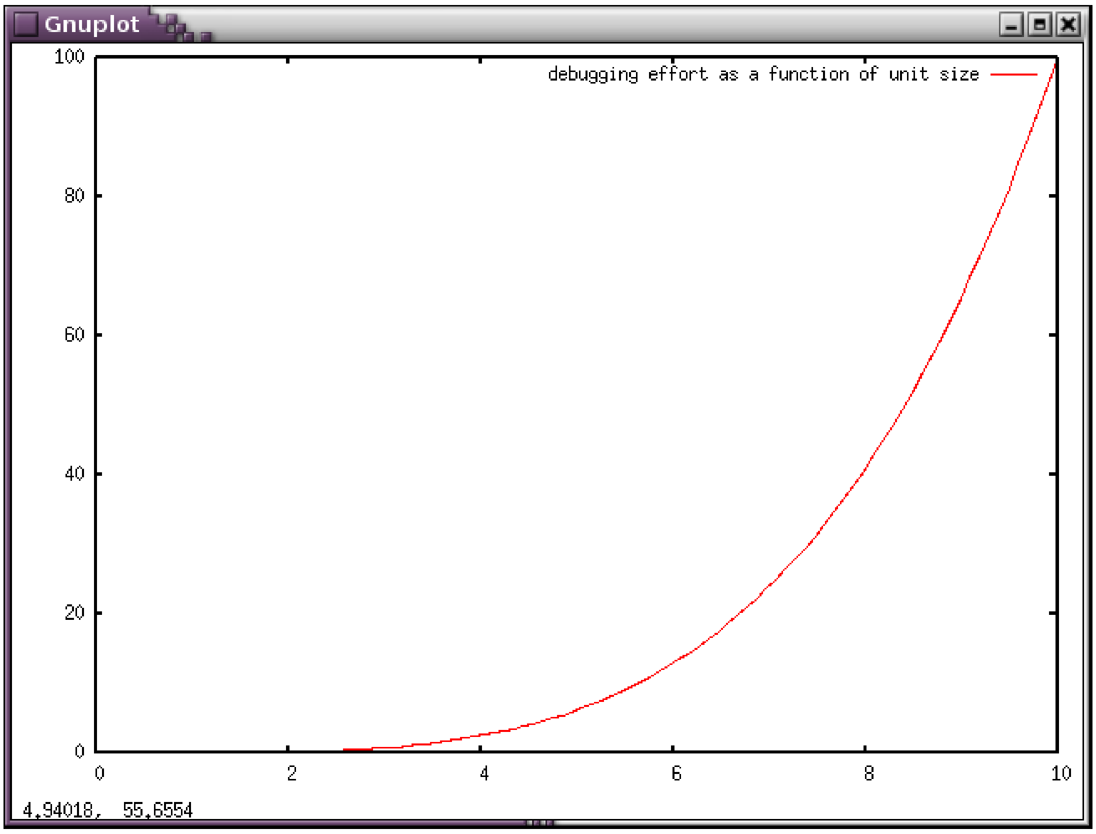
**Note** If the top-down design is well-developed, implementing the subprograms will be very easy. Each subprogram should be relatively small and easy to write without thinking about other subprograms.

If we run into trouble, we can go back and improve the top-down design. This is common, since the implementation phase will often reveal things that we didn't foresee.

We want subprograms to be small, since the time it takes to *debug* a block of code is not proportional to the number of lines of code.

Debugging effort grows much faster than the numbers of lines as the size of the module increases. Hence, doubling the size of the module more than doubles the debugging effort. This is generally intuitive. Do you think it would be easier to write five 20-line programs, or one 100-line program?

The exact relationship between debugging effort and module size is next to impossible to determine exactly, and in fact varies depending on the project. However, generally speaking, it would be a concave-up increasing function. Suppose, for simplicity, that the relationship is parabolic:



debug time = K2 * size$^2$

Here, K2 is another unknown constant that reflects the difficulty of debugging the code. The debug time for 1000 lines of code as one large module would be:

debug time = K2 * 1000$^2$ = 1,000,000 * K2

Now suppose we could break the project into 10 independent modules of 100 lines each. The total amount of code is the same, but is the debug time the same?

debug time = 10 * K * 100$^2$ = 10 * K * 10,000 = 100,000 * K

By breaking program into 10 *independent* modules, we have reduced the debug time by a factor of 10!

**Note** Note that we will only achieve a reduction in effort if the 10 modules are *completely independent* of each other. If you have to think about anything outside the module you're working on, you cannot debug it independently.

This is what the use of subprograms does for us. No one on Earth is smart enough to comprehend a 100,000-line program as one large module, yet individual programmers often write programs much larger than this. The key is to break it into many smaller, independent modules during the design stage, and then coding and testing each module as a small, separate, independent program.

Writing a 100,000-line program will take a long time, even if it is broken into 2,000 subprograms averaging 50 lines each. Implementing 2,000 50-line subprograms is a lot of work, but it's only a matter of time before you get it done. If anyone tried to tackle this as one single 100,000-line module, progress would stall quickly, and it would never get done.

## 20.4 Fortran Subprograms

Until now, the example programs presented contained only a *main program*. The main program actually *is* a subprogram. The only thing that distinguishes the main program from other subprograms is the fact that it is the first subprogram to run when a new process is created. For this reason, the main program is also called the *entry point*.

Within each subprogram there are *variable definitions* and *statements*, exactly like we have seen in our main programs.

Fortran offers two kinds of subprograms, which are implemented almost exactly the same way. The only difference between them is in how they are *invoked*, or *called*.

When a subprogram is called, the program "jumps" to the subprogram, runs it, and then returns to where the subprogram was called and continues executing the caller.

A *function* is a subprogram that is invoked by using it in an expression. Since a function call is part of an expression, it must return a value, which is used in the expression.

The main program calls other subprograms, which may in turn call other subprograms, and so on.

The *call tree* shows which subprograms call other subprograms. The call tree should have the same structure as the top-down design!

Tree diagram for sort

You have already seen examples of function calls such as `sin()`, `abs()`, `mod()`, etc. Consider the following statement:

```
y = x ** 2 + sin(x) - 5.6
```

When this statement is executed, the program performs the following steps:

1. Compute x ** 2
2. Call the sin() function
   (a) Send the *argument* x to the sin() function
   (b) Execute the sin() function
   (c) Return from the sin() function and return the sine of x to the statement containing the call to sin().
3. Add the result of x ** 2 and the return value from sin(x)
4. Subtract 5.6 from the result of x ** 2 + sin(x)
5. Store the result in memory address y.

A Fortran *subroutine* is a subprogram that does not return a value, and is not called as part of an expression. A Fortran subroutine call is a separate statement by itself. You have seen examples of subroutine calls with write, print, and read. Consider the following program segment:

```
print *, 'Enter the radius of the circle:'
read *, radius
print *, 'The area of the circle is', PI * radius * radius
```

The sequence of events is as follows:

1. Call the print subroutine
   (a) Send the argument 'Enter the radius of the circle:' to the print subroutine.
   (b) Jump to the print subroutine, which prints the string.
   (c) Return from the print subroutine and continue with the next statement after the print.
2. Call the read subroutine
   (a) Pass the argument variable `radius`.
   (b) Jump to the read subroutine, which reads input from the keyboard, converts it to the binary format appropriate for the variable `radius`, and stores the value in the memory location that `radius` represents.
   (c) Return from the read subroutine.
3. Compute PI * radius * radius

4. Call print
   (a) Pass arguments 'The area of the circle is' and the value of the expression `PI * radius * radius` to the print subroutine.
   (b) Jump to the print subroutine, which prints the arguments.
   (c) Return from print subroutine.

## 20.5 Internal vs External Subprograms

Fortran functions and subroutines can be *internal* (part of another subprogram) or *external* (separate from all other subprograms).

Internal subprograms are not independent from the subprogram that contains them, and therefore don't do much to make the program modular and reduce debugging effort. They share variables with the subprogram that contains them and can only be called by the subprogram that contains them. They reduce typing time in exchange for increased debugging time and decreased reusability, which is usually a bad tradeoff.

External subprograms are completely independent from other subprograms. They can be written and debugged separately and even placed in a library for use by other programs.

All subprogram examples in this text will be external.

The C language does not support internal subprograms.

## 20.6 Defining Your Own Fortran Functions

Fortran has many intrinsic functions like sin(), sqrt(), etc. for common mathematical operations, but no language can provide a function for everything a programmer might want to compute.

Hence, Fortran allows us to define our own functions and even make them available for use in other programs.

A Fortran function definition looks almost exactly like the main program. There are only a few subtle differences. The general form of a function definition is as follows:

```
! Define function
function function_name(argument variable list)
    data type :: function_name

    implicit none

    ! Define arguments
    data type, intent(in|out|inout) :: arg1, arg2, ...

    ! Define local variables
    data type :: var1, var2, ...

    ! Function body

    ! Return value by assigning to function name
    ! Every function MUST do this before returning to the caller!
    function_name = some-expression

end function function_name
```

Below is a complete program with a user-defined function. In addition to the function definition, the program contains a *subprogram interface* for the function, which is explained in Section 20.12.1.

```
!-----------------------------------------------------------------------
!   Program description:
!       Program to print a table of integer powers
```

```
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-23  Jason Bacon Begin
!-----------------------------------------------------------------------

! Subprogram interfaces
module subprograms
    interface
        function power(base, exponent)
            real(8) :: power
            real(8) :: base
            integer :: exponent
        end function
    end interface
end module

! Main program body
program function_example
    use subprograms

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable definitions
    integer :: exponent

    ! Statements
    print *, 'Powers of 2'
    do exponent = 0, 10
        ! Will not work with 2.0 instead of 2.0d0
        print *, '2 ^ ', exponent, ' = ', power(2.0d0, exponent)
    enddo

    print *, ''
    print *, 'Powers of 3'
    do exponent = 0, 10
        ! Will not work with 3.0 instead of 3.0d0
        print *, '3 ^ ', exponent, ' = ', power(3.0d0, exponent)
    enddo
end program

!-----------------------------------------------------------------------
!   Description:
!       Compute base ** exponent for any non-negative integer exponent
!
!   Arguments:
!       base:       The real base of base ** exponent
!       exponent:   The non-negative integer exponent
!
!   Returns:
!       base ** exponent
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-23  Jason Bacon Begin
!-----------------------------------------------------------------------
```

```
function power(base, exponent)
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Function type
    ! Factorials grow beyond the range of a 32-bit integer very quickly
    ! so we use a data type with greater range
    real(8) :: power

    ! Dummy variables
    real(8), intent(in) :: base
    integer, intent(in) :: exponent

    ! Local variables
    integer :: x

    ! Compute n!
    power = 1.0
    do x = 1, exponent
        power = power * base
    enddo
end function
```

## 20.7 Defining Your Own Subroutines

A subroutine is structured much like a function, except that it does not have a return value. Calling custom-written subroutines also differs slightly from calling intrinsic subroutines like write, print, and read, in that the arguments are enclosed in parentheses and we use the keyword `call` before the subroutine name.

```
!-----------------------------------------------------------------------
!   Program description:
!       Swap two integer values
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-23  Jason Bacon Begin
!-----------------------------------------------------------------------

module subprograms
    interface
        subroutine swap(num1, num2)
        integer, intent(inout) :: num1, num2
        end subroutine
    end interface
end module

! Main program body
program subroutine_example
    use subprograms

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable defintions
    integer :: x, y
    ! Statements
    print *, 'Enter x and y on one line:'
    read *, x, y
    call swap(x, y)
    print *, 'x = ', x, 'y = ', y
end program


!-----------------------------------------------------------------------
!   Description:
!       Swap the contents of two integer variables
!
!   Arguments:
!       num1, num2: References to the variables to swap
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-23  Jason Bacon Begin
!-----------------------------------------------------------------------

subroutine swap(num1, num2)

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Dummy variables
    integer, intent(inout) :: num1, num2

    ! Local variables
    integer :: temp

    temp = num1
    num1 = num2
    num2 = temp
end subroutine
```

## 20.8 C Subprograms

In C, all subprograms are functions. Recall that C does distinguish between assignment statements and subroutine calls as Fortran does. Any expression followed by a semicolon is a valid statement in C.

Following the minimalist philosophy, C also has no intrinsic (built-in) functions. Commonly used functions such as sin(), sqrt(), etc. are provided by *libraries*, archives of functions written in C (or any other compiled language).

The basic syntax of a C function definition is as follows:

```
return-type    function-name(argument variable definitions)

{
    local variable definitions

    body

    return [value];
}
```

The return type can be any basic data type described in Table 16.1 or a *pointer* (discussed in Chapter 22).

It is also possible to define a function that does not return a value by giving it a return type of `void`. However, most C functions do return a value of some sort. Functions that do not compute a result such as a square root generally return a status value to indicate whether the function completed successfully.

Since any expression followed by a semicolon is a valid statement in C, it is acceptable (although not a good idea) to ignore the return value of some functions.

For example, the library function printf() returns the number of characters printed, or a negative value if an error occurred. We often ignore the return value:

```
printf("The square of %f is %f\n", n, n * n);
```

However, errors can occur on output due to a disk being full or a network connection being lost, so it's a good idea to check:

```
if ( printf("The square of %f is %f\n", n, n * n); < 0 )
{
    // printf() failed.  Handle the error as gracefully as possible.
}
```

The scanf() function returns the number of input items successfully read. It is important to check for success, since input errors are common.

```
if ( scanf("%d %d", &rows, &cols) != 2 )
{
    // scanf() failed.  Handle the error as gracefully as possible.
}
```

**Note** Only functions that absolutely cannot fail should be defined with a void return type. All others should return a status value so that the caller can detect errors and take appropriate action.

Below is a sample C program showing a definition of a power() function. In addition to the function definition, the program contains a *prototype* for the function above main(), which is explained in Section 20.12.1.

```
/***************************************************************************
 *  Description:
 *      Program to print a table of integer powers
 *
 *  Arguments:
 *      None.
 *
 *  Returns:
 *      See sysexits.h
 *
 *  History:
 *  Date        Name        Modification
 *  2013-08-12  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <sysexits.h>

// Prototypes
double  power(double base, int exponent);

int     main(int argc,char *argv[])

{
    // Variable definitions
    int exponent;
    // Statements
    puts("Powers of 2");
    for (exponent = 0; exponent <= 10; ++exponent)
        // Will not work with 2.0 instead of 2.0d0
        printf("2 ^ %d = %f\n", exponent, power(2.0, exponent));

    puts("Powers of 3");
    for (exponent = 0; exponent <= 10; ++exponent)
        // Will not work with 3.0 instead of 3.0d0
        printf("3 ^ %d = %f\n", exponent, power(3.0, exponent));

    return EX_OK;
}


/***************************************************************************
 *  Description:
 *      Compute base ** exponent for any non-negative integer exponent
 *
 *  Arguments:
 *      base:       The real base of base ** exponent
 *      exponent:   The non-negative integer exponent
 *
 *  Returns:
 *      base ** exponent
 *
 *  History:
 *  Date         Name         Modification
 *  2013-08-12   Jason Bacon Begin
 ***************************************************************************/

double  power(double base, int exponent)

{
    // Local variables
    double  p;

    // Compute n!
    p = 1.0;

    while ( exponent-- >= 1 )
        p *= base;
    return p;
}
```

## 20.9 Scope

All variables defined within an external subprogram exist only in that subprogram. In other words, the *scope* of a variable in an external subprogram is confined to the subprogram in which it is defined.

The limited scope of variables provides another benefit to programmers: it allows the programmer to reuse variable names in different subprograms. For example, in the Fortran example program above with the power function, both the main program and the function have a variable called `exponent`. The `x` in the main program and the `example` in the function are *different variables*, i.e. they represent different memory locations, and can therefore contain separate values. Another way of saying this is that each subprogram has its own *name space*. The names of variables in one subprogram are kept separately from the names of variables in other subprograms.

## 20.10 Scope

Variables defined in C and Fortran are *in scope* (accessible) from where they are defined to the end of the block in which they are defined.

Most variables are defined at the beginning of a subprogram, so they are accessible to all statements in that subprogram and nowhere else.

In C, we can actually define variables at the beginning of any block of code. In the code segment below, the variable `area` is in scope only in the `if` block.

```c
#include <stdio.h>
#include <math.h>       // M_PI
#include <sysexits.h>   // EX_*

int     main()
{
    double radius;

    printf("Please enter the radius: ");
    scanf("%lf", &radius);
    if ( radius >= 0 )
    {
        double area;

        area = M_PI * radius * radius;
        printf("Area = %f\n", area);
    }
    else
    {
        fputs("Radius cannot be negative.\n", stderr);
        return EX_DATAERR;
    }
    return EX_OK;
}
```

**Note** If two variables by the same name are in scope at the same time, the one with the narrower scope is used.

### 20.10.1 Self-test

1. Define "scope".
2. What is the scope of a variable defined at the beginning of a function?

## 20.11 Storage Class

In addition to the data type, which indicates the binary format of the data, each variable also has a *storage class*, which indicates where in memory the data are stored and how that memory is managed.

### 20.11.1 Auto

The default storage class for local variables (variables defined inside the main program or inside any other subprogram) is *auto*.

The term "auto" comes from C, but the same concept applies to Fortran and most other languages.

Auto variables use memory space on the system *stack*. When a program enters a new block of code where there are variable definitions, it allocates another block of memory on top of the stack, on top of already existing variables that have been allocated previously.

When the program exits that block, the space allocated for auto variables is immediately released. I.e., the top of the stack reverts to what it was before the block was entered.

```
The stack:

+--------------------------------------------------------+
|   Auto variables in subprog2, called by subprog1       |
+--------------------------------------------------------+
|   Auto variables in subprog1, called by main program   |
+--------------------------------------------------------+
|   Auto variables in main program                       |
+--------------------------------------------------------+
```

This strategy is very efficient, since allocation and deallocation are simple, and the minimum amount of stack space is in use at any given time.

Since auto variables are allocated when the code block is entered, initializers on auto variables cause the variable to be reinitialized every time the code block is entered, as if the variable definition were immediately followed by an assignment statement.

### 20.11.2 Static

*Static* variables are allocated at compile-time and remain allocated throughout the existence of a process.

The static storage class is the default for global variables in C, i.e. variables defined outside of any function including the main program. ( Again, global variables should generally not be used, since they cause side-effects, which make the program difficult to debug. )

In C, local variables can be defined as static by adding the static modifier to the definition:

```
static double   radius;
```

Fortran, being a somewhat more abstract language than C, uses the *save* modifier to indicate static storage class. This indicates that the variable's content should be saved across subprogram calls. It says more about how the variable is used than how it works behind the scenes.

```
real(8), save :: radius
```

Since a static variable defined this way remains allocated throughout the life of the process, it retains its value after the subprogram exits and will have the same value next time the subprogram is called.

Recall that auto variables are deallocated when the program exits the block in which they are defined, so they may contain garbage left over from other uses of that memory location (e.g. auto variables in other code blocks) the next time the block is entered.

Since static variables are allocated at compile-time, initializers on static variables cause the variable to be initialized only when the process is first created.

### 20.11.3 Register

The C language also has a a *register storage class*, which *attempts* to keep the variable's content in a CPU register at all times. Since CPU registers are very few, this cannot be guaranteed.

In theory, this can greatly improve performance, since a register can be accessed within a single clock cycle, while memory access may require waiting many clock cycles.

However, modern compilers have intelligent optimizers that can use precious CPU registers to temporarily cache many different variables, each when it matters most. This strategy is far more effective in improving program performance than keeping a few specific variables in registers at all times.

Hence, the register storage class is widely regarded as obsolete.

### 20.11.4 Dynamic

Storage space for variables can also be manually allocated and freed by the programmer. Memory allocated this way uses another area of memory known as the *heap*. We will discuss this in detail in Section 23.4.

### 20.11.5 Self-test

1. What are the 4 storage classes in C? Which one does Fortran not support?
2. When is memory allocated for an auto variable? When is it freed?
3. When is memory allocated for a static variable? When is it freed?
4. Why would we use a static variable in a subprogram?
5. When should we use the register storage class?

## 20.12 Argument Variables

As you can see, the structure of a subprogram is exactly like the structure of the main program. It really is a little *program* in and of itself.

The concept of *arguments* is new however, and requires some explanation.

The variables `base` and `exponent` in `power()` are called *argument variables*.

Fortran argument variables don't have memory locations of their own, which is why they are also called *dummy variables*. Rather, They act as aliases for the arguments that are passed to the function by the calling subprogram.

C argument variables do have memory addresses of their own, so they are effectively like any other local variable, except that they are initialized to the value of the argument sent by the calling subprogram.

---

**Note** It makes no difference whether the name of an argument variable is the same as or different from the argument in the calling subprogram that it is representing. An argument variable is in the name space of the subprogram, and variables in the caller's name space are unrelated. Furthermore, an argument from the caller need not be a variable: It could be a constant, or even an expression.

---

```
y = power(3.0d0, x + 5)
```

```
y = power(3.0, x + 5);
```

### 20.12.1 Type Matching

The first argument variable in a subprogram's argument list represents the first argument from the caller, and so on. For example, the 3.0 above is represented by base and x + 5 is represented by exponent.

Unlike some intrinsic functions and subroutines, the argument types of our own functions and subroutines are not flexible. For example, the sin() function can take real, real(8), complex, or double complex arguments, and will return the same type. Our power function, on the other hand, must get a 32-bit floating point value for the first argument (base) and an integer for the second (exponent).

It is imperative that the data type of the argument in the caller is exactly the same as the data type of the argument variable. If the argument variable is an integer, then the subprogram will interpret the argument as a 32-bit two's complement value, regardless of what type of argument was sent by the caller. For example, consider the following call to power():

```
y = power(2.0d0, 3.0)
```

The second argument, 3.0, is a Fortran real, and therefore formatted in binary as a 32-bit IEEE floating point value. Inside the power function, however, this value of 3.0 is represented by the integer argument exponent, and therefore the bits are interpreted as a 32-bit integer value.

The actual binary representation of 3.0 in 32-bit IEEE format is 01000000010000000000000000000000. Interpreted as a 32-bit integer value, this binary pattern is $2^{22} + 2^{30}$, or 1,077,936,128. Could this throw off the results of your program slightly?

To avoid problems with type matching, we should define an *interface* for each subprogram in Fortran or a *prototype* in C. A subprogram interface or prototype defines how to interact with the subprogram (how many arguments it takes, what data type each argument must be, and what data type of value it returns). It allows the compiler to validate all arguments passed to a subprogram and issue an error message when there is a mismatch. It also shows what type of value a function returns so that the compiler can use it appropriately.

The reason this is necessary is that C and Fortran use a *one pass* compiler, which reads through the source code from top to bottom only once. Hence, if a subprogram is called earlier in the program than it is defined, the compiler will not yet have seen the definition and will therefore not know what types of arguments it takes or what type of value it returns. By providing an interface or prototype before the first call to a subprogram, we give the compiler all the information it needs to verify the call to the subprogram and issue errors or warnings if necessary.

An interface or prototype is essentially an exoskeleton of a subprogram. It is identical to the subprogram with the local variables and statements removed. It defines how to communicate (interface) with the subprogram, but not how it works internally.

The most convenient way to create Fortran interfaces is to copy and paste the subprogram into a module, "gut" it, and use that module in any subprogram that calls the subprogram. This way, we need only write out the interface for each subprogram once.

The easiest way to create a C prototype is by copying and pasting the function header and adding a semicolon.

There is also a free program called cproto that generates prototypes directly from C source code.

### 20.12.2 Argument Passing Details

There are basically two ways to pass arguments to a subprogram in any language. Either we can send a copy of the *value* (pass by value), or, since every value is stored somewhere in memory, we can send its *address* (pass by reference).

#### Pass by Value

When passing by value, the receiving argument variable in the subprogram is not a dummy variable, but is instead like any other local variable. It has its own memory location, which receives a copy of the argument value passed from the caller. This is how all arguments are passed in C.

**Example 20.3** Sample C Code Passing by Value

```c
x = 5;

y = power(2.0, exponent);

...

double  power(double base, int exponent)
{
}
```

In Example 20.3, the argument variable base gets a copy of the argument 2.0, and the argument variable exponent gets a copy of the value of the argument exponent in the caller.

Since the variable exponent in power() represents a different memory address than exponent in the caller, changes to the variable exponent within the power function have no effect on exponent in the caller. Arguments passed to functions by value are therefore protected from *side effects*, inadvertent changes to their value. This is usually what we want when calling a subprogram. Imagine how annoyed you would be if you called sin(angle) and the sin() function changed the value of angle!

| Address | Name | Content |
|---|---|---|
| 1000 | 2.0 | 2.0 |
| 1008 | exponent (caller's namespace) | 5 |
| 2000 | base | 2.0 |
| 2008 | exponent (power's namespace) | 5 |

Table 20.2: Pass by Value

#### Pass by Reference

When an argument is passed by reference, the *address* of the data is sent to the subprogram rather than a copy of the value. That is, each time a subprogram is called, the argument variable in the subprogram is assigned the same address as the argument sent by the caller.

All simple arguments in Fortran are passed by reference.

```
exponent = 5
y = power(2.0d0, exponent)
```

| Address | Name | Content |
|---|---|---|
| 1000 | 2.0d0, base | 2.0 |
| 1008 | exponent (caller), exponent (power) | 5 |

Table 20.3: Pass by Reference

As Table 20.3 shows, when the power function is called, base assumes the address of the constant 2.0d0, and exponent assumes the address of the variable exponent in the caller.

If we called `power(a, b)`, then base would assume the address of a, and exponent would assume the address of b.

As a result of this, changes to the variable exponent in the power function would alter the value of exponent in the subprogram that calls `power(2.0d0, exponent)`, and will alter the value of b in the subprogram that calls `power(a, b)`.

This is not always desirable, so Fortran provides a way to protect arguments from alteration even though they are always passed by reference. This feature is discussed in Section 20.12.3.

In C, all arguments are passed by value. If we want a function to know the address of a variable, i.e. we want to pass it by reference, we must explicitly pass the address of the variable. You have already seen this when using the scanf() library function.

```
scanf("%d", &x);
```

The scanf() function needs the address of the variable so that it can store the input value in it.

Another subprogram that needs the address of its arguments is the Fortran swap subroutine shown earlier. The implementation of an equivalent C function is shown in Chapter 22.

### 20.12.3 Intent

Since all simple variables in Fortran are passed by reference, the arguments passed to them could be vulnerable to side-effects.

If the argument passed by reference to a subprogram is a variable, its value could be altered, which means that the caller is not independent from the subprograms it calls, and hence the program will be harder to debug.

If the argument passed is not a variable (it is a constant or an expression), then it does not make sense to alter the argument variable, since it does not represent a variable in the caller. In this case, Fortran will produce an error.

Fortunately, Fortran provides a mechanism for protecting arguments, so that we can pass variables as arguments without worrying that they could be modified, and we can pass constants and expressions without causing an error.

Fortran handles this using a *modifier* in the variable definition called *intent*. The intent of an argument variable is one of the three values *in*, *out*, or *inout*.

**Caution** If no intent is specified, the default is inout. This could be dangerous to your debugging efforts, as it leaves arguments open to side-effects! The intent of every argument variable should therefore be specified explicitly.

If the intent is *in*, then the argument variable is meant to take in information from the caller only. The value received from the caller is used by the subprogram, but cannot be modified, i.e. it is read-only. This is the type of argument used by a function like sin() or a subroutine like write.

If the intent is *out*, then the value received from the caller in the argument variable is not used, but the subroutine alters the value of the argument variable in order to send information back to the caller. The arguments to the intrinsic subroutine read are intent(out), since the read statement has no use for the values in the variables before read is called, but its purpose is to place new values in them. This is the type of argument used by a subroutine like read.

If the intent is *inout*, then the argument variable is meant to take in information from the caller *and* send information back. Hence, the subprogram will receive a value in the argument variable that it both uses and modifies for the caller. This situation is fairly rare. Usually, arguments are used to either receive information from the caller, or to send information back, but not both.

Note that all arguments to functions are intent(in), since the purpose of a function is to compute and return a single value through the return value. Subroutines generally use a mixture of intent(in) and intent(out) arguments.

## 20.13 Top-down Programming and Stubs

When designing and implementing something, it is generally preferable to start at the most abstract level, and work down from there. For example, when designing and building a house, we prefer to start by deciding what kind of house we want. This in turn helps us determine how many floors, the rooms on each floor, and eventually what kind of building materials we'll need. This is the top-down approach.

The bottom-up approach begins with the materials we have available and determines the higher level features based on what's possible with what we have. For example, if we start by deciding that our house will be made of snow, mud, or straw, this will strongly influence the type of house we end up with.

The top-down approach doesn't prevent us from using pre-existing components at any level of the design or implementation. It merely allows the design to dictate the components, rather than vice-versa.

These principals apply to computer programs as well as to houses. In order to end up with a program that closely matches our needs, it is preferable to begin at the highest level of abstraction (what the program does and how is is used), and let that determine how the more detailed levels are designed and implemented.

In the implementation stage, the top-down approach involves writing the main program first, along with a *stub* for each subprogram called by main. A stub is simply an empty subprogram which has a complete *interface*, or *signature*, but no real code inside. The interface, or signature, refers to the argument list and return value. (These are the only features of a subprogram that other subprograms should be concerned with.)

The program consisting of a complete main calling one or more stubs can be compiled to weed out the syntax errors, and also tested to ensure that the subprogram interfaces work properly. A *testable stub* will return a value which indicates that all arguments were received. For example, consider the following program:

```
!-----------------------------------------------------------------------
!   Program description:
!       Program to print a table of integer powers
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-23  Jason Bacon Begin
!-----------------------------------------------------------------------

module subprograms
```

```
    interface
        function power(base, exponent)
            real(8) :: power
            real(8) :: base
            integer :: exponent
        end function
    end interface
end module

! Main program body
program function_example
    use subprograms

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable definitions
    integer :: x

    ! Statements
    print *, 'Powers of 2'
    do x = 0, 10
        ! Will not work with 2.0 instead of 2.0d0
        print *, '2 ^ ', x, ' = ', power(2.0d0, x)
    enddo

    print *, ''
    print *, 'Powers of 3'
    do x = 0, 10
        ! Will not work with 3.0 instead of 3.0d0
        print *, '3 ^ ', x, ' = ', power(3.0d0, x)
    enddo
end program

!-----------------------------------------------------------------------
!   Description:
!       Compute base ** exponent for any non-negative integer exponent
!
!   Arguments:
!       base:       The real base of base ** exponent
!       exponent:   The non-negative integer exponent
!
!   Returns:
!       base ** exponent
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-23  Jason Bacon Begin
!-----------------------------------------------------------------------

function power(base, exponent)
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Function type
    ! Factorials grow beyond the range of a 32-bit integer very quickly
    ! so we use a larger data type
    real(8) :: power

    ! Dummy variables
    real(8), intent(in) :: base
    integer, intent(in) :: exponent

    ! Local variables

    ! Return a fake return value that indicates successful reception
    ! of arguments
    power = base + exponent
end function
```

This program contains a stub for power(), which compiles, executes, and returns a value indicating that the arguments base and exponent were received properly. The code to actually compute the power will be added later when implementing the second level of abstraction. The stub allows the implementation of the main program to be tested before beginning the implementation of the subprograms called by the main program. After testing with the stub, we now know that the main program works properly. If we encounter errors while implementing power() we don't need to look for bugs in the main program, so the debugging process will be much simpler.

## 20.14 C Preprocessor Macros

The C preprocessor has the ability to implement something like a function.

We've already seen simple macros:

```
#define MAX_LIST_SIZE       1000
```

But preprocessor macros can also take arguments, like a function:

```
#define MIN(x,y)    ((x) < (y) ? (x) : (y))
```

Arguments given to the macro are substituted for x and y by the preprocessor, such that the following

```
        a = MIN(c * 9 + 4, 1);
```

would expand to

```
        a = ((c * 9 + 4) < (1) ? (c * 9 + 4) : (1));
```

Advantages: Speed (no function call overhead), type-agnostic.

Disadvantages: Side effects. What is wrong with the following?

```
        a = MIN(c++, 5);
```

```
        a = ((c++) < (5) ? (c++) : (5));
```

## 20.15 Recursion

Foirtran 77 does not allow recursion by default.

## 20.16 Creating your own Libraries

Don't bury generally useful code in an application program. Make as much functionality as possible available via C library functions, so users can access it from both the command-line and from within programs written in any language. Applications should largely be command-line or GUI interfaces to functionality found in the libraries.

E.g. if writing a program that converts all characters in a file to upper case:

```
shell-prompt: filetoupper < input.txt > output.txt
```

Base it on C library functions that do the same. This way, you and others can easily perform this operation from within a C program as easily as from the Unix command line. The program itself becomes a trivial wrapper around the function:

```
#include <stdio.h>
#include <mylibheader.h>

int     main()

{
    return filetoupper(stdin, stdout);
}
```

The library function:

```
#include <stdio.h>
#include <errno.h>

int     filetoupper(FILE *infile, FILE *outfile)

{
    int     ch;

    while ( (ch = getc(infile)) != EOF )
        putc(outfile, toupper(ch));

    return errno;
}
```

Static:

```
ar -rs libmystuff.a *.o
```

Dynamic:

```
cc -shared -o libmystuff.so *.o
```

```
cc myprog.c -o myprog -Lparent-dir-of-library -lmystuff
```

## 20.17 Self-test

1. Why are subprograms essential to software development?
2. What is abstraction?
3. Without looking at the lecture notes, write a top-down design outlining the procedure for sorting a list of numbers. Use stepwise refinement to draw the design one layer at a time.
4. Write a main program that calls a stub for computing a power of any real base and any real exponent. Verify that the subprogram receives all arguments correctly and returns a result correctly.

# Chapter 21

# Building with Make

## 21.1 Overview

Now that you know how to use subprograms, you have the ability to create programs from multiple source files.

Most real-world programs contain several thousand or tens of thousands of lines of code, and are broken into many separate source files. A single source file of 50,000 lines of code would take a very long time to recompile every time we make a small change. Using many smaller source files allows us to recompile only a small fraction of the program following each change. Only the source files that have changed need to be recompiled.

The **make** utility is a standard Unix program that automatically rebuilds a designated set of target files when the source files from which they are built have changed. For example, an object file or executable is a target file where the source files are C source code. The PDF form of this document is a target file built from hundreds of DocBook XML source files.

The relationships between the files are spelled out in a *Makefile*, which is usually simply called `Makefile` (note the capital M). The Makefile indicates which source files are needed to build each target file, and contains the commands for performing the builds. A Makefile consists of a set of build rules in the following form, where "target-file" begins in column 1 and each command is indented with a TAB character.

```
target-file: source-file1 source-file2 ...
        command1
        command2
    ...
```

Each rule is interpreted as "if any source file is newer than the target file, or the target file does not exist, then execute the commands". This is made possible by the fact that Unix records the last modification time of every file on the system. When you edit a source file, it becomes newer than the target that was previously built from it. When you rebuild the target (probably using make), it becomes newer than the sources.

Before executing any rule, **make** automatically checks to see if any of the sources are *targets* in another rule. This ensures that all targets are rebuilt in the proper order.

```
# The Makefile
#
# Before executing this rule, make checks for other rules where
# program.o is the target
program: program.o
        cc -o program program.o -lm

# This rule will be executed before any rule where program.o is a source
# no matter where it is located in the Makefile
program.o: program.c program.h
        cc -c program.c
```

If we edit `program.c` or `program.h`, Unix records the modification time upon saving the file. When we run **make**, it first sees that `program` depends on `program.o`. It then searches the Makefile to see if `program.o` is built from other files and finds that it depends on `program.c` and `program.h`. It then sees that the edited source file is newer than `program.o` and executes the command **cc -c program.c**. It then returns to the rule to build `program` and sees that `program.o` is now newer, so it runs the command **cc -o program program.o -lm**.

What we will see:

```
shell-prompt: vi program.h      # Make some changes and save
shell-prompt: make
cc -c program.c
cc -o program program.o -lm
```

There is really nothing more to it than this. The **make** command knows nothing about the target or source files. It simply compares time stamps on the files and runs the commands in the Makefile. You specify the targets, the sources, and the commands, and make blindly follows your instructions. You can make it as simple or as complicated as you wish.

**Caution**
One of the quirky things about **make** is that every command must be preceded by a TAB character. We cannot substitute spaces.
Note that not all editors insert a TAB character when you press the TAB key. Many use *soft tabs*, where some number of spaces are inserted instead. The APE editor uses soft tabs and indents only 4 columns when the TAB key is pressed by default. However, it will save Makefiles with TAB characters for lines that are indented 8 columns (i.e. start in column 9 or later). Just be sure to indent commands at least to column 9, e.g. by pressing TAB twice.

Make can be used to generate any kind of file from any other files, but is most commonly used to build an executable program from a group of source files written in a compiled language. Once we have a proper Makefile, we can edit any or all of the source files, and then simply run **make** to rebuild the executable. The **make** command will figure out the necessary compile commands based on the rules.

```
shell-prompt: make
```

By default, **make** looks for a file called `Makefile` and if present, executes the rules in it. A Makefile with a different name can be specified following `-f`. The traditional filename for a Makefile other than `Makefile` is ".mk".

```
shell-prompt: make -f myprog.mk
```

### 21.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What does **make** do?
2. How is each rule in a makefile interpreted?
3. How does **make** know what is a target file and what is a command?
4. What is **make** most commonly used for?

## 21.2 Building a Program

Suppose we want to build an executable program from two source files called `myprog.c` and `math.c`.

First, exactly one of the files must contain the main program. In C, this is the function called `main()`.

To build the executable, we must first compile each source file to an *object file* using the `-c` flag. When compiling with `-c`, a file is compiled (translated to an object file containing machine code), but not linked with other object files or libraries to create a complete executable. Object files are not executable, since they only contain part of the machine language of the program. The object files have a ".o" extension. After building all the object files, they are linked together along with additional object files from libraries to produce the complete executable.

Using the Makefile below, make starts at the first rule it finds, indicating that `myprog` depends on `myprog.o` and `math.o`. But before running the link command, **make** searches the rest of the Makefile and sees that `myprog.o` depends on `myprog.c` and that `math.o` depends on `math.c`. If either of the ".c" files is newer than the corresponding ".o" file, then those rules are executed before the link rule that uses them as sources.

```
// math.h

int     square(int c);
```

```
// math.c

int     square(int c)

{
    return c * c;
}
```

```
// myprog.c

#include <stdio.h>
#include <sysexits.h>
#include <stdlib.h>
#include "math.h"

int     main(int argc,char *argv[])

{
    int     c;

    for (c = -10; c < 10; ++c)
        printf("%d ^ 2 = %d\n", c, square(c));
    return EX_OK;
}
```

```
# Makefile

# Link myprog.o, math.o and standard library functions to create myprog.
myprog: myprog.o math.o
        cc -o myprog myprog.o math.o -lm

# Compile myprog.c to an object file called myprog.o
myprog.o: myprog.c
        cc -c myprog.c

# Compile math.c to an object file called math.o
math.o: math.c
        cc -c math.c
```

**Caution**
Do not explicitly set the compiler to **clang** or **gcc**. Doing so renders the Makefile non-portable. Every Unix system has a **cc** command which is usually the same as **clang** or **gcc**, depending on the specific operating system.
Addendum: The book contains some examples using **gcc** explicitly. At the time the book was written, it was a common practice to install **gcc** on commercial Unix systems and use it instead of the native **cc** compiler. This is no longer common.

Running **make** with this Makefile for the first time, we will see the following output:

```
shell-prompt: make
cc -c myprog.c
cc -c math.c
cc -o myprog myprog.o math.o
```

If we then edit `math.c`, then it will be newer than `math.o`. When we run **make** again, we will see the following:

```
shell-prompt: make
cc -c math.c
cc -o myprog myprog.o math.o
```

### 21.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Show the compiler commands needed to build an executable called `calc` from source files `calc.c` and `functions.c`.
2. What C compiler command should usually be used by default in a makefile? Why?
3. How does **make** know when a source file needs to be recompiled?

## 21.3 Make Variables

We can render this makefile more flexible and readable by using variables to eliminate redundant hard-coded commands and filenames, just as we do in scripts and programs.

To reference variables in the makefile, we enclose them in `${}`. We can also use `$()`, but this is easily confused with Bourne shell output capture, which converts the output of a process to a string expression that can be used by the shell.

```
BIN      = myprog
OBJS     = myprog.o math.o
CC       = cc
CFLAGS   = -Wall -O -g
LD       = ${CC}
LDFLAGS += -lm          # Add -lm to existing LDFLAGS

${BIN}: ${OBJS}
        ${LD} -o ${BIN} ${OBJS} ${LDFLAGS}

myprog.o: myprog.c Makefile
        ${CC} -c myprog.c

math.o: math.c
        ${CC} -c math.c
```

```
# Output capture in a Unix shell command, using the output of the
# "date" command as a string
printf "Today's date is %s.\n" $(date)
```

```
# We could also use the following, but since both make variables and
# shell output capture can be used in a Makefile, this could cause
# some confusion.

$(BIN): $(OBJS)
        $(LD) -o $(BIN) $(OBJS) -lm

...
```

If we want to allow the user to override a variable, we can use the conditional `?=` assignment operator instead of =. This tells **make** to perform this assignment only if the variable was not set in the **make** command or the environment.

The += operator appends text to a variable rather than overwriting it. This is often useful for adding important flags to commands.

```
# Values that the user cannot override are set using '='

BIN     = myprog
OBJS    = myprog.o math.o

# Set only if the user (or package manager) has not provided a value
# -Wall:    Issue all possible compiler warnings
# -g:       Compile with debug info to help locate crashes, etc.

CC      ?= cc
CFLAGS  ?= -Wall -O -g
LD      = ${CC}
LDFLAGS += -lm          # Add -lm to existing LDFLAGS

${BIN}: ${OBJS}
        ${LD} -o ${BIN} ${OBJS} ${LDFLAGS}

myprog.o: myprog.c Makefile
        ${CC} -c myprog.c

math.o: math.c
        ${CC} -c math.c
```

If Makefile contains the conditional assignments above, then make will use **cc -Wall -O -g** to compile the code unless `CC` or `CFLAGS` is defined in the **make** command or as an environment variable. Any of the following will override the defaults:

```
# Set CC and CFLAGS as make variables
shell-prompt: make CC=icc CFLAGS='-O -g'

# Set CC and CFLAGS as environment variables
shell-prompt: env CC=icc CFLAGS='-O -g' make

# Set CC and CFLAGS as environment variables (Bourne shell family)
export CC=icc
export CFLAGS='-O -g'
shell-prompt: make

# Set CC and CFLAGS as environment variables (C shell family)
setenv CC icc
setenv CFLAGS '-O -g'
shell-prompt: make
```

Some variables used in makefiles, such as CC, FC, LD, CFLAGS, FFLAGS, and LDFLAGS, are standardized. They have special meaning to make and to package managers. Hence, you should always use these variable names to indicate compilers, linkers, and compile/link flags. Most package managers will set these variables in the environment or **make** arguments, so the makefile should respect the values provided by using `?=` to only set defaults, rather than override what the package manager provides.

| **Variable** | **Meaning** | **Recommended default** |
|---|---|---|
| CC | C compiler | cc |
| CFLAGS | C compile flags | -Wall -O -g |
| CXX | C++ compiler | c++ |
| CXXFLAGS | C++ compile flags | -Wall -O -g |
| CPP | C Preprocessor (not C++ compiler!) | cpp |
| CPPFLAGS | C Preprocessor Flags | Not needed |
| FC | Fortran compiler | gfortran |
| FFLAGS | Fortran compile flags | -Wall -O -g |
| LD | Linker | ${CC}, ${FC}, or ${CXX} |
| LDFLAGS | Linker flags | -lm if using C math functions |
| PREFIX | Directory under which all files are installed | /usr/local |
| DESTDIR | Directory for temporary staged install | . |
| MKDIR | mkdir command, often provided by package manager as an absolute pathname to avoid aliases and locally installed alternatives | /bin/mkdir or just mkdir |
| INSTALL | install command used to install files | install |
| RM | rm command, mainly for "clean" target | /bin/rm or just rm |

Table 21.1: Standard Make Variables

### 21.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the purpose of **make** variables?
2. How do we set a variable in such a way that it can be overridden by **make** command-line arguments or environment variables?
3. What variables should be used to specify the C compiler? C compiler flags? The linker? Link flags?
4. Write a makefile, using standard **make** variables, that builds the executable "calc" from files "calc.c" and "functions.c".

## 21.4 Phony Targets

Some additional common targets are included in most Makefiles, such as "install" to install the binaries, libraries, and documentation, and "clean" to clean up files generated by the Makefile. These targets are not actual files and usually have no associated source, and are only executed if specified as a command line argument to **make**. Note that **make** builds the first target it finds in the Makefile by default, but if we provide the name of a target as a command line argument, it builds only that target instead.

To ensure that they behave properly even if a file exists with the same name as the target, they should be marked as phony by listing them as sources to the `.PHONY` target. Otherwise, if there happens to be a file called "install" or "clean" in the directory, its time stamp will determine whether the install or clean targets actually run.

```
# Values that the user cannot override

BIN     = myprog
OBJS    = myprog.o math.o

# Set only if the user (or package manager) has not provided a value
# -Wall:    Issue all possible compiler warnings
# -g:       Compile with debug info to help locate crashes, etc.

CC      ?= cc
CFLAGS  ?= -Wall -O -g
LD      = ${CC}
LDFLAGS += -lm

# Defaults for commands that may be provided by the package manager
MKDIR   ?= mkdir
INSTALL ?= install
RM      ?= rm

# Most build systems should perform a staged install (install to a
# temporary location indicated by ${DESTDIR}) rather than install
# directly to the final destination, such as /usr/local.  The package
# manager can then check the staged install under ${DESTDIR} for
# problems before copying to the final target.

# Defaults for paths that may be provided by the package manager.
# This will install under ./stage/usr/local unless the user or package
# manager provides a different location.
DESTDIR ?= ./stage
PREFIX  ?= /usr/local

${BIN}: ${OBJS}
        ${LD} -o ${BIN} ${OBJS} ${LDFLAGS}

myprog.o: myprog.c Makefile
        ${CC} -c ${CFLAGS} myprog.c

math.o: math.c
        ${CC} -c ${CFLAGS} math.c

.PHONY: install clean

install:
        ${MKDIR} -p ${DESTDIR}${PREFIX}/bin
        ${INSTALL} -c -m 0755 ${BIN} ${DESTDIR}${PREFIX}/bin

clean:
        ${RM} -f ${BIN} *.o
```

```
shell-prompt: make install
cc -c myprog.c
cc -c math.c
cc -o myprog myprog.o math.o
mkdir -p ./stage/usr/local/bin
install -c -m 0755 myprog ./stage/usr/local/bin

shell-prompt: make clean
rm -f myprog *.o
```

`PREFIX` is a standard **make** variable that indicates the common parent directory for all installed files. Executables (binaries) and scripts are typically installed in `${PREFIX}/bin`, libraries in `${PREFIX}/lib`, header files in `${PREFIX}/include/`

project-name, and data files in ${PREFIX}/share/project-name. Some projects might also install auxiliary programs or scripts not meant to be run directly by the user. These typically go under ${PREFIX}/libexec/project-name. Using a project-name subdirectory minimizes *collisions*, where multiple projects install files with the same name. For example, several FreeBSD ports install files called version.h, but there is no collision since they install them under their own directories:

```
/usr/local/include/alsa/version.h
/usr/local/include/assimp/version.h
/usr/local/include/bash/version.h
```

DESTDIR is another standard variable that was created to protect systems against install collisions. We do not use subdirectories under bin, and occasionally two projects will install a program or script by the same name. For example, the open source projects splay and mp3blaster both install a program called **splay**. Poorly designed Makefiles or other build systems may not check for collisions, and will simply clobber (overwrite) files previous installed by another project. Most package managers now require that the project install target use DESTDIR, so that the package manager can safely perform a staged install under a temporary directory, and then use its own safe methods to copy the installed files to their final destination, while watching for collisions. A staged installation also allows the package manager to check for proper permissions and verify that the installation conforms to filesystem hierarchy standards. Makefiles do not need to set DESTDIR, but they should prefix all install destinations with it.

**Note**
We do not need a '/' between DESTDIR and PREFIX, since PREFIX should be an absolute path name, which already begins with one.

```
    # Wrong
    ${MKDIR} -p ${DESTDIR}/${PREFIX}/bin
    ${INSTALL} -c -m 0755 ${BIN} ${DESTDIR}/${PREFIX}/bin

    # Right
    ${MKDIR} -p ${DESTDIR}${PREFIX}/bin
    ${INSTALL} -c -m 0755 ${BIN} ${DESTDIR}${PREFIX}/bin
```

### 21.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Which target in a makefile is checked first, if no target is specified in the **make** command?
2. How do we ensure that the install target runs when specified, even if there is a file called "install" in the directory?

## 21.5 Building Libraries

An almost identical makefile can be used to build a library instead of an executable. We need only replace the link command with a command that build a library archive. Also, none of the source files used to build a library should contain a main program.

```
LIB     = libmatrix.a
OBJS    = read.o write.o invert.o

CC      ?= cc
CFLAGS  ?= -Wall -O -g
AR      ?= ar               # Like tar, but for static libraries
RANLIB  ?= ranlib           # Generate an index for searching the library
```

```
# Let user or package manager control installation path
DESTDIR ?= ./stage
PREFIX  ?= /usr/local

# Build a static library
${LIB}: ${OBJS}
        ${AR} r ${LIB} ${OBJS}
        ${RANLIB} ${LIB}

read.o: read.c matlib.h
        ${CC} -c ${CFLAGS} read.c

write.o: write.c matlib.h
        ${CC} -c ${CFLAGS} write.c

invert.o: invert.c matlib.h
        ${CC} -c ${CFLAGS} invert.c

install:
        ${MKDIR} -p ${DESTDIR}${PREFIX}/lib
        ${INSTALL} -c -m 0755 ${LIB} ${DESTDIR}${PREFIX}/lib

clean:
        ${RM} -f ${LIB} *.o
```

This library can then be used by any program by linking with `-lmatrix`:

```
BIN     = myprog
OBJS    = myprog.o
LIBS    = libmatrix.a

CC      ?= cc
CFLAGS  ?= -Wall -O -g
LD      ?= ${CC}
LDFLAGS ?= -L${PREFIX}/lib -lmatrix -lm

DESTDIR ?= ./stage
PREFIX  ?= /usr/local

${BIN}: ${OBJS} ${LIBS}
        ${LD} -o ${BIN} ${OBJS} ${LDFLAGS}

myprog.o: myprog.c Makefile
        ${CC} -c ${CFLAGS} myprog.f90
```

The example above uses a *static library*. Static libraries in Unix have a ".a" extension. When a program is linked to a static library, the object code in the library is copied to the executable file.

Most projects build *shared libraries*, where only a reference to the library file is added to the executable. This saves disk space and allows multiple processes to use the same library code after it is loaded separately into memory. Building shared libraries properly is a bit more complex, since they need to be versioned in order to prevent compatibility problems. If a shared library is upgraded without rebuilding the programs that use it, those programs may not function properly. Shared libraries in most Unix systems have a filename extension of ".so.<version>", where "<version>" starts with "1" and is incremented with major upgrades. We will leave the creation of shared libraries for a more advanced course on software development.

### 21.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Write a makefile that creates a static library called "functions.a" from the source file "functions.c".

## 21.6 Mixing Tool Chains

Mixing object files compiled with different vendor's compilers is not generally advisable. Clang, GNU, Intel, and Sun compilers, for example, use slightly different formats for object files they output. Different major versions of the same compiler may also be incompatible. The one exception is that **clang/llvm** and **gcc** are designed to be compatible with each other. Some problems may occur, but they are generally resolvable.

Users are therefore advised to build all program modules with the same version of the same compiler. If pre-compiled libraries are being used (e.g. MPI, lapack, etc.) be sure to use the same compiler that the libraries were built with. A slightly newer compiler is usually OK as well, but even that introduces some doubt about the stability of the executable.

## 21.7 Mixing C, C++, and Fortran: A Brief Overview

It is possible to build programs using a mixture of C, C++, and Fortran code. There are some pitfalls, however. The details of mixing languages are beyond the scope of this manual, but a few examples are provided here to introduce the general idea.

### 21.7.1 Data formats

One major issue for scientific programmers is the structure of multidimensional arrays. C and C++ use row-major format (all elements of a given row are stored contiguously in memory), while Fortran uses column-major format (all elements of a given column are contiguous).

Hence, `matrix[row][col]` in C or C++ is represented as `matrix(col,row)` in Fortran.

This is just one of many differences in how data structures differ between C/C++, and Fortran. More information can be found on the web by searching for "mixing C and Fortran".

### 21.7.2 Libraries

Fortran compilers typically *decorate* or *mangle* symbol names with the program by appending an underscore character. Hence, a Fortran function called `invert()` would be called as `invert_()` from within a C or C++ program.

Calling C functions from within a Fortran or C++ program doesn't generally require and special effort. Hence, it is often easier to use the Fortran or C++ compiler for the link phase when building a mixed-language program that calls C functions.

This could, however, result in "undefined symbol" errors, since a Fortran compiler will not search the standard C libraries by default. If calling standard C library functions from a Fortran program, you may need to add `-lc` to the link command, so that the Fortran linker will search the standard C library, `libc.so`.

### 21.7.3 Examples

The example makefiles below show how to build programs with various mixtures of C, C++, and Fortran. If you are not familiar with makefiles, you may want to read Chapter 21 first.

```
BIN     = program
OBJS    = main.o functions.o

# Use the same tool chain for all builds to avoid issues
CC      = gcc
CFLAGS  = -Wall -O -g
FC      = gfortran
FFLAGS  = ${CFLAGS}
```

```
# Use Fortran compiler to link to avoid name mangling issues
LD      = ${FC}
LDFLAGS = -lc

# Link
program:    main.o functions.o
        ${LD} -o ${BIN} ${OBJS} ${LDFLAGS}

main.o: main.f90
        ${FC} -c ${FFLAGS} main.f90

functions.o:    functions.c
        ${CC} -c ${CFLAGS} functions.c
```

If calling C++ library functions from Fortran built with gfortran, link with -lstdc++ (the GNU C++ library).

```
BIN     = program
OBJS    = main.o functions.o

# Use the same tool chain for all builds to avoid issues
CXX         = g++
CXXFLAGS    = -Wall -O -g
FC          = gfortran
FFLAGS      = ${CXXFLAGS}

# Use Fortran compiler to link to avoid name mangling issues
LD      = ${FC}
LDFLAGS = -lstdc++

# Link
program:    main.o functions.o
        ${LD} -o ${BIN} ${OBJS} ${LDFLAGS}

main.o: main.f90
        ${FC} -c ${FFLAGS} main.f90

functions.o:    functions.cc
        ${CXX} -c ${CXXFLAGS} functions.cc
```

If calling Fortran library functions from C or C++ with the GNU compiler collection, add -lgfortran.

```
BIN     = program
OBJS    = main.o functions.o

# Use the same tool chain for all builds to avoid issues
CC      = gcc
CFLAGS  = -Wall -O -g
FC      = gfortran
FFLAGS  = ${CFLAGS}

# Use C compiler to link, since the main program is in C
LD      = ${CC}
LDFLAGS = -lgfortran

# Link
program:    main.o functions.o
        ${LD} -o ${BIN} ${OBJS} ${LDFLAGS}

main.o: main.c
        ${CC} -c ${CFLAGS} main.c

functions.o:    functions.f90
```

```
        ${FC} -c ${FFLAGS} functions.f90
```

If calling C++ library functions from Fortran built with gfortran, link with -lstdc++ (the GNU C++ library).

```
BIN     = program
OBJS    = main.o functions.o

# Use the same tool chain for all builds to avoid issues
CXX         = g++
CXXFLAGS    = -Wall -O -g
FC          = gfortran
FFLAGS      = ${CXXFLAGS}

# Use Fortran compiler to link to avoid name mangling issues
LD      = ${FC}
LDFLAGS = -lstdc++

# Link
program:    main.o functions.o
        ${FC} -o ${BIN} ${OBJS} ${LDFLAGS}

main.o: main.f90
        ${FC} -c ${FFLAGS} main.f90

functions.o:    functions.cc
        ${CXX} -c ${CXXFLAGS} functions.cc
```

### 21.7.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. Can we mix C, C++, and Fortran source code in the same makefile? Explain.

## 21.8 Makefile Generators

A common and peculiar theme in human nature is the impulse to *add* something to solve every problem, rather than *eliminate* the root cause of the problem. Many people apparently find this easier than thinking through the problem to understand what's really going on. The result is increasing and needless complexity. In the software world, this translates to added layers of software that compensate for, rather than fix, the original problems.

There are lots of tools that attempt to automatically generate makefiles specific to the user's environment, or even automate the download and installation of dependent projects (a task that should be left to package managers, which do this much better). It really isn't necessary to add this extra complexity, and is generally not a good idea in the modern world of package managers. The results of trying to be overly clever are usually disastrous. K.I.S.S. ( Keep It Simple, Stupid )

cleverness * wisdom = constant

Most such tools, including the most popular, **GNU configure** and **cmake**, attempt to locate libraries and other dependencies installed via completely unknown and often incorrect methods. They may have been installed using multiple different package managers, or willy-nilly via caveman installs performed by unqualified users. Worse yet, there may be multiple installations of the same library waiting to be discovered, and the build system may use the first one it finds, rather than the correct one. Attempts to compensate for the thousands of variables that might be encountered in the chaotic environments created by thousands of different users generally results in an absurdly complex build system that works some of the time and is a nightmare to debug when it doesn't.

In addition to the common problems with build scripts, generator tools such as GNU configure and cmake also have bugs of their own. When you or a user of your software runs into one, it can be a real challenge to solve, and many people will just give up.

Attempts to build and install software this way are partly responsible for the false notion that compiled languages are difficult to use. In fact, the languages are not the problem: the problem is poorly designed and overly complicated build systems that don't work outside the developers' specific environment. When such build systems fail, they are a nightmare to debug.

My advice is to create a simple makefile that looks for dependencies *only* where the user (or package manager) tells it to look. This is easy to do using standard make variables such as CFLAGS and LDFLAGS, which virtually all package managers already provide via **make** command-line arguments or environment variables. This way, we can be sure that the build is using only the dependencies that we want it to use, which were hopefully installed by a trusted procedure, such as the same package manager building the package at hand.

Ideally, the makefile should build and install only this project and leave the installation of dependencies to the package manager. If a program and all of its dependencies are installed independently by the same package manager, then all of the individual makefiles they use can be simple and reliable, and very few problems will occur. This means users won't waste their time struggling to haphazardly install your software, and you won't waste your time trying to help them with self-inflicted problems.

### 21.8.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2 before doing the practice problems below.

1. What is the intended purpose of makefile generators? Are they necessary? Do they always work? Why or why not? What might be a more reliable approach to building and installing software?
2. How do overly complicated build systems contribute to the notion that compiled languages are difficult to use?

# Chapter 22

# Memory Addresses and Pointers

Virtually every value used in a program is stored in memory and has a memory address.

We sometimes need to use the memory address of a variable instead of the data it contains. For example, the Fortran read subroutine and the C scanf() function both need to know the memory address of a variable in order to place input data in it.

Since Fortran passes arguments by reference, Fortran subprograms naturally know the address of all the arguments passed to them.

Since C passes arguments by value, we need to explicitly pass the address of a variable to any function that needs to know it.

```
if ( scanf("%d %d", &rows, &cols) != 2 )
```

So how to functions like scanf() make use of the addresses passed to them?

They use *pointer variables*. A pointer variable is a variable that contains a memory address rather than data.

We can assign a pointer variable the address of any other variable of the appropriate data type. We can then access the data in the variable pointed to using what is called *indirection*.

```
int     rows = 2,
  cols = 3;
int     *ptr;       // Holds the address of an int variable

ptr = &rows;
printf("%d\n", *ptr);  // Indirect access to the value in rows, prints 2

ptr = &cols;
printf("%d\n", *ptr);  // Indirect access to the value in cols, prints 3
```

Functions that need to modify a variable in the caller must know the address of that variable. We've already seen that we can pass the address from the caller using the & operator:

```
scanf("%d %d", &rows, &cols);
```

To receive such an address in the function, we simply define the argument as a pointer. Consider the simple task of swapping the values of two variables:

```
temp = first;
first = second;
second = temp;
```

We might try to implement this as a function for convenience:

```
void    swap_ints(int first, int second)

{
    int     temp;

    temp = first;
    first = second;
    second = temp;
}

swap(a, b);
```

Knowing that C passes arguments by value, however, makes it apparent that this function is useless. The argument variables first and second are at different memory addresses than a and b, so swapping them has no effect on a and b.

In order to create a working swap function, we need to define the argument variables as pointers and pass the addresses of the arguments:

```
void    swap_ints(int *first, int *second)

{
    int     temp;

    temp = *first;
    *first = *second;
    *second = temp;
}

swap(&a, &b);
```

Pointer variables have many additional uses, but their primary purpose in scientific programming is for passing addresses of data to functions.

# Chapter 23

# Arrays

## 23.1 Motivation

Write a program that reads in a list of 1000 numbers and prints them in the order they were entered. This is easy:

```
int     c;
double  number;

for (c = 0; c < 1000; ++c)
{
    scanf("%lf", &number);
    printf("%f\n", number);
}
```

```
integer :: i
double precision :: number

do i = 1, 1000
    read *, number
    print *, number
enddo
```

Now, write a program that reads a list of 1000 numbers from the standard input and prints them in reverse order.

In order to achieve this, the entire list must be stored in a way that it can be traversed backwards.

There are basically two places to store a list of values:

- In a file
- In RAM

Whether it's better to store the list in a file or in RAM depends on how large the list is and how fast we need to be able to read the list backwards. Disks are much larger than RAM, so they will accommodate larger lists. RAM is many times faster than disk, however, and will therefore lead to a faster program.

## 23.2 Design vs. Implementation

As always, we need to separate design and implementation. Deciding whether to store our list in a file or in RAM is an implementation detail. The need to store it *somewhere* and read it backwards is a design detail.

Do not let thoughts about RAM and disk creep into your head while *designing* the solution to this problem. The design demands only that the solution *remember* every value entered. This is the important difference between the solutions for printing the list in forward or reverse order. To print them in forward order, the solution can read one value, print it, and forget it. To print them in reverse order, they must all be remembered until the last one is entered.

Conceptually, a one-dimensional list of data can be represented as a *vector*:

[ $a_0$, $a_{1,}$, ... $a_n$ ]

Vectors can represent any one-dimensional collection of data, such as a simple list, coefficients of a point in space, or a polynomial (each $a_i$ represents a coefficient).

$2.5x^2$ - 3.9x + 7.2 = [ 2.5 -3.9 7.2 ]

The variables we've worked with until now, which hold a single value, are *scalar*, or *dimensionless*.

## 23.3 Array Basics

To implement the solution to printing 1000 numbers in reverse order, we could define 1000 variables, and use 1000 read and print statements:

```
double  num1, num2, ... num1000

scanf("%lf", &num1);
scanf("%lf", &num2);
...
scanf("%lf", &num1000);

printf("%f\n", num1000);
...
printf("%f\n", num2);
printf("%f\n", num1);
```

```
double precision :: num1, num2, ... num1000

read *, num1
read *, num2
...
read *, num1000

print *, num1000
...
print *, num2
print *, num1
```

We now have a program of 2000 lines + the overhead code framing out the program, defining the variables, etc. This is clearly more effort than we want to put into a program, and the program is not at all flexible. Where would we be if we needed to do the same for a billion numbers?

An array is a variable that holds multiple values of the same type. Each value in the array is distinguished from the others using an integer *index*, also known as a *subscript*. The term subscript comes from mathematics, where we might specify one *element* from of an array of values such as $K_0$, $K_1$, ... $K_n$.

In C, we make a variable into an array by providing the number of elements it contains in square brackets following the name. We then select an element by specifying the subscript in square brackets after the name. C subscripts always begin at 0 and end at the array size minus one.

```
#define LIST_SIZE   1000

double list[LIST_SIZE];
```

```
scanf("%lf", &list[0]);
scanf("%lf", &list[1]);
...
scanf("%lf", &list[999]);

printf("%f\n", list[999]);
...
printf("%f\n", list[1]);
printf("%f\n", list[0]);
```

In Fortran, we make a variable into an array by providing the number of elements it contains in parentheses following the name. We then select an element by specifying the subscript in parentheses after the name. By default, Fortran subscripts begin at 1, but we can control this.

```
module constants
    integer, parameter :: LIST_SIZE=1000
end module

double precision :: list(LIST_SIZE)

read *, list(1)
read *, list(2)
...
read *, list(1000)

print *, list(1000)
...
print *, list(2)
print *, list(1)
```

An array makes it much easier to allocate the space for 1000 double precision values, but we still have 2000 statements to read the list and print it backwards. Can we do better?

The only restriction on subscripts is that the must be integers. They may be constants, variables, or complicated expressions. As long as the value of the subscript is within the range of subscripts for the array, there is no problem.

Most commonly, subscripts are a simple variable which is controlled by a loop that traverses all valid subscripts for the array:

Variables used to subscript an array must have enough range. A Fortran integer(1) or C char variable have a maximum value of +127, so they cannot be used as subscripts for an array of 1000 elements. A Fortran integer (integer(4)) variable has a maximum value of $2^{31}$-1 (a little over 2 billion), which is enough for most arrays. It is possible on modern computers to have arrays with more than 2 billion elements, so we may sometimes need to use integer(8).

The C header files define an unsigned integer type called `size_t` which has the same number of bits as a memory address on the underlying hardware. Hence, it is guaranteed to be able to handle an array of any size. This data type should be used for virtually all array subscripts in C.

```
#define LIST_SIZE   1000

double  list[LIST_SIZE];
size_t  c;

for (c = 0; c < LIST_SIZE; ++c)
    scanf("%lf", &list[c]);

for (c = LIST_SIZE-1; c >= 0; --c)
    printf("%f", list[c]);
```

```
module constants
    integer, parameter :: LIST_SIZE=1000
end module
```

```
double precision :: list(LIST_SIZE)
integer :: index

do index = 1, LIST_SIZE
    read *, list(index)
enddo

do index = LIST_SIZE, 1, -1
    print *, list(index)
enddo
```

In Fortran, if we want to define multiple arrays of the same size, we can use the *dimension* modifier instead of specifying the dimension for every array variable.

```
double precision, dimension(LIST_SIZE) :: list1, list2, list3
```

Fortran also allows the programmer to choose any starting and ending subscripts desired. The default starting subscript is 1, so the following is equivalent to the examples above:

```
double precision :: list(1:LIST_SIZE)
```

For some arrays, it may not make sense to use a starting subscript of one. For example, if an array contains the probability of a driver in the U.S. having an accident based on their age, then subscripts less than 15 would be useless, since people under are not allowed to drive.

We should also specify the array dimensions using named constants to make the program more readable and easier to modify.

```
module constants
    integer, parameter :: MIN_AGE=15, MAX_AGE=120
end module

program insurance
    double precision :: accident_probability(MIN_AGE : MAX_AGE)
    integer :: age

    ...

    do age = MIN_AGE, MAX_AGE
        print *, accident_probability(age)
    enddo

    ...
end program
```

```
#include <stdio.h>
#include <sysexits.h>

#define MIN_AGE     15
#define MAX_AGE     120

int     main()
{
    double accident_probability[MAX_AGE - MIN_AGE + 1];
    size_t age;

    for (age = MIN_AGE; age <= MAX_AGE; ++age)
        printf("%f\n", accident_probability[age - MIN_AGE]);

    return EX_OK;
}
```

## 23.4 Dynamic Memory Allocation

In the old days of programming, arrays were always defined with a fixed, or *static* size. However, the amount of input to a program is rarely fixed. This means that static arrays must be defined to accommodate the largest possible inputs. For example, a program made to process lists of up to 1,000,000 elements must use an array of 1,000,000 elements, even when processing a list of 3 elements. This is a colossal waste of memory resources.

All modern languages allow the size of arrays to be determined at run-time, so they only use as much memory as needed. For this reason, static arrays should not be used anymore, unless it is an absolute certainty that the size of the array will not vary much.

More often than not, we don't know how big our list is until run-time.

In C, an array name is actually a pointer. The array name always represents the address of the first element in the array. The only difference between an array name and a pointer variable in C is that we cannot change what the array name points to. I.e., an array name is a pointer constant rather than a pointer variable.

Pointer variables and array names are completely interchangeable, with one exception: An array name by itself cannot be on the left side of an assignment statement. Most importantly, we can use subscripts with pointer variables just like we do with array names.

Hence, to create a dynamically allocated array in C, we begin by defining a pointer variable instead of an array variable.

We then use the malloc() library function to allocate memory, which is prototyped in stdlib.h. The malloc() function returns the address of the allocated memory, or the constant NULL if the allocation failed.

The return type of malloc() is void *, so we should cast it to the pointer type of the array to avoid a compiler warning.

Lastly, note that malloc() takes the number of *bytes* to allocate as an argument, not the number of array elements. With malloc(), we generally use the C sizeof() operator, which returns the size of a variable or data type.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  *temperatures;
    size_t  num_temps,
            c;

    printf("How many temperatures are there? ");
    scanf("%zu", &num_temps);   // %zu for size_t
    printf("%zu\n", num_temps);
    temperatures = (double *)malloc(num_temps * sizeof(*temperatures));
    if ( temperatures == NULL )
    {
        fprintf(stderr, "Cannot allocate memory for temperatures.\n");
        exit(EX_OSERR);
    }

    puts("Enter the temperatures separated by whitespace:");
    for (c = 0; c < num_temps; ++c)
        scanf("%lf", &temperatures[c]);

    // Careful here: c is unsigned, so it is always >= 0
    while (c-- > 0)
        printf("%f\n", temperatures[c]);

    // Always free memory as soon as possible after it's used
    free(temperatures);
    return EX_OK;
}
```

In Fortran, we can indicate that the size of an array is to be determined later by adding the `allocatable` modifier and placing only a ':' in the dimension. We then use the **allocate** intrinsic subroutine to allocate the array at run-time.

```
program allocate
    use iso_fortran_env
    implicit none

    double precision, allocatable :: temperatures(:)
    integer :: num_temps, allocate_status, i

    print *, 'How many temperatures are there?'
    read *, num_temps
    allocate(temperatures(1:num_temps), stat=allocate_status)
    if ( allocate_status /= 0 ) then
        write(ERROR_UNIT, *) 'Cannot allocate memory for temperatures.', &
                            'Error code = ', allocate_status
        stop
    endif

    print *, 'Enter the temperatures one per line:'
    do i = 1, num_temps
        read *, temperatures(i)
    enddo

    do i = num_temps, 1, -1
        print *, temperatures(i)
    enddo
end program
```

We must provide an integer variable along with "stat=" to receive the status of the allocation attempt. If the memory for the array is allocated successfully, the status variable will be set to 0. If the allocation fails (there is not enough memory available to satisfy the request), it will be set to a non-zero value. Programs can be very sophisticated with the status codes returned. They may decide to request a smaller block, or deallocate something else and try again.

The very least we should do is stop the program. Failure to check the status of a memory allocation can cause incorrect output, which could be catastrophic in some cases.

Once the allocatable array is allocated, it can be used like any other array.

When the array is no longer needed, it should be *immediately* deallocated, to free up the memory for other uses.

```
deallocate(temperatures)
```

Garbage collection

## 23.5 Array Constants

Fortran includes the concept of an *array constant*. This can reduce the size of a program if an array must be initialized to a set of constant values.

```
list = (/ 2, 4, 10 /)
```

Same as

```
list(1) = 2
list(2) = 4
list(3) = 10
```

C does not have the same flexible array constant construct, but it does allow array initializers. In addition, when an initializer is used on an array, we can omit the array size, since it can be inferred from the initializer.

```
double  list[] = { 2, 4, 10 };
```

## 23.6 Static Array Initialization

### 23.6.1 The Fortran DATA Statement

In C, if we initialize a `static` array, the initialization occurs before the program begins running.

```
static double   list[] = { 2, 4, 10 };
```

Without the static modifier, the array would be initialized at run time, when the block containing the variable definition is entered. With the static modifier, the initialization takes place at compile time, so the numbers are already in the array as soon as the process begins executing.

Another way to look at it is that variables are always initialized when they are *instantiated*, i.e. when their memory is allocated. Static variables are instantiated at compiled time and auto variables at run time, when they are needed.

Another way to initialize a Fortran array is using a `data` statement.

```
integer :: list(3)

data list / 2, 4, 10 /
```

This differs from assigning the array an array constant in that the data statement is processed at compile-time, whereas the assignment is done at run-time. When the program begins execution, the values from a data statement will already be in the array. Since the data statement is not performed at run-time, it actually makes no difference where it is placed within the subprogram. It is processed and removed by the compiler, not translated to machine code.

You may need to assign values to an array at run-time, or even assign the array several different sets of values at different times. In this case, an assignment statement is necessary.

However, if the contents of an array will not change during program execution, then using a data statement will save execution time, since the data was loaded into the array at the same time the program itself was loaded into memory.

### 23.6.2 Look-up Tables

A look-up table is a list of precomputed values. If a value is expensive to compute (i.e. requires a loop) and frequently needed, it may be quicker to store precomputed values in an array and look them up instead. This only involves indexing the array (computing the address of an element) which is trivial (a few machine instructions).

Ideal candidates for look-up tables are mathematical functions with a small domain that require a loop to compute. Since we are storing precomputed results for all cases, a look-up table will generally require more memory than the code to compute results on-the-fly.

The factorial is one such function. Only a small number of factorials are within the range of typical integer or floating point types, and they must be computed iteratively.

To achieve a performance gain using a look-up table, we must use a Fortran data statement or a C static array. Assigning an array constant at run-time or initializing an automatic array is costly, since it is itself an implied loop.

```
uint64_t   fastfact(unsigned int n)

{
    static uint64_t table[] = { 1ul, 1ul, 2ul, 6ul, 24ul, 120ul, 720ul,
        5040ul, 40320ul, 362880ul, 3628800ul, 39916800ul, 479001600ul,
        6227020800ul, 87178291200ul, 1307674368000ul, 20922789888000ul,
        355687428096000ul, 6402373705728000ul, 121645100408832000ul };

    return table[n];
}
```

```
function fastfact(n)

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Function type
    integer(8) :: fastfact

    ! Argument variables
    integer, intent(in) :: n

    ! Local variables
    integer(8) :: table(0:19)

    fastfact = table(n)

    data table / 1_8, 1_8, 2_8, 6_8, 24_8, 120_8, 720_8, 5040_8, 40320_8, &
        362880_8, 3628800_8, 39916800_8, 479001600_8, 6227020800_8, &
        87178291200_8, 1307674368000_8, 20922789888000_8, &
        355687428096000_8, 6402373705728000_8, 121645100408832000_8 /
end function
```

To generate the factorial data for the lookup table requires a program. Integers are limited to about 19!. Double precision floating point can represent up to 170!, but there's another problem: Even 64-bit floating point systems are limited to about 16 significant figures. As soon as the factorial value reaches 17 significant figures, it gets rounded. All later factorials are computed from this rounded value, and so the round-off error grows rapidly from this point on. 170! has 305 significant digits, but at most 16 of them will be accurate if the value is computed using double precision. Output from a C program using double precision follows. Note that 171! is reported as infinity, since 64-bit IEEE format cannot represent it.

```
    169! =
    4.2690680090047055956e+304

    170! =
    7.2574156153080040246e+306

    171! =
    inf
```

What we need to compute large factorials precisely is an arbitrary-precision integer package. Such a package works by concatenating integer values end-to-end, and processing these large integers in chunks (e.g. 64 bits at a time on a 64 bit computer). There are libraries available for C, C++, Java, etc. However, the Unix **bc** is quite convenient for this purpose, and includes a C-like scripting language. Below is a bc script that prints factorials. You can run it by saving it to a file (e.g. fact.bc) and running `bc fact.bc` at the Unix prompt.

```
f = 1
for (c = 1; c <= 170; ++c) {
    f
    f = f * c
}
quit
```

```
    169! =
    4269068009004705274939251888899566538068818636056736103849163411117\
    9775549421800928543239701427161526538182301399050122215682485679075\
    0177960523574894559464847084134121076211998036035274015123788150487\
    8975040568419670360154453585262827477179746400268937258948624384000\
    0000000000000000000000000000000000000000

    170! =
    72574156153079989673967282111292631147169916812964513765435777989005\
    61843401706157852350749242617459511490991237838520776666022565442753\
    02532890077320751090240043028005829560396661259965825710439855829425\
    75689663134396122625710949468067112055688804571933402126614528000000\
    00000000000000000000000000000000
```

A modified version of the bc script can generate output that can be copied and pasted directly into a program. Note that this version requires GNU bc, which is not standard on all systems.

```
fact = 1
for (c = 1; c <= 170; ++c) {
    print fact, ",\n";
    fact = fact * c;
}
quit
```

## 23.7 Arrays as Arguments

Arrays can be passed as arguments to functions and subroutines. When doing so, we need to also send the dimensions of the array. We will need the size in order to properly set up loops inside the subprogram that access the array.

In C, a one-dimensional array argument need not be given a size. This allows a function to work with arrays of any size, so long as it is told the size of each array when it is called.

```
#define     MAX_TEMPS   1000

int     main()

{
    double  temps[MAX_TEMPS];

    print_list(temp, MAX_TEMPS);
    return EX_OK;
}


void    print_list(double list[], size_t list_size)

{
}
```

Note that we we pass an array to a function, we are not copying the array to the function's argument variable. Passing an array itself by value would mean duplicating the entire contents of the array, which would be very inefficient for large arrays.

Remember that an array name in C is a pointer constant. It represents the address of the array, whereas a scalar variable name represents the value contained in it. When we use an array name as an argument to a function, we are passing the address of the array, much like we did explicitly with our swap() function example:

```
int     a, b;

swap(&a, &b);
```

In fact, we could declare the function argument variable as a pointer rather than an array. The following would be essentially identical to the example above:

```
void    print_list(double * const list, size_t list_size)

{
}
```

**Note** The trick to understanding the use of `const` with pointers is to read it backwards. The above says "list is a constant pointer to a double".

Since we are passing the address of an array, the array contents are not protected from side effects. When we pass an array to a function, the function may be able to modify it.

We can protect against this by declaring the argument variable as an array of constants:

```
void    print_list(const double list[], size_t list_size)

{
}
```

**Note** Again, reading this backwards, we see that "list is a pointer to a double constant", meaning that the data it points to is constant. Since list is declared as an array, the address is also constant. We can neither make change where list points, nor change the content of what it points to.

If we inadvertently try to modify the contents of list[] in the print_list() function, we will now get a compiler error.

Fortran does require a size for an array argument. However, since arguments are never without a value, the array size argument can be used to declare the array in Fortran. Array dimensions can be variables, as long as they have a known value. We must also declare the size argument before the array, so that the compiler has seen it before it encounters the array declaration.

```
program array_arguments

    double precision, allocatable :: temps(:)
    integer :: num_temps

    ...

    call print_list(temps, num_temps)

    ...

end program

subroutine print_list(list, list_size)

    ! Define list_size before list
    integer, intent(in) :: list_size
    double precision, intent(in) :: list(1:list_size)

    integer :: i

    do i = 1, list_size
        print *, list(i)
    enddo

end subroutine
```

## 23.8 Allocate and Subprograms

Unlike some other languages, an array allocated in a Fortran subprogram is automatically deallocated when the subprogram returns.

To some extent, this has the advantage of preventing *memory leaks*, where a programmer allocates more and more memory over time and forgets to deallocate all of it, resulting in a continual reduction in available memory. Automatically deallocating arrays does not completely eliminate memory leaks however, and some might argue that it leads to less disciplined programmers by allowing them to be less vigilant.

Automatically deallocating arrays when a subprogram returns is sometimes a disadvantage as well. There are situations where we *want* a subprogram to allocate memory which can later be used by the calling subprogram, or others. In Fortran, however, allocated memory is only accessible to the subprogram that allocated it, and other subprograms called by it after the allocation and before it returns. Consider the following read_list() implementation:

```
! Main program body
program cant_do_this
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable defintions
    integer :: num_temps
    real(8), allocatable :: temps(:)

    ! Allocate an array for a list of temps and fill it from input
    ! Get back the list and the list size from read_list()
    call read_list(temps, num_temps)

    ! Print the list returned by read_list

end program

subroutine read_list(list, list_size)
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Dummy variables
    ! Define list_size first, since it is used to define list
    integer, intent(out) :: list_size
    real(8), intent(out), allocatable :: list(:)

    ! Local variables
    integer :: i, alloc_status

    read *, list_size
    allocate(list(1:list_size), stat=alloc_status)
    if ( alloc_status /= 0 ) then
        print *, 'Allocate failed.  status = ', alloc_status
        stop
    endif

    do i = 1, list_size
        read *, list(i)
    enddo
end subroutine
```

The code above will compile, but will not work, because the array allocated inside the read_list() subroutine is deallocated when the subroutine returns. Hence, the array no longer exists after coming back to the main program from read_list().

The code below shows how this task must be accomplished in Fortran. The array must be allocated before calling read_list() and passed to read_list() to receive the input. The array can exist in the main program, in read_list(), and in other subprograms called by the main program.

```
! Main program body
program this_one_works
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable defintions
    integer :: num_temps, alloc_status
    real(8), allocatable :: temps(:)

    read *, num_temps
    allocate(temps(1:num_temps), stat=alloc_status)
    if ( alloc_status /= 0 ) then
        print *, 'Allocate failed.  status = ', alloc_status
        stop
    endif

    ! Allocate an array for a list of temps and fill it from input
    ! Get back the list and the list size from read_list()
    call read_list(temps, num_temps)

    ! Print the list returned by read_list

end program

subroutine read_list(list, list_size)
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Dummy variables
    ! Define list_size first, since it is used to define list
    integer, intent(in) :: list_size
    real(8), intent(out) :: list(1:list_size)

    ! Local variables
    integer :: i

    do i = 1, list_size
        read *, list(i)
    enddo
end subroutine
```

In C, memory allocated by malloc() remains allocated until it is released by free(). This is part of the C philosophy of "trust the programmer". This philosophy often leads to programmers shooting themselves in the foot, but also allows programmers to easily do what they need to.

## 23.9 Sorting

One of the most common uses for arrays is sorting lists. Like reversing a list, sorting is inefficient to perform on disk, and easy and fast if done in memory. Of course, if the list is very large, we may not have enough RAM to hold it. In these cases, however, we can sort *parts* of the list into memory, and merge them later.

Sorting is a very well-studied problem, and many sorting algorithms have been developed. The most efficient algorithms are somewhat difficult to understand, so we will focus on the simpler ones in order to demonstrate the use of arrays more easily.

One of the simplest and most intuitive sorting algorithms is the *selection sort*. The design of the selection sort is as follows:

1. Read the list
2. Sort the list
   (a) Find the smallest (or largest) element in the list.

   (b) Swap it with the first element.
   (c) Repeat the previous two steps starting at the next element.
   (d) Repeat the previous three steps until the entire list is sorted.

3. Print the sorted list

To help illustrate a clean implementation process, here is a framed-out first step with stubs for reading and printing the list:

```
/***************************************************************************
 *  Usage:  sort input-file output-file
 *
 *  Sort a list of real numbers in input-file, placing the results
 *  in output-file.  Both input and output files contain one
 *  number per line.
 *
 *  Arguments:
 *  input-file:     file to be sorted
 *  output-file:    file to receive the sorted list
 *
 *  History:
 *  Date        Name        Modification
 *  2017-08-24  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

double  *read_list(size_t *size);
void    print_list(const double list[], size_t list_size);
void    sort_list(double list[], size_t list_size);

int     main(int argc,char *argv[])

{
    double  *list;
    size_t  list_size;

    list = read_list(&list_size);
    print_list(list, list_size);
    free(list);
    return EX_OK;
}


/*
 *  Input list size, allocate array, and read in list.
 */

double  *read_list(size_t *size_ptr)

{
    double  *list;

    *size_ptr = 10;
    list = (double *)malloc(*size_ptr * sizeof(*list));
    return list;
}


void    print_list(const double list[], size_t list_size)
```

```
{
    printf("%zu\n", list_size);
}
```

```
shell-prompt: ./selsort
10
```

After testing the stubs, we fill them out and test the completed read and print functions:

```
/***************************************************************************
 *  Usage:  sort input-file output-file
 *
 *  Sort a list of real numbers in input-file, placing the results
 *  in output-file.  Both input and output files contain one
 *  number per line.
 *
 *  Arguments:
 *  input-file:     file to be sorted
 *  output-file:    file to receive the sorted list
 *
 *  History:
 *  Date        Name        Modification
 *  2017-08-24  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

double  *read_list(size_t *size);
void    print_list(const double list[], size_t list_size);
void    sort_list(double list[], size_t list_size);

int     main(int argc,char *argv[])

{
    double  *list;
    size_t  list_size;

    list = read_list(&list_size);
    print_list(list, list_size);
    free(list);
    return EX_OK;
}


/*
 *  Input list size, allocate array, and read in list.
 */

double  *read_list(size_t *size_ptr)

{
    size_t  c;
    double  *list;

    scanf("%zu", size_ptr);     // No & here, since size_ptr is a pointer
    list = (double *)malloc(*size_ptr * sizeof(*list));
    for (c = 0; c < *size_ptr; ++c)
        scanf("%lf", &list[c]);
    return list;
}


void    print_list(const double list[], size_t list_size)

{
    size_t  c;

    for (c = 0; c < list_size; ++c)
        printf("%f\n", list[c]);
}
```

```
shell-prompt: ./selsort
3
6.0
2.1
9.4
6.000000
2.100000
9.400000
```

Finally, we implement the sort function. Note that while we work on the sort function, we need not worry about anything else, since we know that the rest of the program is already tested.

```
/***************************************************************************
 *  Usage:  sort input-file output-file
 *
 *  Sort a list of real numbers from stdin, printing the results
 *  to stdout.  Both input and output contain one number per line.
 *
 *  History:
 *  Date        Name        Modification
 *  2017-08-24  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

double  *read_list(size_t *size);
void    print_list(const double list[], size_t list_size);
void    sort_list(double list[], size_t list_size);

int     main(int argc,char *argv[])

{
    double  *list;
    size_t  list_size;

    list = read_list(&list_size);
    sort_list(list, list_size);
    print_list(list, list_size);
    free(list);
    return EX_OK;
}


/*
 *  Input list size, allocate array, and read in list.
 */

double  *read_list(size_t *size_ptr)
```

```
{
    size_t  c;
    double  *list;

    scanf("%zu", size_ptr);    // No & here, since size_ptr is a pointer
    list = (double *)malloc(*size_ptr * sizeof(*list));
    for (c = 0; c < *size_ptr; ++c)
        scanf("%lf", &list[c]);
    return list;
}


void    print_list(const double list[], size_t list_size)

{
    size_t  c;

    for (c = 0; c < list_size; ++c)
        printf("%f\n", list[c]);
}


/*
 *  Sort list using selection sort algorithm.
 */

void    sort_list(double list[], size_t list_size)

{
    size_t  start,
            low,
            c;
    double  temp;

    for (start = 0; start < list_size - 1; ++start)
    {
        /* Find lowest element */
        low = start;
        for (c = start + 1; c < list_size; ++c)
            if ( list[c] < list[low] )
                low = c;

        /* Swap first and lowest */
        temp = list[start];
        list[start] = list[low];
        list[low] = temp;
    }
}
```

```
shell-prompt: ./selsort
3
6.0
2.1
9.4
2.100000
6.000000
9.400000
```

And the Fortran implemtation:

```
!-----------------------------------------------------------------------
!   Description:
!       Usage:  sort input-file output-file
!
!       Sort a list of real numbers in input-file, placing the results
!       in output-file.  Both input and output files contain one
!       number per line.
!
!   Arguments:
!       input-file:     file to be sorted
!       output-file:    file to receive the sorted list
!
!   Modification history:
!   Date        Name        Modification
!   Apr 2010    J Bacon     Start.
!-----------------------------------------------------------------------

module constants
    ! Global Constants
    real(8), parameter :: &
        PI = 3.1415926535897932d0, &
        E = 2.7182818284590452d0, &
        TOLERANCE = 0.00000000001d0
end module constants

program selection_sort
    ! Import stuff from constants module
    use constants
    use ISO_FORTRAN_ENV

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Local variables
    integer :: list_size, allocate_status
    real(8), allocatable :: list(:)

    ! Get size of list
    read *, list_size

    ! Allocate array for list
    allocate(list(1:list_size), stat=allocate_status)
    if ( allocate_status /= 0 ) then
        print *, 'Error: Could not allocate array of size ', list_size
        stop
    endif

    ! Read list
    call read_list(list, list_size)

    ! Sort list
    call sort_list(list, list_size)

    ! Output list
    call print_list(list, list_size)
end program


!-----------------------------------------------------------------------
!   Description:
!       Read a list of real numbers from the standard input to
!       the array list.  The input file contains one number per line.
```

```
!
!   Arguments:
!       list:       Array to contain the list
!       list_size:  size of the list and the array list
!
!   Modification history:
!   Date        Name        Modification
!   Apr 2010    J Bacon     Start
!-----------------------------------------------------------------------

subroutine read_list(list, list_size)
    ! Import stuff from constants module
    use constants

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Dummy variables
    integer, intent(in) :: list_size
    real(8), intent(out) :: list(1:list_size)

    ! Local variables
    integer :: i

    do i = 1, list_size
        read *, list(i)
    enddo
end subroutine


!-----------------------------------------------------------------------
!   Description:
!       Print the list of real numbers contained in the array list
!       to a file, one number per line.
!
!   Arguments:
!       filename:   Name of the file to store the list in
!       list:       Array containing the list
!       list_size:  Size of the list and the array
!
!   Modification history:
!   Date        Name        Modification
!   Apr 2010    J Bacon     Start.
!-----------------------------------------------------------------------

subroutine print_list(list, list_size)
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Dummy variables
    integer, intent(in) :: list_size
    real(8), intent(in) :: list(1:list_size)

    ! Local variables
    integer :: i

    do i = 1, list_size
        print *, list(i)
    enddo
end subroutine
```

```
!-----------------------------------------------------------------------
!   Description:
!       Sort the list of numbers contained in the array list.
!
!   Arguments:
!       list:       Array containing the numbers
!       list_size:  Size of the list and the array
!
!   Modification history:
!   Date         Name         Modification
!   Apr 2010     J Bacon      Start.
!-----------------------------------------------------------------------

subroutine sort_list(list, list_size)
    ! Import stuff from constants module
    use constants

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Dummy variables
    integer, intent(in) :: list_size
    real(8), intent(inout) :: list(1:list_size)

    ! Local variables
    logical :: sorted
    integer :: c, low, start
    real(8) :: temp

    do start = 1, list_size
        ! Find low
        low = start
        do c = start+1, list_size
            if ( list(c) < list(low) ) then
                low = c
            endif
        enddo

        ! Swap with first
        temp = list(low)
        list(low) = list(start)
        list(start) = temp
    enddo
end subroutine
```

## 23.10 When to Use Arrays

Many people have a tendency to use the most complicated tools available to implement a solution. This is a bad tendency that stems largely from a desire to look or feel smart. Ironically, when we do this we really make ourselves look foolish, at least in the eyes of those who are wise enough to know better.

More sophisticated solutions cost more. They are harder to implement, consume more resources, are more prone to design and implementation errors, and more prone to failure.

As a simple example, consider the problem of reading a list and printing it back in forward order. This *can* be done with an array, but should it? Drawbacks of using an array:

- The size of the list is limited by available RAM.

- The program is more complicated.
- The program must traverse the list twice: Once to read it and again to print it.
- The program consumes vastly more memory than a program without an array.

In summary, the array version is bigger, slower, harder to write, and places more load on the computer running it. A programmer who uses an array to solve this problem is demonstrating poor judgment and a lack of understanding of the specifications. Good programmers always look for the simplest, most elegant solutions that minimize resource requirements.

The same argument can be made for a case where you need to add two vectors or matrices contained in files. We could inhale them into arrays, add corresponding elements, storing the results in a sum array, and finally dump the sum array to an output file. But this would be a foolish approach. We only need 3 scalar variables for this task. We can read one element at a time from each source file, add them, and immediately write the result to the sum file.

Whenever designing a solution to a problem, ask yourself if it is really necessary to retain large amounts of information. Retain *only* as much as necessary to make the solution correct and efficient.

## 23.11 Array Pitfalls

### 23.11.1 Address Calculation

```
double precision :: list(LIST_SIZE)
integer :: index
double precision :: tolerance
```

An array is a contiguous block of memory. Assuming the base address of list is 1000, the array would map into memory as follows:

| **Address** | **Content** |
|---|---|
| 1000-1007 | list(1) |
| 1008-1015 | list(2) |
| ... | ... |
| 8992-8999 | list(1000) |

Table 23.1: Memory Map of an Array

The address of an array element with index i is the base-address + (i - lowest-subscript) * (number of memory cells used by 1 element).

E.g., the address of list(i) = 1000 + (i-1) * 8.

```
            address of list(1) = 1000 + (1-1) * 8 = 1000
            address of list(2) = 1000 + (2-1) * 8 = 1008
            ...
            address of list(1000) = 1000 + (1000-1) * 8 = 8992
```

### 23.11.2 Out-of-bounds Errors

Most compilers don't generate machine code that checks array subscripts. Validating the subscript on every array reference would slow down array access significantly.

The programmer must ensure that their code uses only valid array subscripts!

```
int     list[5], c;

for (c = 0; c <= 5; ++c)
{
    list[c] = 0
    printf("%d\n", c);
}
```

```
integer :: list(5), i

do i = 1, 6
    list(i) = 0
    print *, i
enddo
```

This code will overrun the end of the array 'list'.

Assume the base address of list is 4000.

| Address | Content |
|---|---|
| 4000-4003 | list(1) |
| 4004-4007 | list(2) |
| 4008-4011 | list(3) |
| 4012-4015 | list(4) |
| 4016-4019 | list(5) |
| 4020-4023 | i |

Table 23.2: Memory Map of list and i

When setting list(6), it computes the address 4000 + 4 * (6-1) = 4020, which is the address of the integer variable 'i'. It places a 4 byte integer 0 there, overwriting 'i', and causing the loop to start over.

This demonstrates the value of using named constants or variables to specify array boundaries!

```
integer :: list(LIST_MAX), i

do i = 1, LIST_MAX
    list(i) = 0
    print *, i
enddo
```

It's easy to type 6 when you meant 5, but not so easy to type LIST_MAX+1 when you meant LIST_MAX.

### 23.11.3 Pinpointing Crashes with a Debugger

Out-of-bounds array subscripts can cause a variety of problems.

- They may cause data corruption that reveals itself immediately as shown above.
- They may corrupt other data in a way that doesn't cause any problems, or causes problems at a later stage of the program.
- They may cause different types of corruption depending on the compiler, the operating system, and the types of optimizations you compiled with. (You will often hear inexperienced programmers claim that the optimizer broke their program. This is possible in theory, but the vast majority of the time, the optimizer simply exposed a bug in their code by changing the type of corruption it caused.)

- Finally, if you're lucky, a runaway subscript may cause your program to crash. In Unix, this is usually associated with a *segmentation fault* or a *bus error*. Both of these errors indicate that the program attempted an illegal memory access.

  A segmentation fault means an attempt to access memory in a way that's not allowed (e.g. a memory address not allocated for data), whereas a bus error indicates an attempt to access memory that does not physically exist, an attempt to write to a ROM, etc.

  These errors often cause the program to dump a core, which is a snapshot of the program code and data as it appeared in memory at the moment of the crash. The core file can be used by a debugger to pinpoint exactly which statement in the program was executing when the crash occurred. For most Unix compilers, compiling with the -g flag causes more useful information for debuggers to be placed in the executable file. Increasing the optimization level (-O, -O2, -O3) reduces the debugger's ability to pinpoint problems. For more information, consult the documentation on your compiler and debugger.

## 23.12 Fortran 90 Vector Operations

### 23.12.1 Implicit Loops

Fortran provides shorthand notation for many simple array operations:

```
list(1:list_size) = 0.0d0
```

is the same as

```
do i = 1, list_size
    list(i) = 0.0d0
enddo
```

The `list(1:list_size)` is an example of an implicit loop for operating on the array. The latter example is an explicit loop. There is no difference in performance between these two Fortran code segments. The compiler will translate both to the same sequence of machine instructions.

**Caution** MATLAB has similar notations, but in MATLAB, the implicit loop is many times faster, because MATLAB is an interpreted language. The implicit loop in MATLAB would use a compiled loop, which is roughly equivalent to the loop in Fortran. The explicit loop in MATLAB is interpreted, hence explicit loops should be avoided at all cost in MATLAB.

The general form of an implicit loop of this type is

```
array(start:end:stride)
```

Examples:

```
vector(1:vector_length) = vector(1:vector_length) + 4.0d0
```

same as

```
do i = 1, vector_length
    vector(i) = vector(i) + 4.0d0
enddo
```

Example:

```
vector1(1:vector_length) = vector2(1:vector_length)
```

same as

```
do i = 1, vector_length
    vector1(i) = vector2(i)
enddo
```

Example:

```
vector(1:vector_length:2) = 3.0d0       ! Odd-numbered subscripts
```

same as

```
do i = 1, vector_length, 2
    vector(i) = 3.0d0
enddo
```

Example:

```
print *, vector1(1:vector_length)
```

same as

```
print *, (vector1(i), i=1,5)
```

It is *legal*, but *incompetent* to omit the starting and ending subscripts from an implicit loop:

```
vector1 = vector2 + vector3
```

If you do this, the program will use the minimum and maximum subscripts for the arrays. This is fine if all elements contain useful data. If the array is not fully utilized, however, the loop above will add all the useful elements *and* all the garbage. For example, if the arrays hold 1,000,000 elements, and the vectors they contain use only 3, then your loop will take about 333,333 times as long as it should.

### 23.12.2 Where

```
where (a > 0)
    b = sqrt(a)
elsewhere
    b = 0.0
endwhere
```

Same as

```
do i = 1, LIST_SIZE
    if ( a(i) > 0 ) then
        b(i) = sqrt(a(i))
    else
        b(i) = 0.0
    endif
enddo
```

## 23.13 Code Quality

1. Use named constants for the size of all arrays
2. Use size_t,

## 23.14 The Memory Hierarchy

### 23.14.1 Levels

Crude estimate as of 2022 of highly variable cache size and performance. Depends heavily on purpose of processor (low power laptop vs high performance computing / gaming server)

```
            SWAP        Gibibytes   milliseconds
            DRAM        Gibibytes   ~50+ nanoseconds
            L3 cache    ~100 MiB    ~20-40 ns
            L2 cache    ~10 MiB     ~5-10 ns
            L1 cache    ~1 MiB      ~1 ns
```

The less memory a program uses, the more likely most of its data will fit in faster levels of memory, and the lower the average memory access time (AMAT).

### 23.14.2 Virtual Memory

Average disk access time for a random block is a few milliseconds, write buffering, disk caching and predictive reads reduce this.

### 23.14.3 Cache

Multiple levels, hit ratio.

Restricting memory access to a smaller range improves hit ratio and can have a dramatic impact on program speed.

```
Filling a 64.00 KiB array 16384 times 1 bytes at a time...
   64.00 KiB array         1.00 B blocks       653.00 ms      1568.15 MiB/s

Filling a 64.00 KiB array 16384 times 2 bytes at a time...
   64.00 KiB array         2.00 B blocks       263.00 ms      3893.54 MiB/s

Filling a 64.00 KiB array 16384 times 4 bytes at a time...
   64.00 KiB array         4.00 B blocks       147.00 ms      6965.99 MiB/s

Filling a 64.00 KiB array 16384 times 8 bytes at a time...
   64.00 KiB array         8.00 B blocks        89.00 ms     11505.62 MiB/s

Testing large array, low cache hit-ratio...
Filling a 512.00 MiB array 5 times 1 bytes at a time...
  512.00 MiB array         1.00 B blocks      1905.00 ms      1343.83 MiB/s

Filling a 512.00 MiB array 5 times 2 bytes at a time...
  512.00 MiB array         2.00 B blocks       681.00 ms      3759.18 MiB/s

Filling a 512.00 MiB array 5 times 4 bytes at a time...
  512.00 MiB array         4.00 B blocks       525.00 ms      4876.19 MiB/s

Filling a 512.00 MiB array 5 times 8 bytes at a time...
  512.00 MiB array         8.00 B blocks       509.00 ms      5029.47 MiB/s
```

### 23.14.4 Memory Access Time

Example: 30ns DRAM, 2ns cache. Hit ratio 0.9, avg access time is 0.9 * 2 + 0.1 * 30 = 4.8 ns. Hit ration 0.5, avg access time is 0.5 * 2 + 0.5 * 30 = 16 ns.

With 1% VM swapping: 4ms disk (1ms avg access time), 30ns DRAM, 2ns cache. 0.01 * 1,000,000ns + 0.69 * 30ns + 0.3 * 2ns = 10,021.3ns

## 23.15 Performance

1. Eliminate arrays whenever possible Example: Input data into an array and then compute average
2. Traverse arrays sequentially rather than jump around in order to maximize memory performance
3. Compilers and optimizers assume that arrays are traversed in order.
4. Maximize locality of reference. Best done by minimizing memory space used.

### 23.15.1 Code Examples

## 23.16 Self-test

1. When should arrays be used? Why?
2. Why is it important to use dynamic memory allocation for all large arrays?
3. How does a lookup table improve program performance? What is the down side of lookup tables?
4. What is an out-of-bounds error? What kinds of problems can it cause?
5. When passing an array as an argument to a subprogram, what must always be passed with it? Why?

# Chapter 24

# Strings

## 24.1 Motivation

Most programs need to communicate with end users. To facilitate this, we have character sets such as ASCII, ISO-Latin1, and Unicode. A sequence of such characters is called a *string*.

All input from and output to a terminal is in the form of a strings, even if that input and output is numeric. When you type in the number 451.32 as input to a program, you are sending the program the characters '4', '5', '1', '.', '3', and '2'. The Fortran **read** subroutine then converts this string of characters to the appropriate binary format (usually two's complement or IEEE floating point) and stores it in some variable.

Some programs are meant to process character data, not numeric data. For example, a program to manipulate genetic data might use strings of 'a' for adenine, 'c' for cytosine, 'g' for guanine, 't' for thymine (and perhaps 'u' for uracil, which is almost identical to thymine).

## 24.2 Defining String Variables

Strings are difficult for compilers and interpreters to deal with efficiently, because they vary so much in length. Conserving memory when dealing with strings requires frequently allocating and deallocating small blocks of memory, which is costly in terms of CPU time. The only way to avoid this is by over-allocating strings (e.g. allocate 100 characters for a string even if we only need 12 at the moment). Hence, there is a trade-off to be made between CPU efficiency and memory efficiency.

There are programming techniques to minimize the cost of string manipulation, and thus improve the efficiency of programs that process character data. Some of them will be mentioned in the sections that follow.

A string in Fortran is, for all practical purposes, an array of characters. However, Fortran gives special treatment to strings, so the syntax for working with them is slightly different than it is for arrays of numbers or logical values.

When defining a string, we must indicate the *maximum* length. The syntax for this changed starting with Fortran 90, but the old syntax used through Fortran 77 is still supported for backward compatibility. All new code should use the Fortran 90 syntax, but older programs using the Fortran 77 syntax will compile with a Fortran 90 compiler.

The Fortran 90 syntax for defining string variables is:

```
character(maximum-length) :: name1, name2, ...
```

As always, we should use named constants to define things like the maximum length of a string.

```
module constants
    integer, parameter :: MAX_NAME_LENGTH=40
end module

program string_example
```

```
    implicit none

    character(MAX_NAME_LEN) :: name

    print *, 'Please enter your name:'
    read *, name
    print *, 'Hello, ', name, '!'
end program
```

The Fortran 77 and earlier syntax for defining string variables is:

```
character*maximum-length :: name1, name2, ...
```

This syntax should not be used for new programs. It is presented here so that you will recognize it if you see it in older code.

## 24.3 String Constants

A string constant, as you have already seen, is a sequence of characters between quotation marks. You can use either single or double quotes. Single quotes are most common, since they do not require pressing the shift key. However, if a string must contain a single quote, then it must be delimited using double quotes. If a string starts with a ', then the compiler will think the next ' is the end of the string.

```
print *, "You can't delimit this string with ''"
```

## 24.4 Truncation and Padding

String variables are rarely "full", i.e. they rarely contain a string of the maximum length. Fortran *pads* string variables with spaces to fill the unused elements in the string. Hence, the following two statements are equivalent:

```
name = 'Alfred E. Neumann'
name = 'Alfred E. Neumann                        '
```

**Caution** If we attempt to assign a string that is too large for the variable, it will be truncated. This can be disastrous for some programs!

```
module constants
    integer, parameter :: MAX_NAME_LENGTH=10
end module

program string_example
    implicit none

    character(MAX_NAME_LEN) :: name

    name = 'Alfred Hitchcock'
    print *, name
end program
```

Output:

```
Alfred Hit
```

## 24.5 Common String Operations

Fortran supports many operators and intrinsic functions for manipulating strings.

### 24.5.1 Concatenation

Strings can be concatenated (pasted end-to-end) using the // operator:

```
module constants
    integer, parameter :: MAX_NAME_LENGTH=20
end module

program string_example
    ! Variable definitions
    character(MAX_NAME_LEN) :: first_name, last_name
    character(MAX_NAME_LEN * 2) :: full_name

    ! Statements
    first_name = "Alfred"
    last_name = "Neumann"
    full_name = first_name // last_name
    print *, full_name
end program
```

Output?

### 24.5.2 Trimming

One of the most common operations with strings is *trimming*, or removing the trailing spaces to get just the "meat". Trimming is done with the trim() intrinsic function. Note that we cannot change the maximum size of a string variable. The trim() function allocates a smaller temporary string, whose maximum length is the actual length of its argument.

```
character(20) :: name

name = 'Alfred Nobel'

print *, name, ' is the namesake for the Nobel prize.'
print *, trim(name), ' is the namesake for the Nobel prize.'
```

```
 Alfred Nobel         is the namesake of the Nobel prize.
 Alfred Nobel is the namesake of the Nobel prize.
```

We can use trimming to fix the issue with concatenating names above.

```
module constants
    integer, parameter :: MAX_NAME_LENGTH=20
end module

program string_example
    ! Variable definitions
    character(MAX_NAME_LEN) :: first_name, last_name
    character(MAX_NAME_LEN * 2) :: full_name

    ! Statements
    first_name = "Alfred"
    last_name = "Neumann"
    full_name = trim(first_name) // ' ' // trim(last_name)
    print *, full_name
end program
```

### 24.5.3 String length

Sometimes we want to know the length of a string. The `len()` and `len_trim()` functions return an integer length of the string variable and the actual content, respectively.

```
            character(20) :: name

            name = 'Alfred Nobel'

            print *, len(name)
            print *, trim_len(name)
```

```
             20
             12
```

trim_len(string) is equivalent to len(trim(string))

### 24.5.4 Substrings

Sometimes we want to extract part of a string. We can simply specify a starting and ending subscript separated by a ':'. If either is ommitted, and a ':' is present, the missing subscript is assumed to be 1 or the maximum subscript for the string.

This syntax can also be used on the left-hand-side of an assignment statement.

```
            character(MAX_NAME_LEN) :: name

            name = 'Alfred Pennyworth'

            print *, name(1:6)  ! Alfred
            print *, name(8:17) ! Pennyworth
            print *, name(:6)   ! Alfred
            print *, name(8:)   ! Pennyworth

            name(8:) = 'E. Neumann" ! name is now 'Alfred E. Neumann'
```

## 24.6 Strings as Subprogram Arguments

When a subprogram takes a string as an argument, we may want to pass strings of different lengths.

```
        module constants
            integer, parameter :: MAX_CHROMOSOME_LEN = 1000000, &
                                  MAX_GENE_LEN = 20000, &
                                  MAX_MARKER_LEN = 20
        end module

        program string_subprograms
            implicit none

            character(MAX_CHROMOSOME_LEN) :: chrom1, chrom2
            character(MAX_GENE_LEN) :: eye_gene1, eye_gene2, hair_gene1
            character(MAX_MARKER_LEN) :: start_seq, stop_seq
            integer :: start_offset, code_offset, stop_offset

            ...

            ! Find location of sequences within a chromosome
            start_offset = locate(chrom1, start_seq)
            stop_offset = locate(chrom1, stop_seq)
            code_offset = locate(chrom1, gene_seq)

            ...
        end program
```

Unlike other arrays, it is not necessary to pass the maximum length of a string. We simply define dummy string variables with a * for the size, and the compiler takes care of the rest. The compiler will pass the length of the string to the subprogram for us, and we can retrieve it using len(string), where string is the name of the dummy argument.

```
    subroutine sub(string)
        character(*) :: string

        ! Produces the correct length of the string passed in each time
        print *, len(string)
    end subroutine
```

The locate function above can then be implemented as follows:

```
    function locate(big_sequence, sub_sequence)
        integer :: locate
        character(*) :: big_sequence, sub_sequence

        integer :: sub_len
        character :: trimmed_seq

        ! Indicate that sub_sequence was not found
        locate = -1

        ! Set locate to index of first match, if found
        trimmed_seq = trim(sub_sequence)
        sub_len = len(trimmed_seq)

        do i = 1, trim_len(big_sequence)
            if ( big_sequence(i:i+sub_len) == trimmed_seq ) then
                locate = i
                exit
            endif
        enddo
    end function
```

## 24.7 Command-line Arguments

Knowing how to handle string variables allows us to process arguments to a program from the command line. You've used programs that take arguments on the command line:

```
    mypc: ape asg01.f90
    mypc: f90 asg01.f90
```

How do programs like **ape** and **f90** get arguments like asg01.f90 from the command line?

In C, command-line arguments are received in the argument variable `argv` in the main program.

```
    cla.c.dbk
```

In a Fortran program, this is done using the intrinsic subroutine **getarg()**.

Suppose we want to write a program that computes base$^{exponent}$, but instead of asking the user to input the base and exponent, it takes them on the command line.

```
        mypc: f90 power.f90 -o power
        mypc: power 2 10
                   2  **           10  =          1024
```

The getarg() subroutine grabs a command line argument and stores it in a string variable. The first argument after the program name is argument 1, and so on. Argument 0 is the name of the program as it was invoked from the Unix command-line.

```
        character(MAX_INTEGER_DIGITS) :: program_name, &
                                         base_string, exponent_string

        call getarg(0, program_name)
        call getarg(1, base_string)
        call getarg(2, exponent_string)
```

If the argument number given is greater than the actual number of arguments, getarg() simply returns a blank string. We can use this to check whether the correct number of arguments were given. Note that argument 1 will never be missing if argument 2 is present, so we need only check the last argument to ensure that all are there.

```
        if ( exponent_string == '' ) then
            print *, trim(program_name), ': Usage: power base exponent'
            stop
        endif
```

To convert the strings to integers or reals, we use a read statement, with the string variable as the unit:

```
        integer :: base, exponent

        read (base_string, *) base
        read (exponent_string, *) exponent
```

```
!-----------------------------------------------------------------------
!   Program description:
!       Compute a power of command line arguments base and exponent
!
!   Arguments:
!       First:  An integer base
!       Second: An integer exponent
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-25  Jason Bacon Begin
!-----------------------------------------------------------------------

module constants
    ! Global Constants
    integer, parameter :: MAX_INTEGER_DIGITS = 10
end module constants

! Main program body
program power
    use constants           ! Constants defined above

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable defintions
    integer :: base, exponent
    character(MAX_INTEGER_DIGITS) :: base_string, exponent_string
    ! First command line argument is the base, second is the exponent
    call getarg(1, base_string)
    call getarg(2, exponent_string)

    ! Make sure user provided both base and exponent
    if ( exponent_string == '' ) then
  stop 'Usage: power base exponent'
    endif

    ! Convert strings to integers
    read (base_string, *) base
    read (exponent_string, *) exponent

    ! Compute power
    print *, base, ' ** ', exponent, ' = ', base ** exponent
end program
```

If we do not know how many command line arguments there are, we can use the fact that getarg() returns a blank string for any index higher than the number of arguments.

## 24.8 Code Quality

1. Use named constant + 1 for size of all string arrays
2. Do not use dangerous library functions strcpy(), strcat(), sprintf(), etc.
3. Use size_t

## 24.9 Performance

1. Avoid copying strings. Use pointers instead.
2. Avoid repeated use of strlen().

### 24.9.1 Code Examples

## 24.10 Self-test

1. Write a Fortran program that simply prints all of the command line arguments.

# Chapter 25

# File I/O

## 25.1 Motivation

Files are used for two main reasons:

- Long term storage: To store data while the programs that use is are not running.
- Short-term storage: To store temporary data that is too big for an, or is not accessed often enough to warrant keeping it in an array.

## 25.2 Filesystem Structure

As you saw in Chapter 3, files are organized into a tree-shaped structure of directories, also known as folders.

All the information presented there regarding the current working directory, relative and absolute pathnames, also applies within Fortran programs. An absolute or relative pathname is represented in Fortran as a string constant, variable, or expression.

## 25.3 C File Streams

### 25.3.1 fopen() and fclose()

### 25.3.2 Stream Read Functions

### 25.3.3 Stream Write Functions

### 25.3.4 Example Program

```
/***************************************************************************
 *  Usage:  sort input-file output-file
 *
 *  Sort a list of real numbers in input-file, placing the results
 *  in output-file.  Both input and output files contain one
 *  number per line.
 *
 *  Arguments:
 *  input-file:     file to be sorted
 *  output-file:    file to receive the sorted list
 *
 *  History:
 *  Date         Name        Modification
 *  2017-08-24   Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

double  *read_list(size_t *size, const char filename[]);
int     print_list(const double list[], size_t list_size, const char filename[]);
void    sort_list(double list[], size_t list_size);
void    usage(void);

int     main(int argc,char *argv[])

{
    double  *list;
    size_t  list_size;
    int     status = EX_OK;
    char    *input_file,        // Just for readability
            *output_file;

    if ( argc != 3 )
        usage();
    input_file = argv[1];
    output_file = argv[2];

    list = read_list(&list_size, input_file);
    if ( list == NULL )
    {
        perror(input_file);
        return EX_NOINPUT;
    }
    sort_list(list, list_size);
    if ( print_list(list, list_size, output_file) != EX_OK )
    {
        perror(output_file);
        status = EX_CANTCREAT;
    }
    free(list);
    return status;
}


/*
 *  Input list size, allocate array, and read in list.
 */

double  *read_list(size_t *size_ptr, const char filename[])

{
    size_t  c;
    double  *list = NULL;
    FILE    *input_file;

    /*
     *  If open fails, just return an error code to the caller.  This is a
     *  general-purpose function, usable in many programs, so it should not
     *  take any specific action.
     */
    input_file = fopen(filename, "r");
    if ( input_file != NULL )
    {
        fscanf(input_file, "%zu", size_ptr);    // No & here, since size_ptr is a pointer
        list = (double *)malloc(*size_ptr * sizeof(*list));
        for (c = 0; c < *size_ptr; ++c)
            fscanf(input_file, "%lf", &list[c]);
    }
    fclose(input_file);
    return list;
}

/*
 *  Print contents of a list to a file.
 */

int     print_list(const double list[], size_t list_size, const char filename[])

{
    size_t  c;
    FILE    *output_file;

    /*
     *  If open fails, just return an error code to the caller.  This is a
     *  general-purpose function, usable in many programs, so it should not
     *  take any specific action.
     */
    output_file = fopen(filename, "w");
    if ( output_file != NULL )
    {
        for (c = 0; c < list_size; ++c)
            fprintf(output_file, "%f\n", list[c]);
        return EX_OK;
    }
    fclose(output_file);
    return EX_CANTCREAT;
}


/*
 *  Sort list using selection sort algorithm.
 */

void    sort_list(double list[], size_t list_size)

{
    size_t  start,
            low,
            c;
    double  temp;

    for (start = 0; start < list_size - 1; ++start)
    {
        /* Find lowest element */
        low = start;
        for (c = start + 1; c < list_size; ++c)
            if ( list[c] < list[low] )
                low = c;

        /* Swap first and lowest */
        temp = list[start];
        list[start] = list[low];
        list[low] = temp;
    }
}


void    usage(void)
{
    fputs("Usage: selsort-files input-file output-file\n", stderr);
    exit(EX_USAGE);
}
```

```
4
5.4
4.1
3.9
2.7
```

## 25.4 C Low-level I/O

### 25.4.1 open() and close()

### 25.4.2 read()

### 25.4.3 write()

## 25.5 Fortran File Operations

Fortran, like most programming languages, uses the refrigerator model for file access. Like a refrigerator, a file must be *opened* before we can put anything into it (write) or take anything out (read). After we're done reading or writing, it must be *closed*.

A file is essentially a sequence of bytes stored on a disk or other non-volatile storage device.

Access to files is, for the most part, *sequential*, meaning we start reading a file at the first byte, and go forward from there. After we read the first byte, the next read operation automatically gets the second, and so on. It is possible to control the order in which we read or write data to a file, but doing non-sequential access is a bit more cumbersome than it is with an array. Unlike array access, it takes separate statements to go to a different location within the file and then read or write data.

### 25.5.1 Open and Close

Before we can access a file from Fortran, we must open it. The **open** statement creates a new unit, which can then be used in read and write statements to input from or output to the file.

As you saw in Section 17.2, all input and output in Fortran uses *unit numbers*.

The unit number is Fortran's way of representing the more general concept of a *file handle*. The term "file handle" is a metaphor for a mechanism used to grasp or control something. Just like a suitcase or door has a handle that allows you to manipulate it, files in Fortran and other languages have conceptual "handles" for manipulating them.

You might be wondering, isn't this what the filename is for? To some extent, yes, but when we're reading or writing a file, we need more than just the name. We need to keep track of where we are in the file, for example.

The open statement creates a structure containing information about the file, such as its name, whether we're reading or writing to is, where we are in the file at a given moment, etc. As a Fortran programmer, you do not need to keep track of this information yourself. The compiler and operating system take care of all of this for you. All you need in order to work with an open file is the unit number. This unit number is your handle to the file.

It is your job as a Fortran programmer to choose a unique unit number for each file you open. Low-numbered unit numbers such as 0, 1, and 2 are reserved for special units like INPUT_UNIT, OUTPUT_UNIT, and ERROR_UNIT. Most Fortran programmers use 10 as the lowest unit number, and go up from there if they need more than one file open at a time.

```
module constants
    integer, parameter :: &
        MAX_PATH_LEN = 1024, &
        CHROMOSOME_UNIT = 10
end module

program files
    implicit none
    integer open_status, close status
    character(MAX_PATH_LEN) :: filename

    filename = 'input.txt'

    open (unit=CHROMOSOME_UNIT, file=filename, status='old', &
        iostat=open_status, action='read', position='rewind')
    if ( open_status /= 0 ) then
        print *, 'Could not open ',filename,' for reading.', &
            'unit = ', unit
        stop
    endif

    ...

    close(CHROMOSOME_UNIT, iostat=close_status)
    if ( close_status /= 0 ) then
        print *, 'Error: Attempt to close a file that is not open.', &
            'unit = ', CHROMOSOME_UNIT
        stop
    endif
end program
```

The open statement uses a number of *tags* to specify which file to open and how it will be used.

Required tags:

- unit: integer unit number to be used by read, write, and close.
- filename: String constant, variable, or expression representing the absolute or relative pathname of the file.
- status: String constant, variable, or expression that reduces to:
  - 'old': For existing files, usually used when reading.
  - 'new': Used when writing to a file that does not yet exist.
  - 'replace': Used to overwrite a file that already exists.
- iostat: Integer variable to receive the status of the open operation. If the file is opened successfully, the variable is set to 0. Otherwise, it will contain a non-zero error code that indicates why the file could not be opened. (Does not exist, no permission, etc.)

Optional tags:

- action: String
  - 'read': Open for reading only
  - 'write': Open for writing only
  - 'readwrite': Allow both reading and writing
- position: String

- 'rewind': Start at beginning of file
- 'append': Writes add to file rather than overwrite

The close statement makes sure any writes to a file opened for writing are complete, and then disables the unit. About the only way a close can fail is if the unit does not represent an open file. This generally means there's a bug in the program, since a close statement should only be attempted if the open succeeded.

### 25.5.2 Read

The read statement works for files exactly as it does for the standard input. Recall that you can actually make the standard input refer to a file instead of the keyboard by using redirection in the Unix shell:

```
shell> asg02 < input.txt
```

The above example reads from standard input using a statement such as `read (*,*) variable`. Recall that the first '*' represents the default unit (standard input) and the second represents the default format.

When reading from a file that was opened by the program, we simply replace the first '*' with an explicit unit number.

We also introduce here the use of the iostat tag. The iostat variable will receive 0 is the read is successful, and a non-zero error code if it failed (at end of file, file is not open, etc.) Technically, the iostat tag could and should be used when reading from the standard input as well, but it was omitted in Chapter 17 for simplicity.

```
integer :: read_status
character(MAX_CHROMOSOME_LEN) :: chromosome1

read (CHROMOSOME_UNIT, *, iostat=read_status) chromosome1
if ( read_status /= 0 ) then
    print *, 'Error reading file, unit = ', CHROMOSOME_UNIT
    stop
endif
```

### 25.5.3 Write

The same ideas apply to write as to read.

```
integer :: write_status
character(MAX_CHROMOSOME_LEN) :: chromosome1

write (CHROMOSOME_UNIT, *, iostat=read_status) chromosome1
if ( write_status /= 0 ) then
    print *, 'Error writing file, unit = ', CHROMOSOME_UNIT
    stop
endif
```

## 25.6 File Format Standards

Netcdf, hdf5, nifti, minc

## 25.7 Code Quality

## 25.8 Performance

### 25.8.1 Code Examples

## 25.9 Self-test

1. What are the general steps involved in accessing a file from within a Fortran program?
2. How does reading and writing files differ from reading and writing to/from the terminal (keyboard and screen)?

# Chapter 26

# Matrices

Assigned readings: Sec 9.1, 9.2

## 26.1 Motivation

We have seen how to use arrays to store one-dimensional lists in memory.

Often there is a need to store and manage data which is conceptually multidimensional.

### 26.1.1 Tables

A table is any two-dimensional arrangement of data. Common uses are timetables, score tables, etc.

| | **Sun** | **Mon** | **Tue** | **Wed** | **Thu** | **Fri** |
|---|---|---|---|---|---|---|
| 8:00am | | | | | | |
| 9:00 | | | | | | |
| 10:00 | | | | | | |
| 11:00 | | | | | | |

Table 26.1: Time Table

### 26.1.2 Grids

Grids are representations of values at physical locations in two dimensions. Grids are commonly used to represent weather data.

| **Lat/Long** | **86** | **87** | **88** | **89** | **90** | **91** |
|---|---|---|---|---|---|---|
| 44 | 56 | 54 | 53 | 55 | 59 | 52 |
| 43 | 57 | 56 | 58 | 60 | 55 | 54 |
| 42 | 59 | 61 | 58 | 55 | 56 | 56 |

Table 26.2: Temperature

### 26.1.3 Bricks

A brick is like a grid, but with more than two dimensions. Bricks of data can represent things like MRI images, or weather data including altitude as well as latitude and longitude.

### 26.1.4 Systems of Equations

Systems of polynomial equations are commonly stored as a matrix of coefficients, which can then be solved using techniques such as Gaussian elimination.

### 26.1.5 Matrix Representation

In all of the examples above, we wish to identify one piece of data in a collection using multiple coordinates. For example, we may want to check our schedule for (Mon, 10:00), find the temperature at latitude, longitude ($43^o$, $78^o$), or measure brain activity at 10mm sagittal, 8mm coronal, and 12mm transverse (10, 8, 12).

Multidimensional data can be represented as a *matrix*, which is a multidimensional collection of numbers. A vector, defined in Chapter 23 is a is simply a matrix where only one dimension has a measurement greater than 1. For a two-dimensional matrix, the row subscript is first by convention.

| $a_{1,1}$ | $a_{1,2}$ | $a_{1,3}$ |
|---|---|---|
| $a_{2,1}$ | $a_{2,2}$ | $a_{2,3}$ |
| $a_{3,1}$ | $a_{3,2}$ | $a_{3,3}$ |

Table 26.3: Generic 2D Matrix

## 26.2 Multidimensional Arrays

C does not technically support multidimensional arrays. However, we can define an array or arrays, which accomplished the same thing.

```
#define MAX_ROWS    100
#define MAX_COLS    100

double  coefficients[MAX_ROWS][MAX_COLS];
```

Statically sized 1-dimensional arrays are wasteful enough, but the waste increases by an order of magnitude with each dimension we add, so this is generally a bad practice unless the maximum matrix size is small.

The key to conserving memory with matrices is hidden in plain view in the argv[] array received by main();

```
int     main(int argc, char *argv[])
```

The argv[] array is essentially a 2-dimensional array of characters. We can use the same concept for any 2-dimensional array.

TBD: Graphic of a pointer array

To implement a pointer array, we first allocate a 1-dimensional array of pointers to the data type contained in the matrix.

Each of these pointers will point to a 1-dimensional array of values representing a row in the matrix.

```
/***************************************************************************
 *  Description:
 *      Read a matrix into a pointer array and print it out to verify.
 *
 *  History:
 *  Date        Name        Modification
 *  2017-08-25  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <sysexits.h>
#include <stdlib.h>
```

```
typedef double  real_t;

int     main(int argc,char *argv[])

{
    size_t  rows, cols, r, c;
    real_t  **matrix;

    printf("Rows? ");
    scanf("%zu", &rows);
    printf("Cols? ");
    scanf("%zu", &cols);

    /* Allocate pointer array */
    matrix = (real_t **)malloc(rows * sizeof(real_t *));

    /*
     *  Read matrix data separated by whitespace.  Well-formatted input
     *  will have a newline after each row.
     */
    for (r = 0; r < rows; ++r)
    {
        matrix[r] = (real_t *)malloc(cols * sizeof(real_t));
        for (c = 0; c < cols; ++c)
            scanf("%lf", &matrix[r][c]);
    }

    /* Print matrix one row per line. */
    for (r = 0; r < rows; ++r)
    {
        for (c = 0; c < cols; ++c)
            printf("%5.2f", matrix[r][c]);
        putchar('\n');
    }

    /* Free the arrays */
    for (r = 0; r < rows; ++r)
        free(matrix[r]);
    free(matrix);

    return EX_OK;
}
```

```
2 3
2.0 4.3 1.0
9.7 4.7 8.2
```

In Fortran, we store matrix data in a multidimensional array. Defining multidimensional arrays is done much like one-dimensional arrays. We add more dimensions separated by commas:

```
        double precision :: coefficients(1:MAX_ROWS, 1:MAX_COLS)
```

Note that as the number of dimensions increases, so does the potential for memory waste if the array size is larger than needed. If we use 50 elements in an array of 100, half the array is wasted. If we store a 50x50 matrix in a 100x100 array, 3/4 of the memory (7,500 elements) is wasted. If we store a 50x50x50 brick in a 100x100x100 array, 7/8 of the memory (875,000 elements) is wasted. Therefore, using dynamic memory allocation becomes increasingly important as the number if dimensions grows.

```
        double precision, allocatable :: coefficients(:, :)
        integer :: rows, cols
```

```
read *, rows, cols

allocate(matrix(1:rows, 1:cols), stat=allocate_status)

if ( allocate_status /= 0 ) then
    print *, 'Error: Unable to allocate ', rows, ' x ', cols, &
        ' matrix.'
    stop
endif
```

Working with multidimensional arrays will always involve a *nested loop*, i.e. a loop inside a loop. In Fortran, the outer loop should traverse the columns, and the inner loop should traverse the rows. The reason for this is explained in Section 26.4.

## 26.3 Reading and Writing Matrices

When matrix data is stored in a file, it is most convenient to store each row of the matrix in on line of the file. In addition, it is helpful, but not necessary, to store the dimensions of the matrix at the beginning of the file. This makes it easy to read the matrix. The example below shows a matrix with two rows and three columns as it would appear in the input stream. Note that it makes no difference whether it is read from the keyboard or a file. This is how the data would appear on the screen as it is typed, or in a text file opened with an editor.

```
2 3
10.4    5.8     -1.2
0.6     3.9     8.3
```

With the file in this format, we can easily read the data into a two-dimensional array, remembering the "one statement equals one line" rule for reading input. That is, each line of the file must be read using a single read statement.

```
!-----------------------------------------------------------------------
!   Program description:
!       Input a matrix from standard input and echo to standard output
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-03-26  Jason Bacon Begin
!-----------------------------------------------------------------------

! Main program body
program echo_matrix
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Variable definitions
    double precision, allocatable :: matrix(:,:)
    integer :: rows, cols, r, allocate_status, read_status

    ! Get size of matrix
    read *, rows, cols

    ! Allocate array to hold matrix
    allocate(matrix(1:rows, 1:cols), stat=allocate_status)
    if ( allocate_status /= 0 ) then
        print *, 'Error allocating ', rows, ' x ', cols, ' matrix.'
        stop
    endif
```

```
        ! Read matrix into the array
        do r = 1, rows
            read (*, *, iostat=read_status) matrix(r,1:cols)
        enddo

        ! Print matrix to standard output
        do r = 1, rows
            print *, matrix(r,1:cols)
        enddo
    end program
```

## 26.4 Importance of Traversal Order

Fortran is a column-major language, which means that all the elements of a column in a two-dimensional matrix are adjacent in memory.

```
double precision :: matrix(1:2, 1:3)

Memory map:

2000    matrix(1,1)
2008    matrix(2,1)
2016    matrix(1,2)
2024    matrix(2,2)
2032    matrix(1,3)
2040    matrix(2,3)
```

As discussed in Chapter 12, most modern computers use *virtual memory*, which extends the apparent size of RAM using *swap space* on disk.

When a program accesses a given memory address, it is really using a virtual address, and the data stored at that virtual address could be in RAM or in swap space. If the data is in RAM, it will take nanoseconds to access. If it is in swap, it will take milliseconds, about 1,000,000 times longer.

Because it is more efficient to access disk in large chunks than in individual bytes, virtual memory is divided into *pages*. If a particular virtual address is in RAM, then the entire page surrounding it is in RAM, and if it is in swap, the entire page is in swap.

The goal when writing code is to minimize the frequency of accessing a different page than the last one accessed. If we access a value that is in swap, then we have to go to disk, which is expensive. At this point, the system moves the entire page into RAM, not just the value. This is called a *page fault*. If the next memory access is in the same page, there is no chance of causing a page fault.

Every time a page fault occurs, your program has to wait for the slow disk access, and loses a lot of time:

```
4 ms on a CPU that runs 3 billion instructions per sec =

    4 ms    1 sec     3 billion inst
          x ------- x -------------- = 12,000,000 instructions
            1000 ms        1 sec
```

Hence, if we structure our programs so that they do as many consecutive memory references as possible within the same page, we will reduce the number of page faults, and increase program performance.

This is where it is useful to know that Fortran is a column-major language. Since each column of a two-dimensional array is grouped together in memory, much or all of the column will be in the same page. Therefore, if our program works within a column for as long as possible, instead of jumping to a different column every time, it may run faster.

**Example 26.1** Comparison of Row-major and Col-major Access

Initialize a 20,000 * 20,000 matrix of doubles (400,000,000 doubles * 8 bytes = 3.2 gigabytes).

The code below traverses the matrix in row-major order, i.e. is completely sweeps one row before going to the next. In other words, it access memory addresses sequentially over the entire 2D array. This program took an average of 1.8 seconds to run over several trials on a 2.6 GHz Core i5 processor with 8 GiB RAM.

```
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

#define ROWS    20000
#define COLS    ROWS

int     main(int argc,char *argv[])

{
    static double  mat[ROWS][COLS];
    size_t  row, col;

    for (row = 0; row < ROWS; ++row)
    {
        for (col = 0; col < COLS; ++col)
            mat[row][col] = row + col;
    }

    // Access a random element of the matrix to prevent smart optimizers
    // from eliminating the loop entirely when the results are not used
    printf("%f\n", mat[random() % ROWS][random() % COLS]);
    return EX_OK;
}
```

If we switch the inner and outer loops, the program will traverse the matrix in column-major order, i.e. it will sweep each column before going to a new one, which means it accesses a different column (and hence a different page) every time it accesses the array. This causes memory addresses to be accessed non-sequentially. This slight change to the code caused the program to take an average of 10.5 seconds instead of 1.8.

```
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>

#define ROWS    20000
#define COLS    ROWS

int     main(int argc,char *argv[])

{
    static double  mat[ROWS][COLS];
    size_t  row, col;

    for (col = 0; col < COLS; ++col)
    {
        for (row = 0; row < ROWS; ++row)
            mat[row][col] = row + col;
    }

    // Access a random element of the matrix to prevent smart optimizers
    // from eliminating the loop entirely when the results are not used
    printf("%f\n", mat[random() % ROWS][random() % COLS]);
    return EX_OK;
}
```

Note that when reading or writing a matrix, we don't have much choice but to access it in row-major order. This doesn't make that much difference, since I/O is slow anyway. Traversal order is most important when processing a matrix entirely in memory. (Matrix addition, multiplication, Gauss elimination, etc.)

## 26.5 Multidimensional Arrays as Arguments

When passing a multidimensional array as an argument to a subprogram, is is even more critical that the dimensions are known.

With a one-dimensional array, the subprogram must know the size of the array in order to avoid out-of-bounds errors.

With a multidimensional array, we still have the possibility of out-of-bounds errors. In addition, if the correct dimensions are not known to the subprogram, it will not be able to compute the correct location of an element from the subscripts! Look again at the memory map of a small 2D array:

```
double precision :: matrix(2, 3)

Memory map:

2000    matrix(1,1)
2008    matrix(2,1)
2016    matrix(1,2)
2024    matrix(2,2)
2032    matrix(1,3)
2040    matrix(2,3)
```

The memory address of an element is computed as:

address(r, c) = base + (total-rows * (c-1) + (r-1)) * sizeof(type)

For an array of double precision values based at address 2000:

address(r,c) = 2000 + (2 * (c-1) + (r-1)) * 8

address(2,3) = 2000 + (2 * 2 + 1) * 8 = 2040

Since the calculation of an address involves the total number of rows in the matrix, we must at least get this right just to be able access the array elements correctly. Getting the number of columns right is also essential to prevent out-of-bounds problems.

```
subroutine print_matrix(matrix, rows, cols)
    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Dummy variables
    integer, intent(in) :: rows, cols
    double precision, intent(in) :: matrix(rows, cols)

    ! Local variables
    integer :: r

    do r = 1, rows
        print *, matrix(r, 1:cols)
    enddo
end subroutine
```

## 26.6 Fortran Intrinsic Functions

Fortran has many intrinsic functions for performing common matrix operations such as multiplication, transposition, etc. The functions provided are well optimized, and should be used in real-world applications.

However, students new to Fortran are encouraged to write their own functions for matrix transpose, multiplication, etc. in order to gain a better understanding of programming. This will help you understand and appreciate the complexity, advantages, and limitations of intrinsic functions and the Fortran language.

## 26.7 Homework

1. What does column-major mean?
2. How should two-dimensional arrays be traversed in Fortran? Why?
3. Why do two-dimensional arrays passed as arguments have to have the correct dimensions for the subprogram?

# Chapter 27

# Software Performance

## 27.1 Motivation

Many computing tasks, especially in scientific research, can take days, weeks, or months to run. Knowing how to predict the run time for a given program and set of inputs is critical to making research deadlines.

## 27.2 Performance Factors

Algorithm, language, implementation (vectorizing, mem hier), hardware.

## 27.3 Analysis of Algorithms

*Analysis of algorithms* is one of the core areas of study in computer science. It is usually one of the main topics in multiple undergraduate courses, and in graduate courses.

Analysis of algorithms is an attempt to understand the computational costs of various algorithms in a mathematical sense. Algorithms are dissected and studied carefully to determine the relationship between inputs, the algorithm, and the number of computations required.

### 27.3.1 Order of Operations: Big-O

One of the basic tools in algorithm analysis is the *Big-O notation*, or *order of operations*.

Since computing algorithms can be extremely complex, and their exact implementation will vary across different languages and hardware, predicting the exact behavior of an algorithm is too difficult a task to pursue. Part of the analysis process is eliminating factors that will not contribute significantly to the run time.

For example, the run-time of a fairly simple algorithm may turn out to be $4n^2 - 5n + 4$ seconds, where n is the number of elements in the input.

If n is 100, then $4n^2$ is 40,000, -5n is -500, and 4 is just 4. We see here that the -5n and 4 account for only about 1% of the run time. As n gets larger, they become even less significant, since $n^2$ grows much faster than n as n increases. Furthermore, we are most interested in run times when n is large, and the run times are long. When n is very small, the -5n and 4 terms may be more significant, but who cares? The program will complete in practically no time anyway.

In short, the dominant term in the run time formula when n is large is $4n^2$. We therefore discard the other terms to simplify the process of predicting run time. We also discard the constant factor 4, since it will vary with the language and hardware, and therefore tells us nothing about the algorithm.

Finally, the actually run-time will depend on many factors that are difficult or impossible to predict, such as other load on the computer at the time it runs.

The algorithm itself determines what the run time will be *proportional to*, and the constant of proportionality will be determined empirically for each computer we run it on.

Hence, for this algorithm, we say the *complexity* is O($n^2$) ("order n squared"). This means that the run time is approximately proportional to $n^2$.

### 27.3.2 Predicting Run Time

If we know the order of operations for an algorithm and have a single sample run time, we can predict run times for other input sizes fairly accurately, if certain assumptions hold.

If an O($N^2$) program takes 10 seconds to process 10,000 elements on a given computer, how long will it take to process 50,000?

1. time = K * $n^2$
2. 10 seconds = K * $10{,}000^2$
3. K = 10 / 100,000,000 = $10^{-7}$
4. $time_{50{,}000}$ = K * $50{,}000^2$ = $10^{-7}$ * $50{,}000^2$ = 250 seconds

The assumption is that the value K is the same for both values of n. This will only be true if the CPU and memory performance are the same. One case where this might not hold is when n is sufficiently large to cause the system to run out of RAM and begin using swap space. On this case, memory performance will drop, and K will increase.

### 27.3.3 Determining Order

The order of operations is relatively easy to determine for many algorithms, whereas the exact run time is nearly impossible.

Determining the order of operations for a given algorithm requires carefully analyzing what the algorithm does.

**Example 27.1** Selection Sort

Consider the selection sort algorithm. Sorting algorithms consist mainly of comparisons and swaps.
To find the smallest element in a list of N elements, the selection sort must perform N-1 comparisons. It then swaps this element with the first element.
To find the next smallest element, it must perform N-2 comparisons, and then does another swap.
Continuing on, it will do N-3 comparisons and another swap, N-4 and another swap, until the last round where it does 1 comparison and 1 swap.
In all, it does (N-1) + (N-2) + ... + 1 comparisons, which is approximately $N^2/2$, and N-1 swaps. Since number of comparisons has the highest order, the algorithm is O($N^2$).

**Example 27.2** Binary Search

Now consider searching a sorted list, such as a phone book. To conduct this search systematically, we might start in the middle, and see how the middle entry compares to the search key (the item we're searching for). If the item matches the key, we're done. If it is either less than or greater than the key, we have eliminated half the list as potential candidates. We then go to the middle of the half of the list that remains. We repeat this process until we either find the key, or we're down to one element. This algorithm is known as the *Binary Search*.
So what is the order of operations? If the search key is in the list, the actual number of comparisons is a matter of luck. We could hit it on the first try, or any try after. This is only predictable by examining the entire list. What we are interested in, though is the worst and average case. The worst case occurs when the key is either not found, or is the last item we check. The number of comparisons is the number of times we can divide the list in half, which is $\log_2(n)$. ( $2^n$ = the size of a list if we start with one element and double the list size n times. )
Hence, the order of operations is O(log(n)). We drop the base 2, since it is a constant factor: $\log_2(n) = \ln(n) / \ln(2)$.

### 27.3.4 Common Algorithms

Most commonly used algorithms have been carefully studied and their orders of operation are well known.

Note that for multidimensional data, n is taken to be a single dimension. For example, for a 100 x 100 matrix, n is 100. For a non-square matrix, we can use n = sqrt(rows * cols).

| **Algorithm** | **Order of Operation** |
|---|---|
| Selection Sort | O($n^2$) |
| Bubble Sort | O($n^2$) |
| Quicksort | O(n*log(n)) |
| Heapsort | O(n*log(n)) |
| Matrix Addition | O($n^2$) |
| Matrix Multiplication | O($n^3$) |
| Traveling Salesman | O($2^n$) |

Table 27.1: Common Orders of Operation

Note that the Traveling Salesman algorithm has exponential order. The goal of the traveling salesman problem is to determine the shortest route for a salesman to visit all of n cities once. It has been shown that to find the shortest path deterministically (with certainty) requires constructing every possible route. For n cities, there are $2^2$ possible routes that visit each city once.

Any algorithm with an exponential order is considered *intractable* (unsolvable), even on a very fast computer, because the number of calculations required grows too quickly to make it possible to solve for large values of n. Even worse than exponential time are O(n!) and O($n^n$).

When faced with problems like this, computer scientists may look for *heuristic* (probabilistic) solutions. A good heuristic solution won't guarantee an optimal result, but will provide a near-optimal result with high probability.

## 27.4 Language Selection

## 27.5 Implementation Factors

### 27.5.1 Hardware Utilization

#### Overview

*Main memory* consists of large RAM chips outside the CPU, usually mounted in special slots on the *motherboard*. Because main memory has a large number of addresses, it takes a long time to decode the address before it can read or write the memory cell. Also, RAM chips are designed to pack a large amount of memory into a small space, so power and heat dissipation take precedence over speed. This often results in the CPU sitting idle for many clock cycles while waiting for a response from the memory unit.

To reduce this memory bottleneck, most modern systems employ *cache* memory. The word cache comes from French, and means "hidden".

The cache RAM is a small, very fast RAM unit that is managed entirely by the hardware. The full details of cache operation are left to a course in computer architecture.

Access to cache requires a fraction of the time it takes to access main memory. Neither user programs nor the operating system have direct access to cache memory, but knowing that it is there and how it works can help programmers take better advantage of it.

```
+--------------------------------+
|            +---------------+   |
|    RAM     |               |   |
|            |               |   |
                        |           |               |   |
        +-------+       | +-------+ |               |   |
        |  CPU  |-------|-| cache |-|  main memory  |   |
        +-------+       | +-------+ |               |   |
                        |           |               |   |
                        |           |               |   |
                        |           |               |   |
                        |           +---------------+   |
                        +-------------------------------+
```

Each time main memory is read, a copy of the data is stored in the cache. The next time that same main memory address is accessed, the cache is checked first. If the data is still in the cache, it is taken from there, and it is not necessary to access the main memory chips at all. Since cache is much smaller than main memory, only recently accessed data will be present in the cache. The cache fills quickly when programs are running, and when it is full, old data is overwritten by the most recent data read from main memory.

The *hit ratio* for cache is defined as the number of RAM references satisfied by the cache divided by the total number of RAM references. The cache hit ratio is not as critical as the virtual memory hit ratio, but a good hit ratio can double or triple program performance. It is common in well-designed software to see a cache hit-ratio around 0.9, even if the size of cache is 1/1000 the size of main memory.

If a cache hit ratio is 0.8, cache access take 3ns, and main memory access takes 10ns, what is the average memory access time?

```
    avg access time = 0.8 * 3ns + 0.2 * 10ns
                    = 4.4ns
```

With a fairly good hit ratio, cache can more than double average memory speed. Experience has shown that a hit ratio of 0.8 is achievable with a very small amount of cache, and well-written program code. Given that a large percentage of memory references by most subprograms are to the loop counters and a few other scalar variables, hit ratios tend to be high unless arrays larger than the cache memory are in heavy use.

#### Vectorizing Interpreted Languages

Substantial improvement in some cases, but increases memory requirements if arrays are not inherently necessary and still does not approach the speed of compiled languages.

## 27.6 Hardware Speed

Throwing more hardware at a program is generally the costliest way to improve program speed. May be the only option with closed-source software or software that is beyond your ability to optimize.

## 27.7 Performance Measurement

The next question is how to obtain the sample run time needed to predict longer run times.

### 27.7.1 Timing Programs

In Unix, the **time** can be used to measure the actual time, CPU time, and system time of any program. To use it, we simply place the command "time" before the command to be benchmarked:

```
            408: f90 -O selsort.f90 -o selsort
            409: time selsort 50000nums > output
            4.985u 0.044s 0:05.03 99.8%     10+1147k 0+12io 0pf+0w
```

This shows tha the program used 4.98 seconds of user time (the program itself running), 0.044 seconds of system time (time spent by the operating system providing service to the program, such as reading the input file), and took 5.03 seconds of "wall clock time", or "real time" to complete from the moment the command was entered.

If the program is the only intensive process running, then real time should be about equal to user time + system time. If there are other programs running that compete for the CPU, memory, or disk, the real time could be significantly longer than the user + system time.

### 27.7.2 Profiling: Timing Subprograms

The time command is an easy way to determine run time for an entire process, but what if we want to determine run time for an individual subprogram? Large programs may implement many algorithms, and the time command does not allow us to see how much time each of them is using.

Profiling is a feature of most Unix compilers that allows us to see how much time each subprogram used. To use profiling, we must first compile the program with the `-p` flag (or `-pg` for GNU compilers. This causes the compiler to insert code that checks the time before and after every subprogram call. The results of these measurements are saved in a file called `mon.out`. We then use the **prof** command to see the results.

In the profile below, we can see that the sort_list() subroutine used most of the CPU time in our selection sort program.

```
weise bacon ~ 413: f90 -O -p selsort.f90
weise bacon ~ 414: ./selsort 50000nums > output
weise bacon ~ 415: prof ./selsort
 %Time Seconds Cumsecs  #Calls   msec/call  Name
  94.9    2.60    2.60       1   2600.      sort_list_
   2.2    0.06    2.66                      __f90_slw_ia32
   1.1    0.03    2.69   50000      0.0006  __f_cvt_real
   0.7    0.02    2.71                      __mt_prof_release_lock
   0.4    0.01    2.72                      __f90_open_for_output_r
   0.4    0.01    2.73   50001      0.0002  __f90_sslr
   0.4    0.01    2.74   50001      0.0002  __f90_eslw
   0.0    0.00    2.74       1      0.      main
   0.0    0.00    2.74       1      0.      read_list_
   0.0    0.00    2.74       1      0.      MAIN_
   0.0    0.00    2.74       1      0.      print_list_
   0.0    0.00    2.74       3      0.      __getarg_
   0.0    0.00    2.74       1      0.      __f90_deallocate
   0.0    0.00    2.74       1      0.      __f90_allocate2
   0.0    0.00    2.74       1      0.      f90_init
   0.0    0.00    2.74   50001      0.0000  __f90_sslw
   0.0    0.00    2.74       1      0.      __f90_slw_ch
   0.0    0.00    2.74   50000      0.0000  __f90_slw_r8
   0.0    0.00    2.74       1      0.      __f90_init
   0.0    0.00    2.74       1      0.      __f90_slr_i4
   0.0    0.00    2.74   50000      0.0000  __f90_slr_r8
   0.0    0.00    2.74   50001      0.0000  __f90_eslr
   0.0    0.00    2.74   50001      0.0000  __f90_write_a
   0.0    0.00    2.74   50001      0.0000  __f90_get_default_output_unit
   0.0    0.00    2.74   50000      0.0000  __f90_get_numbered_unit_a
   0.0    0.00    2.74       1      0.      __f90_get_numbered_unit_r
   0.0    0.00    2.74  100004      0.0000  __f90_release_unit
   0.0    0.00    2.74       1      0.      __f90_open
   0.0    0.00    2.74       1      0.      __f90_close_unit_a
   0.0    0.00    2.74       1      0.      __f90_close_a
   0.0    0.00    2.74       1      0.      __f90_close
   0.0    0.00    2.74       1      0.      __f90_initio_a
   0.0    0.00    2.74       1      0.      __f90_opencat
```

## 27.8 Homework

1. What does Big-O notation represent? How does it relate to program performance?
2. Most of the run-time for a given program results from an O($n^3$) algorithm, where n represents the number of inputs. The program takes 10 seconds to process 1000 inputs. About how long will it take to process 5000 inputs?
3. What is the order of operations for a linear search?
4. What is the order of operations for a binary search?
5. What is the order of operations for a matrix addition (n x n matrix).

# Chapter 28

# Structures

Assigned readings: Sections 10.1 - 10.3

## 28.1 Motivation

## 28.2 C Structures

## 28.3 Fortran Structures

## 28.4 Classes and Object-oriented Programming (OOP)

## 28.5 Homework

# Chapter 29

# Object Oriented Programming

Object orientied programming can be done in any language, even machine language or assembly language. It can be done quite conveniently in any language with structures and type definitions.

The decision to use a particular language should not be because "it is object-oriented". It should be based on practical considerations such a portability, performance, and needs dictated by libraries to be used. For example, Eigen is a C++ library for linear algebra. If you need to use Eigen, then you need to use C++. If you can use BLAS and LAPACK instead, then you can choose between C, C++, and Fortran. Other languages may have interfaces to BLAS and LAPACK as well, such as NumPy for Python.

```
#include <iostream>     // cin, cout, cerr
#include <sysexits.h>   // Standardized exit codes EX_OK, EX_USAGE, etc.

using namespace std;

class animal_t
{
    private:
        double weight;
    public:
        animal_t(void);
        double get_weight(void);
        void set_weight(double);
        void print(void);
};

// Constructor
animal_t :: animal_t(void)
{
    weight = 0.0;
}

// Accessor
double animal_t :: get_weight(void)
{
    return weight;
}

// Mutator
void animal_t :: set_weight(double new_weight)
{
    weight = new_weight;
    // Or this->weight = new_weight;
}

void animal_t :: print(void)
```

```
{
    cout << weight << '\n';
}

int     main()
{
    animal_t hedgehog;

    hedgehog.print();
    hedgehog.set_weight(2.4);
    hedgehog.print();
    cout << hedgehog.get_weight() << '\n';

    return EX_OK;
}
```

```
#include <stdio.h>
#include <sysexits.h>

typedef struct
{
    double weight;
}   animal_t;

// Constructor
void    animal_init(animal_t *animal)

{
    animal->weight = 0.0;
}

// Accessor
double  animal_get_weight(animal_t *animal)
{
    return animal->weight;
}

// Mutator
void    animal_set_weight(animal_t *animal, double weight)
{
    animal->weight = weight;
}

void    animal_print(animal_t *animal)
{
    printf("%f\n", animal->weight);
}

int     main(int argc,char *argv[])

{
    animal_t    hedgehog;

    // Constructor must be called explicitly, unlike C++
    animal_init(&hedgehog);

    animal_print(&hedgehog);
    animal_set_weight(&hedgehog, 2.4);
    animal_print(&hedgehog);
    printf("%f\n", animal_get_weight(&hedgehog));
    return EX_OK;
}
```

The syntax `object.function(arg);` is nothing more than a different kind of abstraction. The variable `object` is an argument to `function`, but this object-oriented syntax highlights the fact that it is an object of the class to which `function` belongs. To the compiler, this syntax has exactly the same meaning as `function(&object, arg);`.

It's a common misconception that choosing a so-called object-oriented language will result in an object-oriented program. This is false. An object-oriented design can be implemented in *any* language, and there are many programs written in object-oriented languages that do not follow object-oriented design methods at all.

Consider the C program and the Java program below. While C is not considered an object-oriented language, and Java is considered among the strictest of object-oriented languages, neither of these programs follows object-oriented design. The C program is a classic example of what we call *spaghetti code*, i.e. code without structural organization. The Java program is simply spaghetti code wrapped in a class. As this Java code is written, the public class variables are essentially global variables.

```
#include <stdio.h>
#include <sysexits.h>

void    sideEffect(void);

int     global_variable1, global_variable2, global_variable3;

int main()
{
    global_variable1 = 1;
    global_variable2 = 2;
    sideEffect();
    printf("%d\n", global_variable3);
    return EX_OK;
}

void sideEffect()
{
    global_variable3 = global_variable1 + global_variable2;
    return;
}
```

```
import java.io.*;
import java.util.Scanner;

class spaghetti
{
    static int      global_variable1, global_variable2, global_variable3;

    public static void main(String args[])
    {
        global_variable1 = 1;
        global_variable2 = 2;
        sideEffect();
        System.out.printf("%d\n", global_variable3);
        return;
    }

    public static void sideEffect()
    {
        global_variable3 = global_variable1 + global_variable2;
        return;
    }
}
```

# Chapter 30

# The Preprocessor

Assigned readings: This material is not in the textbook.

Use .F90 instead of .f90.

## 30.1 Defined Constants

## 30.2 Conditional Compilation

## 30.3 Homework

# Chapter 31

# Software Project Management

## 31.1 Build Systems

Non-trivial compiled programs are generally not contained in a single source file. At the very least, they will incorporate code from libraries (archives of precompiled code containing commonly useful functions). Many programs are also broken up into multiple source files to avoid recompiling all of the code every time a change is made. A build system allows us to recompile only the source files that have changed, and then link all the compiled modules together. Compiling is the expensive part, linking is easy and quick.

There are many different build systems available for performing basic program builds, as well as higher level tools for generating build configurations.

Browse the web and you can find many heated debates about which build system is best, mostly based on arguments that will never apply to you, e.g. build system X will perform a parallel build on a 32-core machine 15% faster than build system Y. This might matter to a developer who runs enormous builds all day every day, but for the other 99.9% of us, it's irrelevant.

As with programming languages, build systems often get blamed for user errors. "You shouldn't use build system Z, because it allows you to shoot yourself in the foot". This is a nonsensical argument of course, because careless people will find a way to mess things up no matter what tools they use, while conscientious people will do good work with any tool. Be conscientious and your life will be simpler.

**Note** For most of us, the best build system is the one we use correctly.

In this text, we will focus on **make**, the tried and true de facto standard build system for Unix. The make utility is discussed in Chapter 21. There are other systems with cool fancy features that some people get excited about, as well as a plethora of tools for generating *makefiles*, the project descriptions used by **make** for building.

However, make is easy to learn, available on every Unix system by default, and if you use it wisely, it will likely serve all of your needs very well.

## 31.2 Software Deployment

Software management is the installation and deinstallation of programs and libraries on your system.

Doing it well is very complex. It requires keeping track of dependencies between packages, including specific version requirements. It involves preventing conflicts between various packages and different versions of the same package.

With the huge number of open source applications and libraries available today, we need sophisticated tools to manage our software installations and maintain a clean system.

Generally, software developers should stay out of the software deployment business and focus their time and effort on development. Incorporating a deployment system into your own build system is an attempt to reinvent a very sophisticated wheel. It will soak up a lot of your time and you will not be very successful. Either learn to target an existing package manager like Debian packages, FreeBSD ports, Home Brew, MacPorts, pkgsrc, etc. or collaborate with a packager in one of those systems. This will save a lot of man hours for both you and end-users.

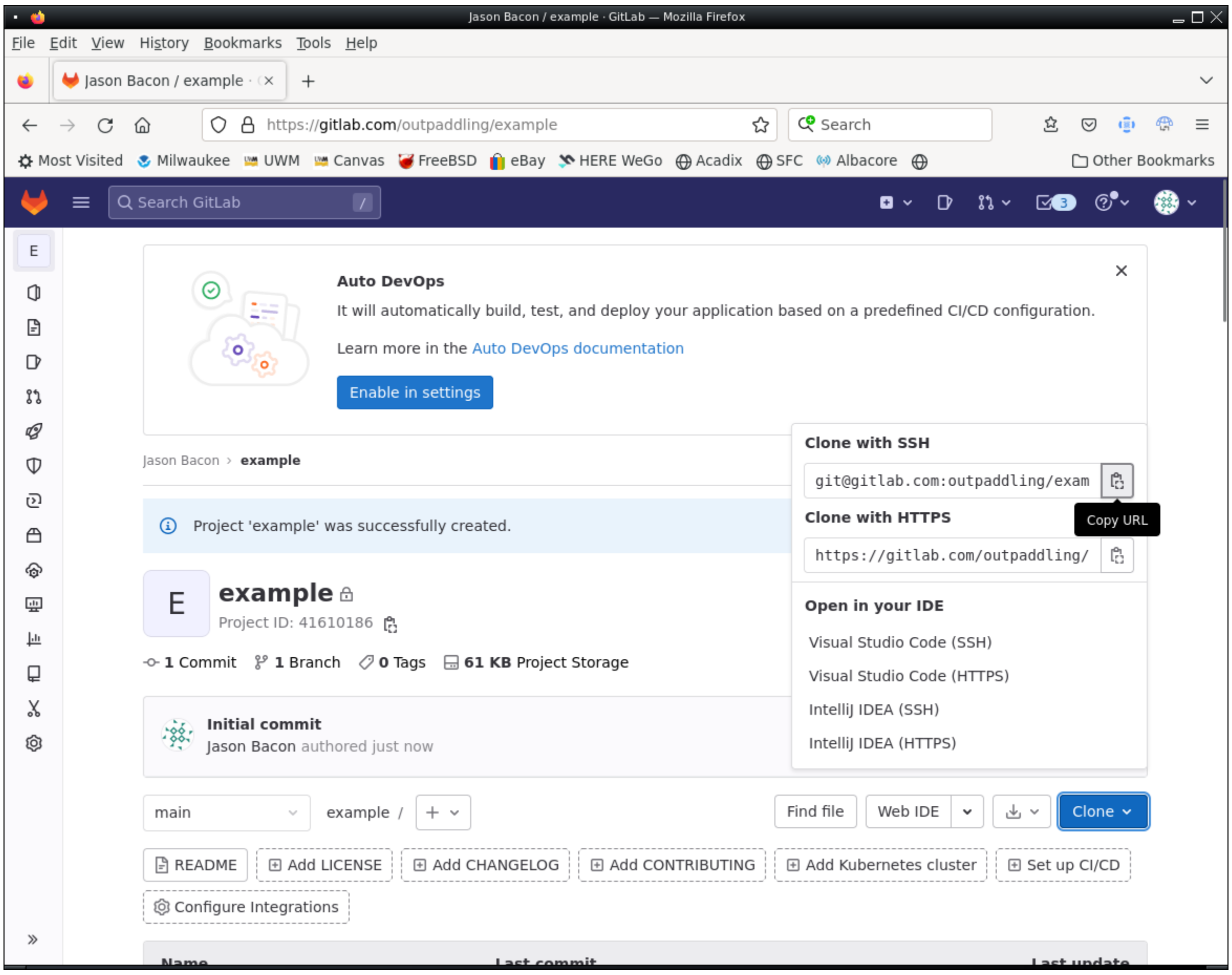
Chapter 2 and Chapter 39 discuss the basics of software installation from the end-users’ perspective. In this chapter, we will approach software development and management from the programmers’ perspective.

## 31.3 Version Control Systems

Git, Subversion

## 31.4 Source Code Hosting

1. Follow the instructions on the hosting site to create a project if you do not already have one.
2. Clone / check out the project to your local machine. Most sites offer a button on the project page to provide the URL for cloning. Using https is generally recommended over ssh. Be sure to add your account name to the URL. The URLs provided are meant for read-only access, so you will not be able to push changes unless you use add your account name.


Jason Bacon / example · GitLab — Mozilla Firefox

https://gitlab.com/outpaddling/example

Auto DevOps

It will automatically build, test, and deploy your application based on a predefined CI/CD configuration.

Learn more in the Auto DevOps documentation

Project 'example' was successfully created.

example

Project ID: 41610186

1 Commit 1 Branch 0 Tags 61 KB Project Storage

Initial commit

Jason Bacon authored just now

Clone with SSH

`git@gitlab.com:outpaddling/exam`

Clone with HTTPS

`https://gitlab.com/outpaddling/`

Open in your IDE

Visual Studio Code (SSH)

Visual Studio Code (HTTPS)

IntelliJ IDEA (SSH)

IntelliJ IDEA (HTTPS)

Figure 31.1: Getting the Gitlab Clone URL

```
shell-prompt: git clone https://outpaddling@gitlab.com/outpaddling/example.git
Cloning into 'example'...
Password for 'https://outpaddling@gitlab.com':
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

3. Add files to the project by copying them from another directory or creating them in a text editor or other tool.

```
shell-prompt: cd example
shell-prompt: cp ~/Project-files/*.txt .
shell-prompt: ape example.c
shell-prompt: git add *.txt example.c
```

4. Frequently commit and push changes to the files. This should generally be done when things are in a working state. Make small changes, perform integration testing (testing the small changes even though the program is far from complete), and then commit and push them so that they will be safe from accidental deletion, disk failure, computer theft, etc.

```
shell-prompt: git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        example.c
        input1.txt
        input2.txt

nothing added to commit but untracked files present (use "git add" to track)

shell-prompt: git add example.c input*.txt
shell-prompt: git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   example.c
        new file:   input1.txt
        new file:   input2.txt

shell-prompt: git commit -a -m 'Add example.c and test input files'
[main 288b9b4] Add example.c and test input files
 3 files changed, 40 insertions(+)
 create mode 100644 example.c
 create mode 100644 input1.txt
 create mode 100644 input2.txt

shell-prompt: git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 640 bytes | 640.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To https://gitlab.com/outpaddling/example.git
   03105ab..288b9b4  main -> main
```

## 31.5 Don't Invent your own Package Manager

Many developers are tempted to create their own automated system to build and install their software and other programs and libraries required by it. This strategy, known as *bundling* invariably creates more problems than it solves for both developers and end-users.

There are many problems with bundling:

- You cannot begin to imagine what your build system will encounter on the end-users' systems. They may be running different operating systems and many of their systems are badly managed. Your build system will fail much of the time, leading to many inquiries from frustrated users.

  Attempts to help all of these users will result in a build system that grows in complexity without limit and becomes a major drain on your time. Many developers get frustrated with this situation and end up limited support to one or a few platforms. This is very unfortunate for the community, since different developers use different platforms, and end-users may not be able to install their programs on the same machine as a result, unless they resort to using virtual machines or other systems adding unnecessary overhead.

- Libraries you bundle may conflict with installed versions during build and/or at run time.

- Bundling dependencies also makes it more difficult for people to create packages for your software, since package managers are designed to handle things in a modular fashion.

  You might argue that you need a specific version of a library that it different from what's available in your operating system's package manager. It may actually be easier to make your software work with the mainstream version of the library than to bundle another version. Even if you really need an alternative version, it's generally less problematic to create a separate package that can coexist with the mainstream version than it is to bundle the library with your software.

  If you need to modify a function in a dependent library, then rather than bundle the entire library, you could simply include the modified function in your code until your patches have been incorporated "upstream". If the linker finds the function in your program, it simply won't look for it in the installed library. Hence, your patched version will take precedence.

Bundling is, in effect, inventing and maintaining your own esoteric package manager. If you try it, you will soon discover how complex the task is and end up regretting it.

If you instead aim at making it easy to include your software in existing package managers, you will be free from all these headaches and able to focus on developing your code. Ways to do this are described in Section 31.7.

99% of software projects can be built using a simple Makefile.

Most unnecessary complexity in build systems is due to misguided attempts to automate the building of a package and some or all of the other packages on which it depends, such as libraries and build tools.

Package managers like Debian packages, FreeBSD ports, Gentoo Portage, MacPorts, pkgsrc, etc. are designed to automatically install software and track dependencies. They are used and maintained by many people around the world and thus are very sophisticated and reliable.

**Key Point**
If you make your software easy to deploy with one package manager, it will be easy to deploy with all of them.
For example, if you develop on Debian, Ubuntu, or any other Debian-based system, maintain a Debian package instead of a custom build system.
At first it may seem that this will only serve Debian users, but in fact it will serve everyone better in the long run. If you devise a simple Makefile that the Debian packaging system can use without patching or custom environment settings, then it will be easy for others to create a FreeBSD port, a MacPort, a Portage port, a pkgsrc package, an RPM, etc.
You won't have to discuss deployment issues with end users. You'll only need to deal with a handful of very savvy people who create and maintain various packages for deploying your software.

Package managers provide by far the easiest way to install, uninstall, and upgrade software.

Unfortunately, many software developers attempt to replicate the functionality of package managers with esoteric, custom build systems designed only for their software and the software it depends on.

Almost none of them work well, because developers don't have the time or resources to test them in any environment other than their own. They usually make it more difficult for end users to install your software. Your custom build system cannot come close to replicating the capabilities of a highly evolved package manager.

As a software developer, you can help yourself and end-users immensely by simply making it package-friendly, i.e. making it easy to incorporate your software into existing package managers. Ways to do this are described in Section 31.7.

Doing so leverages the work of thousands of other people, saving you a lot of work, and saving end users a lot of problems.

If you let package managers do what they're meant for, even Makefile generators like CMake and GNU autotools are largely unnecessary. A simple Makefile that respects the environment, utilizing standard make variables like CC, CXX, FC, LD, CFLAGS, CXXFLAGS, FFLAGS, and LDFLAGS, will allow your software to be easily built by most package managers.

The argument for using autotools, cmake, or some other high level build system is usually based on past problems using makefiles. However, those problems are usually the result of badly written makefiles, not the make system itself. If people don't know how to write a good makefile, adding another layer of software that they don't know how to use is not a solution. First master the use of make and do your best to make it work for your project. Then see if you really need more complexity in the build system.

Alan Greenspan famously stated that any bank that's too big to fail (without impacting the economy) is too big. Similar language can be used to describe software build systems: A project that is too big for a simple Makefile is too big, and should probably be divided into several smaller projects that can be built and installed independently of each other.

Package managers all want basically the same thing: The ability to control the building and installation of your software. If you provide a simple Makefile that works well with one package manager, it will be easy to port your software to any other. Your software will become easy to install and widely available with very little effort on your part.

For example, if you develop on a Debian-based system and focus on supporting installation your software via Debian packages, you will quickly learn how to make it easy to create a Debian package for your software. In doing so, you will fortuitously make it easy for yourself or others to create a Cygwin package, a FreeBSD port, a MacPort, an RPM, a pkgsrc package, etc. Others can even use your Debian package as a model, to see what other packages it requires, etc.

All in all, this will minimize the development effort needed to make the software accessible on numerous platforms, as well as minimize problems for end-users trying to install it.

## 31.6 Follow the Filesystem Hierarchy Standard

Most Unix-compatible systems conform to the *Filesystem Hierarchy Standard*, as described on Wikipedia, the Linux hier man page and the FreeBSD hier man page.

This standard ensures that files such as programs, headers, libraries, configuration files, and documentation are easy to find for both people and software. Installing files where the system naturally looks for them relieves users of the need to modify their environment with problematic alterations to environment variables such as PATH and LD_LIBRARY_PATH. Altering these variables generally has unintended consequences for other programs, creating headaches for users and those who support the software.

The hierarchy standard also eliminates the need for programs to use clever tricks to locate their own files, such as Perl's findbin.

Some people believe that ignoring the hierarchy standard and installing every program into its own directory

Note that the hierarchy is relocatable: Different systems implement the hierarchy for add-on software under different prefixes. For example, by default, the Debian package system installs directly under `/usr`, the FreeBSD Ports system uses `/usr/local`, and MacPorts uses `/opt/local`.

Your Makefile should allow these systems to control the installation location by using standard variable names such as DESTDIR and PREFIX, e.g.

```
install:
        ${INSTALL} myprog ${DESTDIR}${PREFIX}/bin
        ${INSTALL} myheader ${DESTDIR}${PREFIX}/include
```

This is discussed in detail in Section 31.7.

## 31.7 Package-friendly Software

If you oppose the idea of making your software easy to install, please watch this video from the FOSDEM '18 conference, How To Make Package Managers Cry.

### 31.7.1 Modularity is the Key

TBD

### 31.7.2 All You Need is a Simple Makefile

TBD

### 31.7.3 Use Conventional Tools

TBD

### 31.7.4 Archiving Standards

TBD

### 31.7.5 Respect the Environment

Look for dependencies ONLY where told to. Do not hard-code search paths for libraries, etc.

Use standard compiler variables such as provided as environment variables or make variables.

CC, CFLAGS, CXX, CXXFLAGS, FC, FFLAGS, CPP (this is the C preprocessor, not C++ compiler!), CPPFLAGS

### 31.7.6 Install the Same Files Everywhere

TBD

### 31.7.7 Versioning and Distribution Files

Semantic Versioning

### 31.7.8 A Simple Example

#### Debian Package

TBD

#### FreeBSD Port

TBD

#### MacPort

TBD

#### Pkgsrc Package

TBD

# Part IV

# Parallel Programming

# Chapter 32

# Parallel Programming

## 32.1 Serial vs. Parallel

The first rule in parallel programming: Don't, unless you really have to.

### 32.1.1 Optimize the Serial Code First

Anything you can do in serial, you can do in parallel with a lot more effort.

People often look to parallelize their code simply because it's slow, without realizing that it could be made to run hundreds of times faster without being parallelized. Using better algorithms, reducing memory use, or switching from an interpreted language to a compiled language will usually be far easier than parallelizing the code. Parallel programming is difficult, and running programs on a cluster is inconvenient compared to running them on your PC.

Some people are motivated by ego or even a pragmatic desire to impress someone, rather than a desire to solve a real problem.

Using a $1,000,000 cluster to make inefficient programs run faster or to make yourself look smart is an unethical abuse of resources. If an organization spends this much money on computing resources, it's important to ensure that they are used productively. Programmers have a responsibility to ensure that their code is well optimized before running it on a cluster.

### 32.1.2 Parallel Computing vs Parallel Programming

As explained in Chapter 33, you don't necessarily need to write a parallel program in order to utilize parallel computing resources.

In many cases, you can run simply multiple instances of a serial program at the same time. This is known as "embarrassingly parallel" computing. This is not only far easier than writing a parallel program, it also achieves better speedup in most cases, since there are no communication bottlenecks between the many processes. This type of parallel computing also scales almost infinitely. While some parallel programs can't effectively utilize more than a few processors, embarrassingly parallel computing jobs can usually utilize thousands and achieve nearly perfect speedup. ( Running N processes at once will reduce the total computation time by a factor of N. )

Think to parallelize your entire computing project, not your program. If an individual run takes only hours or days, and you have to do many runs, then embarrassingly parallel computing will serve your needs very well. Parallelizing a program is only worthwhile when you have a very long running program (usually weeks or more) that will only be run a few times.

### 32.1.3 Don't Do Development on a Cluster or Grid

You don't need parallel computing resources to develop and test parallel programs.

Parallel code can be developed and tested on a single machine, even with a single core. This is much easier and faster than developing in the more complex scheduled environment of a cluster or grid, and avoids wasting valuable resources on test runs that won't produce useful output.

You can control the number of processes used by OpenMP, MPI, and other parallel computing systems, even using more than one process per core.

Of course, you won't be able to measure speed-up until you run on multiple cores, but that doesn't matter through most of the development cycle.

Develop small test cases to run on your development and testing system, which could be a server, a workstation, or even your laptop. This will be sufficient to test for correctness, which is the vast majority of the development effort.

### 32.1.4 Self-test

1. What should always be done to a program before parallelizing it? Describe at least two reasons.
2. Explain the difference between parallel computing and parallel programming. Which implies the other? Explain.
3. What does embarrassingly parallel mean?
4. Describe at least two advantages of embarrassingly parallel computing vs other parallel computing paradigms.

## 32.2 Scheduling Compilation

### 32.2.1 Why Schedule Compilation?

A cluster consists of one or more head nodes and a number of *compute nodes*. Users log into a head node, and submit jobs which then run on one or more compute nodes. Because a head node serves a different purpose than the compute nodes within a cluster, it necessarily has a different set of software installed, and potentially a slightly different runtime environment.

Because of the configuration differences between the head node and the compute nodes, it is possible that software configured and compiled on the head node may not work properly on the compute nodes. It is therefore advisable to configure and compile software on a compute node, to ensure that it will run in the exact same environment in which it was configured and compiled.

Multiple users compiling large programs on the head node would also create a heavy load on the head node that would impact other users.

Since compute nodes are allocated by the scheduler for computational processes, it would not be safe to simply choose a compute node manually and compile our software on it. Doing so could interfere with a scheduled job. Therefore, compilations should be scheduled like any other process that requires a compute node.

On a heterogeneous cluster or grid (different nodes have different CPU architectures or different operating systems) you may want to compile the program on every node as part of the execution job. If a program is going to run for hours or days, adding seconds or minutes of compilation time to each process is a trivial cost.

### 32.2.2 Batch Serial and Batch Interactive

For large compilations, you may wish to schedule a batch serial job as outlined in Section 7.3.3. This method will create an output file containing screen output from the build commands. An example build script for a software package with a Makefile is shown below. Note that each package may have its own build method, so the commands in the script below will need to be changed for each package you build.

```
#!/bin/sh

# SLURM submit script for compilation

make
```

For smaller builds, you may prefer to watch the compiler output as it occurs, in which case a batch interactive job would be more suitable. Batch interactive jobs are described in Section 7.3.3.

```
#!/bin/sh -e

# -e terminates script if any command fails

./configure --prefix $HOME/myprogs
make
make install
```

Scheduling compilations this way will not introduce any measurable delays in compiling your code. Dispatching of batch-serial and interactive jobs generally occurs almost immediately, since there is almost always one free core available in the cluster.

### 32.2.3 Self-test

1. Why should compilation be done under the scheduler on a cluster or grid?
2. Write a script for the scheduler or your choice that submits a job for compiling the program `matrix.c` to an executable called `matrix`.

## 32.3 Further Reading

More information on parallel programming resources such as books, courses, and websites is available on cluster services website: http://www4.uwm.edu/hpc/

# Chapter 33

# Programming for HTC

**Before You Begin**
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 3), the Unix shell (Section 3.3.3, redirection (Section 3.13), shell scripting (Chapter 4), and have some experience with computer programming.

## 33.1 Introduction

*High Throughput Computing* (HTC) is the simplest and generally most economical form of distributed parallel computing.

In high throughput computing, processes are dispatched to run *independently* on multiple computers at the same time. The processes are typically *serial* programs, not *parallel* programs, so HTC can be thought of as parallel computing without parallel programming.

HTC is sometimes referred to as *embarrassingly parallel* computing, since it is so easy to implement.

Usually, the same program runs on all computers with different data or inputs, but the definition of HTC is not limited to this scenario. If you are running multiple independent processes simultaneously on different computers, you're using HTC.

HTC has several advantages:

- It's the easiest type of parallelism to implement. No parallel programming is necessary. It's just a matter of running a serial program in multiple places at the same time.
- It does not require a high-speed network or any other special hardware. HTC often utilizes lab computers across a college campus, or even home computers around the world.
- It scales almost infinitely. Since the processes are independent of each other, there is no communication overhead between them, and you can run as many processes as you have cores to run them on with nearly perfect speedup. That is, if you run 1000 processes at once, the computations will finish almost 1000 times faster than if you ran them one at a time.

### 33.1.1 Self-test

1. Define high throughput computing.
2. What is meant by embarrassingly parallel?
3. Describe three advantages of HTC over other types of parallel computing.

## 33.2 Common Uses for HTC

### 33.2.1 Monte Carlo Experiments

Monte Carlo experiments or simulations are so named because they use random inputs to a system to uncover the behavior of that system. (Monte Carlo is a famous gambling site in the French Riviera).

One of the Monte Carlo experiments that is easiest to understand yet quite paradigmatic is the calculation of pi. It begins with the simple calculation. If one has a circle with radius r, the area is pi*$r^2$. A square that encompasses that circle has side length of 2*r and area of 4*$r^2$. Therefore the ratio of the area of the circle over the area of the square is (pi*$r^2$) / (4*$r^2$) = pi/4.

Monte Carlo calculation of pi is the equivalent of throwing a dart at the square many times, then dividing the number of times it hit the circle by the total number of throws. If our dart thrower is bad enough to produce a spread of darts that is randomly and uniformly distributed across the entire square, that ratio would be pi/4. A more skilled dart thrower would produce a non-uniform distribution, which would render the simulation useless.

This task can easily be broken down in an embarrassingly parallel fashion. Each calculation of the "dart throw" is independent of the others, so we can distribute the work to as many independent processes as we like.

The main trick with this and other Monte Carlo simulations is making sure that the input to the calculation is as random as it can be and of the correct statistical distribution (uniform in this case). While it is impossible to generate truly random numbers on a computer, generation of pseudo-random numbers is well developed and provided by most programming languages. Calculation of large groups of pseudo-random numbers that is as close to truly random as possible is a major part of Monte Carlo simulation.

**Caution** The standard C libraries include older random number functions `rand()` and `srand()`. These functions are considered obsolete because they produce relatively predictable and frequently repeating sequences. They have been superseded by `random()` and `srandom()`, which generate much higher quality pseudo-random sequences. Many other languages also offer multiple pseudo-random number generators of varying quality, so be sure to find out which ones will work best for your purposes.

This task is easily broken up into independent smaller tasks. Hence, a simple approach to improve our results without taking more time would be running many simulations on different computers at the same time, and then averaging the results of all of them.

However, since pseudo-random number generators follow a predictable sequence, we have to be careful with this approach.

If we split this job into N processes, each one generating 500,000 random points, and the pseudo-random number generator started with the same value for each, then each process would generate the exact same pseudo-random number sequence and ultimately the exact same estimate for pi! Averaging the results of this parallel simulation would therefore produce exactly the same answer as each individual process.

The solution to this problem is to use a different *seed* for each process. The seed serves as a starting point for the pseudo-random number sequence, so if we use a different seed for each process, we will get different results. In C, the pseudo-random numbers generated are integers, so some commonly used seeds are the current system clock and the process ID.

```
srandom((unsigned int)time(NULL));
srandom((unsigned int)getpid());
```

Below is a simple C program that approximates pi given a number of "dart throws".

```
/***************************************************************************
 *  Description:
 *      Estimate PI using Monte Carlo method.
 *
 *  Returns:
 *      NA
 *
 *  History:
 *  2012-06-29  Jason Bacon Derived from calcpi-parallel.c
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sysexits.h>
#include <time.h>

#define TOTAL_POINTS    100000000

int     main(int argc, char *argv[])

{
    int     i;
    int     inside_circle;

    double  x;
    double  y;
    double  distance_squared;

    // Initialize counters
    inside_circle = 0;

    // Initialize pseudo-random number sequence with a different value
    // each time
    srandom(time(NULL));

    // Compute X random points in the quadrant, and compute how many
    // are inside the circle
    for (i = 0; i < TOTAL_POINTS; i++)
    {
        // random() and RAND_MAX are integers, so integer division will
        // occur unless we cast to a real type
        x = (double)random() / RAND_MAX;
        y = (double)random() / RAND_MAX;

        // No need to compute actual distance.  We need only know if
        // it's < 1 and it will be if distance_squared < 1.
        distance_squared = x * x + y * y;
        if (distance_squared < 1.0)
            inside_circle++;
    }

    printf("Inside circle: %d  Total: %d  PI ~= %f\n",
        inside_circle, TOTAL_POINTS, (double)inside_circle / TOTAL_POINTS * 4);

    return EX_OK;
}
```

**Caution**

The time() function has a resolution of seconds, so assuming the clocks of all the computers are in sync, it is likely that it will return the same value to many of the processes in a job array, which are all started at about the same time. Hence, many processes would produce duplicate results, rendering all but one of them useless.
Although very unlikely, the process ID returned by getpid() could by sheer coincidence be the same for processes running on different computers, since it is the Unix PID, not scheduler job/task/process ID.
For job arrays, using the scheduler job/task/process ID is a simple way to ensure a different value for every process.

Shown below is another version of the C program, calcpi-parallel.c. There are three command-line arguments as inputs:

```
shell-prompt: ./calcpi-parallel points process-index total-processes
```

We made the number of points a command-line argument rather than a constant in the program so that we don't have to recompile in order to run a different experiment.

The process index provides a different seed to each process. The total-processes argument is not necessary, but is useful in the program to check for sanity errors in the command such as in invalid process index. Requiring a little redundancy from the end-user can often help catch mistakes that would be hard to track down later.

The output of calcpi-parallel is the counts of points inside the circle and the total number of points. We need a separate program that averages all this output to approximate pi.

```
/***************************************************************************
 *  Description:
 *      Estimate PI using Monte Carlo method.  Multiple instances of
 *      this program are run in parallel, and results of all are used
 *      to estimate the value of PI.
 *
 *  Arguments:
 *      argv[1]: index for this process
 *      argv[2]: total number of processes
 *
 *  Returns:
 *      NA
 *
 *  History:
 *  Date        Name        Modification
 *  2011-08-15  Lars Olson  Begin
 *  2011-09-27  Jason Bacon Replace rand()/srand() with random()/srandom()\
 *                          Add usage message
 *  2012-06-29  Jason Bacon Use different seed for each process to simplify
 *                          code.  Add comments and use more descriptive
 *                          variable names.
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sysexits.h>

void    usage(char *progname)

{
    fprintf(stderr, "Usage: %s total-points process-index total-processes\n", progname);
    exit(EX_USAGE);
}


int     main(int argc, char *argv[])

{
    int     process_index;
    int     total_processes;
    // Unsigned long can be either 32 or 64 bits and is limited to about
    // 4.3 billion if the compiler defaults to 32-bits.
    // Unsigned long long (64 or 128 bits) allows the number of points
    // to be at least 2^64-1, or 1.84*10^19.
    unsigned long long  total_points,
                        inside_circle,
                        i;

    double  x;
    double  y;
    double  distance_squared;

    char    *end_ptr;

    // Make sure program is invoked with 2 command line arguments
    // argc = 3 for program name + arguments
    if ( argc != 4 )
        usage(argv[0]);

    // First command line argument is which job out of the 10 it is
    total_points = strtouq(argv[1], &end_ptr, 10);

    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n", argv[1]);
        usage(argv[0]);
    }

    // First command line argument is which job out of the 10 it is
    process_index = strtol(argv[2], &end_ptr, 10);

    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n", argv[1]);
        usage(argv[0]);
    }

    // Total number of processes in the job, 10 in our examples.
    total_processes = strtol(argv[3], &end_ptr, 10);

    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 2 (%s) is not an integer.\n", argv[2]);
        usage(argv[0]);
    }

    if ( total_processes < 1 )
    {
        fputs("Total processes must be > 0.\n", stderr);
        exit(EX_USAGE);
    }

    if ( (process_index < 1) || (process_index > total_processes) )
    {
        fputs("Process index must be between 1 and total processes.\n", stderr);
        exit(EX_USAGE);
    }

    // Use a different seed for each process so that they don't all
    // compute the same pseudo-random sequence!
    srandom(process_index);

    // Initialize counters
    inside_circle = 0;

    // Compute X random points in the quadrant, and compute how many
    // are inside the circle
    for (i = 0; i < total_points; i++)
    {
        // random() and RAND_MAX are integers, so integer division will
        // occur unless we cast to a real type
        x = (double)random() / RAND_MAX;
        y = (double)random() / RAND_MAX;

        // If distance squared < or > 1.0, then distance < or > 1.0, so don't
        // waste time calculating the square root.
        distance_squared = x * x + y * y;
        if (distance_squared < 1.0)
            inside_circle++;
    }

    // %q wants unsigned long long, which can be 64 or 128 bits
    // Cast the uint64_t variables to silence gcc warnings
    printf("%llu %llu\n", inside_circle, total_points);

    // Debugging only: not normal output used by scripts
    // Comment out before running job
    // printf("%f\n", (double)inside_circle / total_points * 4.0);

    return EX_OK;
}
```

### Software Performance Optimization

Software efficiency is a huge part of high performance and high throughput computing.

Improvements to software often result in an order of magnitude or more reduction in run time and in many cases make the use of parallel computing unnecessary.

Optimizing software before consuming expensive parallel computing resources can therefore be an ethical matter in many cases.

A little understanding of computer hardware can go a long way toward improving performance.

One prime example is understanding the difference between integers and floating point. Floating point types such as float and double (real and double precision in Fortran) are inherently more complex than integers and therefore require more time for operations such as addition and multiplication.

Floating point types are stored in a format similar to scientific notation. As you may recall, adding scientific notation values is a three-step process:

1. Equalize the exponents
2. Add the mantissas
3. Normalize the result

Hence, we might expect floating point addition to take about three times as long as integer addition. Due to optimizations in floating point hardware, the difference is not that great, but it can be significant.

It pays to examine your program code and consider whether the use of floating point is really necessary. In most cases, including our calcpi program, it is not. The random() function in C produces integer values between 0 and RAND_MAX, a constant defined in the header file stdlib.h. We can just as easily perform these calculations using only integers, if we use quadrant dimensions of RAND_MAX x RAND_MAX instead of 1 x 1 and reduce the units of X and Y, so that all of our X and Y values are integers between 0 and RAND_MAX.

A modified version of our calcpi program using only integers is shown below:

```
/***************************************************************************
 *  Description:
 *      Estimate PI using Monte Carlo method.  Multiple instances of
 *      this program are run in parallel, and results of all are used
 *      to estimate the value of PI.
 *
 *  Arguments:
 *      argv[1]: index for this process
 *      argv[2]: total number of processes
 *
 *  Returns:
 *      NA
 *
 *  History:
 *  Date        Name        Modification
 *  2011-08-15  Lars Olson  Begin
 *  2011-09-27  Jason Bacon Replace rand()/srand() with random()/srandom()\
 *                          Add usage message
 *  2012-06-29  Jason Bacon Use different seed for each process to simplify
 *                          code.  Add comments and use more descriptive
 *                          variable names.
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sysexits.h>
#include <inttypes.h>

void    usage(char *progname)

{
    fprintf(stderr, "Usage: %s total-points process-index total-processes\n", progname);
    exit(EX_USAGE);
}


int     main(int argc, char *argv[])

{
    int     process_index;
    int     total_processes;
    // Unsigned long can be either 32 or 64 bits and is limited to about
    // 4.3 billion if the compiler defaults to 32-bits.
    // Unsigned long long (64 or 128 bits) allows the number of points
    // to be at least 2^64-1, or 1.84*10^19.
    // The printf() function has no standard placeholders for uint64_t,
    // so we'll use unsigned long long here to avoid compiler warnings.
    // This won't affect performance much, since these variables are only
    // incremented in the main loop.
    unsigned long long  total_points,
                        inside_circle,
                        i;

    // random() returns 32-bit values.  The square of a 32-bit value can
    // be up to 64-bits long, so make sure the program uses 64-bit integers
    // on all processors and for all calculations.
    uint64_t    x;
    uint64_t    y;
    uint64_t    distance_squared,
                // RAND_MAX is a 32-bit integer, so cast it to 64 bits
```

```
            // before multiplying to avoid a truncated result.
            rand_max_squared = (uint64_t)RAND_MAX * (uint64_t)RAND_MAX;

    char    *end_ptr;

    // Make sure program is invoked with 2 command line arguments
    // argc = 3 for program name + arguments
    if ( argc != 4 )
        usage(argv[0]);

    // First command line argument is which job out of the 10 it is
    total_points = strtouq(argv[1], &end_ptr, 10);

    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n", argv[1]);
        usage(argv[0]);
    }

    // First command line argument is which job out of the 10 it is
    process_index = strtol(argv[2], &end_ptr, 10);

    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n", argv[1]);
        usage(argv[0]);
    }

    // Total number of processes in the job, 10 in our examples.
    total_processes = strtol(argv[3], &end_ptr, 10);

    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 2 (%s) is not an integer.\n", argv[2]);
        usage(argv[0]);
    }

    if ( total_processes < 1 )
    {
        fputs("Total processes must be > 0.\n", stderr);
        exit(EX_USAGE);
    }

    if ( (process_index < 1) || (process_index > total_processes) )
    {
        fputs("Process index must be between 1 and total processes.\n", stderr);
        exit(EX_USAGE);
    }

    // Use a different seed for each process so that they don't all
    // compute the same pseudo-random sequence!
    srandom(process_index);

    // Initialize counters
    inside_circle = 0;

    // Compute X random points in the quadrant, and compute how many
    // are inside the circle
    for (i = 0; i < total_points; i++)
    {
        x = (uint64_t)random();
        y = (uint64_t)random();

        // If distance squared is < or > randmax_squared, then distance is
        // < or > randmax, so don't waste time calculating the square root.
        distance_squared = x * x + y * y;
        if (distance_squared < rand_max_squared)
            inside_circle++;
    }

    // %q wants unsigned long long, which can be 64 or 128 bits
    printf("%llu %llu\n", inside_circle, total_points);

    // Debugging only: not normal output used by scripts
    // Comment out before running job
    // printf("%f\n", (double)inside_circle / total_points * 4.0);

    return EX_OK;
}
```

Timing this program on an AMD64 system shows that it is more than twice as fast as the earlier floating point version.

Note, however, that this calcpi requires the use of 64-bit integers. 32-bit processors can only do integer operations of 32 bits at a time, although most can to 64-bit floating point operations all at once.

When using 64-bit integers on a 32-bit processor, we don't see the same benefit, since the 32-bit processor has to do the 64-bit integer operations in two steps. However, on a 32-bit Pentium 4 processor, the integer calcpi still performed about 25% faster than the floating point version. If you're looking at a month of calculations, the switch to integer arithmetic would still knock off about a week.

Furthermore, many programs can get by with 32-bit integers, in which case you'll realize the full performance benefit even on 32-bit hardware.

### Calcpi on a SLURM Cluster

When compiling a program on a cluster or grid, it should be compiled on the same node(s) it will run on to ensure that it is built and run in the exact same environment and in the presence of the exact same libraries and compiler tools. The head node necessarily has different software installed, which may cause a program to be compiled differently than it would be on a compute node. Hence, we use a simple sbatch script to do the compilation:

```
#!/bin/sh -e

# Show commands in output
set -x
cc -O -o calcpi-parallel-integer calcpi-parallel-integer.c
```

```
peregrine: sbatch calcpi-parallel-build.sbatch
```

To run the program in embarrassingly parallel fashion, we use a facility in SLURM and other queuing systems to run multiple instances of a single program, where each can receive different command line arguments and/or inputs. Consider the following SLURM submit script:

```
#!/bin/sh -e

#SBATCH --array=1-20
#SBATCH --output=calcpi-parallel-%a.out
#SBATCH --error=calcpi-parallel-%a.err
```

```
points=1000000000

# Make sure last argument matches total number of jobs in array!
./calcpi-parallel-integer $points $SLURM_ARRAY_TASK_ID 20
```

This script starts up an embarrassingly parallel job, known as a *job array*. The submit script is executed almost simultaneously on multiple cores in the cluster, one for each value in the range specified with `--array`.

The element ${SLURM_ARRAY_TASK_ID} is a reference to an environment variable which is set by the SLURM scheduler to a different value for each job in the job array. With --array=1-10, SLURM_ARRAY_TASK_ID will be 1 for one process, 2 for another, and so on up to 10.

Each process in the job array will also produce a separate output and error file with the value of SLURM_ARRAY_TASK_ID appended. I.e., there will be files called calcpi-parallel-1.out, calcpi-parallel-2.out, etc.

After all the processes in the job array complete, the output can be tallied using a simple script such as the following:

```
#!/bin/sh

# If this is non-trivial, it should be done by submitting another job
# rather than run on the submit node.

# Results are on last line of each output file calcpi-parallel.out-1, etc.
# Both the SLURM and HTCondor scripts must name their output files to match
# this script.

# Send last line of each output file through a simple awk script that
# sums up each column and prints the quotient of the two sums.
tail -1 -q calcpi-parallel-*.out | awk \
'BEGIN  {
            in_circle=0;
            total=0;
        }
        {
            in_circle+=$1;
            total+=$2;
        }
END     {
            printf("%0.20f\n", in_circle / total * 4.0);
        }'
```

**Note** If a script such as the tally script above is trivial, some users may choose to run it on the head node or on their own computer after transferring the results from the cluster. If it requires any significant amount of CPU time or memory, however, it should be scheduled as a batch-serial job.

Another script (to be run directly, not submitted to the scheduler) could be used to clean up from previous runs and submit the job again:

```
#!/bin/sh -e

##########################################################################
#   Script description:
#       Run a set of SLURM jobs to estimate PI, and combine the results.
#       An alternative to this approach is a DAG.
#
#   Arguments:
#       None.
#
```

```
#   Returns:
#       0, or status of first failed command.
#
#   History:
#   Date        Name        Modification
#   2013-01-29  Jason Bacon Begin
##########################################################################

##########################################################################
#   Main
##########################################################################

# Remove old output and log files
./clean

# Build program
sbatch calcpi-parallel-build.sbatch

# Submit computational jobs
sbatch calcpi-parallel-run.sbatch
```

Finally, the directory can be cleaned up using yet another script that we run directly:

```
#!/bin/sh

rm -f calcpi-parallel *.out* *.err* \
    calcpi-parallel.[0-9]* \
    calcpi-parallel-condor.log
```

### Calcpi on an HTCondor Grid

Recall from Chapter 9 that HTCondor is a scheduling tool like SLURM, PBS or LSF. Unlike most schedulers, which are geared toward HPC clusters built entirely with dedicated hardware, HTCondor is specifically designed for grids that utilize a variety of hardware owned by a variety of people or departments, such desktop machines throughout your institution.

On the cluster, where all compute nodes run the same operating system and have access to the same files, we ran a separate build script to compile our code on one node, and then ran that executable on all nodes.

Since grids do not typically have shared file space, and are often heterogeneous (use different hardware and/or operating systems on various execute hosts), our approach to running code must be different.

If we are running software that's preinstalled on the HTCondor compute hosts, as is often the case with programs like Octave and R, we must be aware that it may not behave identically on all hosts. Some hosts may be running a 32-bit operating system, limited to a few gigabytes of RAM, while others run 64-bit operating systems. There may be different versions of the software installed on different hosts.

These are additional issues we may need to deal with when using an HTCondor grid, but they are typically not difficult to resolve.

**Note** In HTCondor lingo, an *execute host* is equivalent to what we call compute nodes in a cluster.

Since we typically have no shared file system, we may need to transfer the program and input files to every execute host we use, and bring back any output files when the processes are done. All of this is handled automatically by HTCondor.

Since the execute hosts may have different hardware and operating systems, we can't compile the program on one of them and expect it to run on all of them. Instead, it's a common practice to transfer the source code to each execute host and compile it there as part of the job. This ensures that every binary file is compatible with the host it's running on.

An HTCondor description file for running `calcpi-parallel.c` might appear as follows:

```
###########################################################################
# Sample condor submit description file.
#
# Use \ to continue an entry on the next line.
#
# You can query your jobs by command:
# condor_q

###########################################################################
# Choose which universe you want your program is running with
# Available options are
#
# - standard:
#      Defaults to transfer executables and files.
#      Use when you are running your own script or program.
#
# - vanilla:
# - grid:
#      Explicitly enable file transfer mechanisms with
#      'transfer_executable', etc.
#      Use when you are using your own files and some installed on the
#      execute hosts.
#
# - java:
#      Explicitly enable file transfer mechanism. Used for java jobs.
#
# - scheduler
# - local
#
# - parallel:
#      Explicitly enable file transfer mechanism. Used for MPI jobs.
#
# - vm
#      Refer http://research.cs.wisc.edu/condor/manual/v7.6/2_4Road_map
#      Running.html for detailes.

universe = vanilla

# Macros (variables) to use in this submit description file
Points = 1000000000
Process_count = 100

###########################################################################
# Specify the executable filename.  This can be a binary file or a script.
# NOTE: The POVB execute hosts currently support 32-bit executables only.

executable = calcpi-parallel-condor.sh

# Command-line arguments for the execute command
# arguments =

###########################################################################
# Set environment variables for use by the executable on the execute hosts.
# Enclose the entire environment string in quotes.
# A variable assignment is var=value (no space around =).
# Separate variable assignments with whitespace.

environment = "Process=$(Process) Process_count=$(Process_count) Points=$(Points)"

###########################################################################
# Where the standard output and standard error from executables go.
# $(Process) is current job ID.

# We run calcpi-parallel under both SLURM and Condor, so we use same output
# names here as in calcpi-parallel-run.sbatch so that we can use the same
# script to tally all the outputs from any run.

output = calcpi-parallel-$(Process).out
error = calcpi-parallel-$(Process).err

############################################################################
# Logs for the job, produced by condor.  This contains output from
# Condor, not from the executable.

log = calcpi-parallel-condor.log

############################################################################
# Custome job requirements
# Condor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!!
# For example, if the jobs is submitted from peregrine, target.arch is
# "X86_64" and target.opsys is "FREEBSD8", which do not match
# POVB execute hosts.
#
# You can query if your submitting host is accepted by command:
#  condor_q -analyze

# Memory requirements in megabytes
request_memory = 1000

# Requirements for a binary compiled on 32-bit CentOS 4 (POVB hosts):
# requirements = (target.arch == "INTEL") && (target.opsys == "LINUX")

# Requirements for a Unix shell script or Unix program compiled on the
# execute host:
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
               ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Requirements for a job utilizing software installed via FreeBSD ports:
# requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
#     (target.opsys == "FREEBSD")

############################################################################
# Explicitly enable executable transfer mechanism for vanilla universe.

# true | false
transfer_executable = true

# yes | no | if_needed
should_transfer_files = if_needed

# All files to be transferred to the execute hosts in addition to the
# executable.
transfer_input_files = calcpi-parallel.c

# All files to be transferred back from the execute hosts in addition to
# those listed in "output" and "error".
# transfer_output_files = file1,file2,...

# on_exit | on_exit_or_evict
when_to_transfer_output = on_exit

############################################################################
```

```
# Specify how many jobs you would like to submit to the queue.

queue $(Process_count)
```

The executable script:

```
#!/bin/sh -e

# Use Bourne shell (/bin/sh) tomaximize portability of this script.
# There is no guarantee that other shells will be installed on a given host.

# Use /bin/sh -e to quit on the first error, so we don't try to run
# subsequent commands after a command in this script already failed.

# This script is meant to run on any Unix system (FreeBSD, Linux, Solaris,
# etc.), so use only generic, portable Unix commands and flags.
# (e.g. cc instead of gcc or icc).

# Bourne shell has a bare-bones PATH that won't find some add-on software.
# Also, startup scripts are not executed by Bourne shells under condor,
# so local additions to PATH in /etc/* and ~/.* will not be picked up.
# If you know the path of programs you use on certain hosts, add it here.
# /usr/libexec      Dependencies for some standard tools
# /usr/local/bin    FreeBSD ports
# /opt/local/bin    MacPorts
# /sw/bin           Fink
PATH=${PATH}:/usr/libexec:/usr/local/bin:/opt/local/bin:/sw/bin
export PATH

# Some slots may be on the same execute node so use a different executable
# name for each process.  Compilation could fail if both compiles try
# to write the same executable file at the same time.
cc -o calcpi-parallel.$Process calcpi-parallel.c

# Condor process IDs start at 0, and calcpi-parallel expects indexes
# to start at one.
./calcpi-parallel.$Process $Points $((Process+1)) $Process_count
```

If we have designed our scripts carefully, so that they use the same output filename, etc., then we can use the same tally and cleanup scripts that we used with SLURM:

```
#!/bin/sh

# If this is non-trivial, it should be done by submitting another job
# rather than run on the submit node.

# Results are on last line of each output file calcpi-parallel.out-1, etc.
# Both the SLURM and HTCondor scripts must name their output files to match
# this script.

# Send last line of each output file through a simple awk script that
# sums up each column and prints the quotient of the two sums.
tail -1 -q calcpi-parallel-*.out | awk \
'BEGIN  {
            in_circle=0;
            total=0;
        }
        {
            in_circle+=$1;
            total+=$2;
        }
END     {
            printf("%0.20f\n", in_circle / total * 4.0);
        }'
```

```
#!/bin/sh

rm -f calcpi-parallel *.out* *.err* \
    calcpi-parallel.[0-9]* \
    calcpi-parallel-condor.log
```

A simple shell script can also be used to automate the execution of the HTCondor jobs:

```
#!/bin/sh -e

##########################################################################
#   Script description:
#       Run a set of condor jobs to estimate PI, and combine the results.
#       An alternative to this approach is a DAG.
#
#   Arguments:
#       None.
#
#   Returns:
#       0, or status of first failed command.
#
#   History:
#   Date        Name        Modification
#   2013-01-29  Jason Bacon Begin
##########################################################################

##########################################################################
#   Main
##########################################################################

# Remove old output and log files
./clean

# Submit computational jobs
condor_submit calcpi-parallel.condor
```

As an alternative to the shell script above, there is also a companion tool for HTCondor called DAGMan, which can be used to automate work flows (sequences of HTCondor jobs and other tasks). DAGMan uses DAGs (Directed Acyclic Graphs) to represent work flows. Some users may find this visual approach preferable to a script, especially for complex work flows.

### 33.2.2 Parameter Sweeps and File Transfer

One way in which the previous calcpi job is efficient is that it minimizes the amount of traffic passed throughout the cluster or grid. Ideally, only N small files each with 2 integers in them were passed back to the head node or written to the shared file system of the cluster or grid. This was made possible by the fact that the calcpi program generates its own input in the form of pseudo-random numbers.

That kind of efficiency is not always possible and oftentimes one needs to pass input files from the head node to those working on the jobs.

An example of this is illustrated by a class of problems known as *parameter sweeps*. A parameter sweep is similar to a Monte Carlo simulation, in that the same program is run on a variety of inputs.

However, the inputs in a parameter sweep are not usually random, nor are they necessarily even computable by the program. The inputs might be a range of values, and hence computable, or they may consist of vast amounts of previously collected or generated data stored in files.

In some cases, we know the correct output and need to find the input or inputs that will produce it. An example of this would be password cracking. Passwords are stored in a non-reversible encrypted form, which is often easy to obtain by reading a password file or eavesdropping on a network connection. It is not possible to *decrypt* a password directly (convert the encrypted form back to the raw form). However, it is relatively easy to *encrypt* (convert the raw form to the encrypted form), although such one-way encryption algorithms are deliberately designed to be expensive in order to slow down parameter sweeps aimed at password cracking.

In other cases, we might be given a vast amount of data collected by a scientific instrument such as a telescope, a DNA sequencer, or a microphone. Vast amounts of data could also come in the form of a large collection of electronic documents such as research papers, court documents, or medical records.

In cases like this, we will need to distribute the data to the compute nodes so that each process can work on a portion of the inputs.

When working on a cluster with a shared file system, we may be able to get away with leaving all the data in a shared directory where all of the processes can access it. If the amount of data and the number of processes working on it are very large, however, access to the shared file system can become a serious bottleneck. This will not only defeat the purpose of parallel computing for you, but will also seriously impact other users on the cluster.

In cases like this, it would be better to *prestage* the data, i.e. divvy it out to local disks on the compute nodes before computations begin. This allows the individual disks on the compute nodes to work in parallel as do the CPUs and memory.

Prestaging is especially important if the data files will be read more than once. However, even if they'll only be read once, prestaging properly will prevent impacting the shared file system while the job runs.

Note, however, that individual disks on compute nodes are typically slower than the RAID arrays used for shared space. Also, compute nodes with many cores may have processes competing for the same local disks. All these factors must be considered when deciding how to prestage the data and how to distribute the processes across the nodes. For example, when running a data-intensive parallel job in a SLURM environment, we might ask the scheduler to run one process per node to ensure that each process is using a different local disk.

On a grid with no shared file system, prestaging the data is a requirement in all cases. For this reason, HTCondor has built-in facilities for file transfer both to and from the execute hosts.

#### Calcpi-file on a SLURM Cluster

For the sake of efficient learning, we will reuse the calcpi example to illustrate file transfer. In reality, no one would choose to provide inputs to calcpi this way, but using a familiar example should make it easier for the reader to follow, since new material is limited to just the process of using input files.

For this example we will calculate the random numbers first, and stores them in N files. Then we submit an array of N processes running a modified calcpi program that reads an input file instead of generating its own random numbers and outputs the same two numbers as the previous calcpi program does. There are several steps in this overall work flow:

1. Create N data files containing different pseudo-random number sequences.
2. Create a new submission script to run a job array with N different file names as inputs.
3. If there is no shared file system, or if using a shared file system would cause a bottleneck, prestage the N files to the compute nodes (or execute hosts).
4. Submit the job(s).
5. Tally the results.

This new work flow requires a new program called calcpi-file.c.

```
/***************************************************************************
 *  Description:
 *      Estimate PI using Monte Carlo method.
 *
 *  Returns:
 *      NA
 *
 *  History:
 *  2012-07-07  Jason Bacon Derived from calcpi-parallel.c
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>

void    usage(char *progname)

{
    fprintf(stderr, "Usage: %s filename\n", progname);
    exit(EX_USAGE);
}


int     main(int argc, char *argv[])

{
    unsigned long long  i,
                        inside_circle,
                        total_points;
    extern int  errno;

    double  x;
    double  y;
    double  distance_squared;

    char    *filename;

    FILE    *points_file;

    if ( argc != 2 )
        usage(argv[0]);

    // No need to copy the string, just point to it
    filename = argv[1];

    points_file = fopen(filename, "r");
    if ( points_file == NULL )
    {
        fprintf(stderr, "Could not open %s: %s\n", filename, strerror(errno));
        exit(EX_NOINPUT);
    }

    if ( fscanf(points_file, "%llu\n", &total_points) != 1 )
    {
        fprintf(stderr, "Failed to read total_points from %s: %s\n",
            filename, strerror(errno));
        exit(EX_NOINPUT);
    }
    // Initialize counters
    inside_circle = 0;

    // Compute X random points in the quadrant, and compute how many
    // are inside the circle
    for (i = 0; i < total_points; ++i)
    {
        if ( fscanf(points_file, "%lg %lg\n", &x, &y) != 2 )
        {
            fprintf(stderr, "Failed to read point %llu from %s: %s\n",
                i, filename, strerror(errno));
            exit(EX_NOINPUT);
        }

        distance_squared = x * x + y * y;
        if (distance_squared < 1.0)
            inside_circle++;
    }

    // Error checking an fclose() is really only useful when writing to
    // a file, but we do it anyway to promote good habits.
    if ( fclose(points_file) != 0 )
    {
        fprintf(stderr, "Failed to close %s: %s\n",
            filename, strerror(errno));
        exit(EX_NOINPUT);
    }

    printf("%llu %llu\n", inside_circle, total_points);

    return EX_OK;
}
```

This program is executed as follows:

```
peregrine: ./calcpi-file rand1.txt
```

An example input file:

```
10
0.17667887507783197609 0.07581055493830263226
0.19486713744460937292 0.98018012101770379818
0.83923330383339589389 0.04710093003097033659
0.01152008865565065654 0.36655766114897919694
0.01075634686777198097 0.44170310275708468684
0.59835347514522885248 0.87296829506427431333
0.99172840173902376826 0.05354796585326453834
0.59876983128430782966 0.59682797808052412414
0.12375027133326524376 0.20448832502797634203
0.81670404123920203876 0.59261837768956016070
```

**Note** This program doesn't need to receive the number of data points as an argument. That information is determined by reading the input file.

The input files are generated by a separate program:

```
/***************************************************************************
 *  Description:
```

```
 *      Generate random (x,y) tuples.
 *
 *  Usage:
 *      gen-points count > file
 *
 *  Returns:
 *      NA
 *
 *  History:
 *  2012-07-07  Jason Bacon Derived from calcpi.c
 ***************************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sysexits.h>
#include <time.h>
#include <sys/types.h>
#include <unistd.h>

void    usage(char *arg0)

{
    fprintf(stderr, "Usage: %s count\n", arg0);
    exit(EX_USAGE);
}


int     main(int argc, char *argv[])

{
    int     i;
    int     total_points;

    double  x;
    double  y;

    char    *end_ptr;

    if ( argc != 2 )
        usage(argv[0]);

    // Total number of points in the job, 10 in our examples.
    total_points = strtol(argv[1], &end_ptr, 10);

    // Make sure whole argument was a valid integer
    if ( *end_ptr != '\0' )
    {
        fprintf(stderr, "Error: Argument 1 (%s) is not an integer.\n",
            argv[1]);
        usage(argv[0]);
    }

    if ( total_points < 1 )
    {
        fputs("Total points must be > 0.\n", stderr);
        exit(EX_USAGE);
    }

    // First line of output is the total number of points
    // Not strictly necessary, but provides a level of error-detection
    // in case a file is truncated.
    printf("%d\n", total_points);

    // Initialize pseudo-random number sequence with a different value
    // each time
    srandom((unsigned long)getpid());

    // Compute X random points in the quadrant, and compute how many
    // are inside the circle
    for (i = 0; i < total_points; i++)
    {
        // random() and RAND_MAX are integers, so integer division will
        // occur unless we cast to a real type
        x = (double)random() / RAND_MAX;
        y = (double)random() / RAND_MAX;

        // Print to 20 sig-figs, slightly more than most CPUs can use
        printf("%0.20f %0.20f\n", x, y);
    }

    return EX_OK;
}
```

We can generate N input files with a simple script such as the following:

```
#!/bin/sh

printf "Compiling...\n"
cc -O -o gen-points gen-points.c
i=0
while [ $i -lt 10 ]; do
    printf "Generaring rand$i.txt...\n"
    ./gen-points 100 > rand$i.txt
    i=$((i+1))
done
```

The SLURM submission script:

```
#!/bin/sh -e

#SBATCH --output=calcpi-parallel.out
#SBATCH --error=calcpi-parallel.err
#SBATCH --array=1-10

./calcpi-file rand$SLURM_ARRAY_TASK_ID.txt
```

### Calcpi-file on an HTCondor Grid

To run a job like this on an HTCondor grid, the trick is to transfer the appropriate input file(s) to each of the execute hosts.

Since we're compiling the C program on the execute host, we use a shell script as the executable, and list the source file as an input file, along with the actual input file for the program. The shell script is executed on the execute host, where it first compiles the program and then runs it. Below is the job description file:

```
############################################################################
# Sample HTCondor submit description file.
#
# Use \ to continue an entry on the next line.
#
```

```
# You can query your jobs by command:
# condor_q

############################################################################
# Choose which universe you want your program is running with
# Available options are
#
# - standard:
#      Defaults to transfer executables and files.
#      Use when you are running your own script or program.
#
# - vanilla:
# - grid:
#      Explicitly enable file transfer mechanisms with
#      'transfer_executable', etc.
#      Use when you are using your own files and some installed on the
#      execute hosts.
#
# - parallel:
#      Explicitly enable file transfer mechanism. Used for MPI jobs.

universe = vanilla

# Macros (variables) to use in this submit description file
Process_count = 10

############################################################################
# Specify the executable filename.  This can be a binary file or a script.
# NOTE: The POVB execute hosts currently support 32-bit executables only.
# If compiling a program on the execute hosts, this script should compile
# and run the program.
#
# In calcpi-file-condor.sh, be sure to give the executable a different
# name for each process, since multiple processes could be on the same host.
# E.g. cc -O -o prog.$(Process) prog.c

executable = calcpi-file-condor.sh

# Pass the process index so that we can use different inputs for each
arguments = $(Process)

############################################################################
# Where the standard output and standard error from executables go.
# $(Process) is current job ID.

# If running calcpi-file under both PBS and HTCondor, use same output
# names here as in calcpi-file-run.pbs so that we can use the same
# script to tally all the outputs from any run.

output = calcpi-parallel.out-$(Process)
error = calcpi-parallel.err-$(Process)

############################################################################
# Logs for the job, produced by HTCondor.  This contains output from
# HTCondor, not from the executable.

log = calcpi-file-condor.log

############################################################################
# Custome job requirements
# HTCondor assumes job requirements from the host submitting job.
# IT DOES NOT DEFAULT TO ACCEPTING ANY ARCH OR OPSYS!!!
# For example, if the jobs is submitted from peregrine, target.arch is
# "X86_64" and target.opsys is "FREEBSD8", which do not match
# POVB execute hosts.
#
# You can query if your submitting host is accepted by command:
#  condor_q -analyze

# Memory requirements in megabytes
request_memory = 50

# Requirements for a binary compiled on CentOS 4 (POVB hosts):
# requirements = (target.arch == "INTEL") && (target.opsys == "LINUX")

# Requirements for a Unix shell script or Unix program compiled on the
# execute host:
requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
               ((target.opsys == "FREEBSD") || (target.opsys == "LINUX"))

# Requirements for a job utilizing software installed via FreeBSD ports:
# requirements = ((target.arch == "INTEL") || (target.arch == "X86_64")) && \
#     (target.opsys == "FREEBSD")

############################################################################
# Explicitly enable executable transfer mechanism for vanilla universe.

# true | false
transfer_executable = true

# yes | no | if_needed
should_transfer_files = if_needed

# All files to be transferred to the execute hosts in addition to the
# executable.  If compiling on the execute hosts, list the source file(s)
# here, and put the compile command in the executable script.
transfer_input_files = calcpi-file.c,rand$(Process).txt

# All files to be transferred back from the execute hosts in addition to
# those listed in "output" and "error".
# transfer_output_files = file1,file2,...

# on_exit | on_exit_or_evict
when_to_transfer_output = on_exit

############################################################################
# Specify how many jobs you would like to submit to the queue.

queue $(Process_count)
```

The executable, called calcpi-file-condor.sh, would look something like this:

```
#!/bin/sh

# Make sure the script was invoked with a filename as a command-line argument
if [ $# != 1 ]; then
    printf "Usage: $0 process-ID\n"
    exit 1
fi

# First command-line argument is the process index
process=$1
```

```
# Compile
cc -O -o calcpi-file.$process calcpi-file.c

# Pass the argument to this script on to the C program
./calcpi-file.$process rand$process.txt
```

### 33.2.3 Self-test

1. What is a Monte Carlo experiment?
2. Write a program and a submit script that uses multiple independent processes estimate the probability of rolling a 7 on two dice. This can, or course, be computed directly using simple statistics, but serves as a good exercise for the Monte Carlo method.

   Run the simulation using each of the following parameters. Run with each set of parameters several times, noting the consistency of the results and the computation time for each.

   - One process rolling the dice 100 times.
   - One process rolling the dice 100,000,000 times.
   - Ten processes rolling the dice 1,000,000,000 times each. Make sure each process uses a different random number sequence! (Hint: Use the job array index as a seed.)

# Chapter 34

# Programming for HPC

**Before You Begin**
Before reading this chapter, you should be familiar with basic Unix concepts (Chapter 3), the Unix shell (Section 3.3.3, redirection (Section 3.13), shell scripting (Chapter 4), and have some experience with computer programming.

## 34.1 Introduction

There are many computational problems that cannot be decomposed into completely independent subproblems.

*High Performance Computing* refers to a class of distributed parallel computing problems where the processes are not completely independent of each other, but must cooperate in order to solve a problem. The processes within a job are more tightly coupled, i.e. they exchange information with each other while running.

Because HPC processes are not independent of each other, the programming is more complex than HTC. Due to the complexity of HPC programming, it's usually worth the effort to search for previously written solutions. Most well-known mathematical functions used in science and engineering that can be solved in a distributed parallel fashion have already been implemented. Many complex tasks such as finite element analysis, fluid modeling, common engineering simulations, etc. have also been implemented in both open source and commercial software packages. Chances are that you won't need to reinvent the wheel in order to utilize HPC.

HPC jobs also may require a high-speed dedicated network to avoid a communication bottleneck. Hence, HPC models are generally restricted to clusters, and few will run effectively on a grid.

HPC problems do not scale as easily as HTC. Generally, more communication between processes means less scalability, but the reality is not so simple. Some HPC models cannot effectively utilize more than a dozen cores. Attempting to use more will increase communication overhead to the point that the job will take as long or longer than it does using fewer cores. On the other hand, some HPC models can scale to hundreds or thousands of cores. Each model is unique and there is no simple way to predict how a model will scale.

### 34.1.1 Self-test

1. Explain the basic difference between HTC and HPC.
2. What are the advantages of HTC?
3. What are the advantages of HPC?

## 34.2 Common Uses for HPC

One of the most common uses for HPC is modeling fluid flow. This general class of models has a wide range of applications, including weather forecasting, hydraulics, blood flow, etc. When modeling fluid flow, we cannot simply divide the volume into sectors and model flow independently in each sector, since some fluid flows across the boundaries between neighboring sectors. If one process models each sector, each process must exchange some information with the processes modeling neighboring sectors in order to account for all of the fluid in the system.

Another common use for HPC is *finite element analysis*. Finite element analysis is used to solve partial differential equations, often to model engineering simulations. There are many open source and commercial finite element software packages that utilize HPC.

## 34.3 Real HPC Examples

Weather Research and Forecasting Model (WRF), http://www.wrf-model.org/index.php, is an HPC software package commonly used in atmospheric science research.

Finite Volume Coastal Ocean Model (FVCOM), http://fvcom.smast.umassd.edu/FVCOM/index.html is an HPC software package used to predict ocean currents in a near-shore environment.

Finite element software packages are too numerous to list here. For a current listing, see the Wikipedia article on finite element analysis software http://en.wikipedia.org/wiki/List_of_finite_element_software_packages.

## 34.4 HPC Programming Languages and Libraries

HPC programming can be done using a wide variety of languages and tools, but most major HPC software is written in C, C++, or Fortran using *OpenMP*, *POSIX Threads*, or the *Message Passing Interface* (*MPI*) libraries.

C and Fortran are pure compiled languages, so programs run at the highest possible speed (often orders of magnitude faster than the same program written in an interpreted language). C++ also offers good performance, but C++ has a higher learning curve and best suited for well-trained, experienced programmers.

When developing libraries to be used from C, C++, and Fortran programs, C is generally the most trouble-free choice. C libraries are very easy to link into C++ and Fortran programs, and generally provide marginally better performance than C++ or Fortran. C++ and Fortran libraries can also be linked into programs written in other languages, but the process requires additional knowledge and effort.

### 34.4.1 Self-test

1. What are the most popular languages for HPC programming? Why?

## 34.5 Shared Memory Parallel Programming with OpenMP

OpenMP, short for Open Multiprocessing, is a set of tools embedded into many Fortran compilers since 1997, and C/C++ compilers since 2000. OpenMP allows the programmer to utilize multiple cores within a single computer, and in some cases, using extensions, multiple computers.

OpenMP allows for finer grained parallelism than embarrassingly parallel computing or distributed models such as MPI.

The latter two paradigms utilize multiple processes, potentially running on different computers, which require a great deal of overhead to create. Hence, individual processes need to be long-running so that the overhead of starting them doesn't use a significant portion of the total run time.

OpenMP, on the other hand, spawns lightweight threads within a single process, which requires very little overhead. Hence, OpenMP can effectively parallelize very short-running sections of code such as individual loops within a program.

OpenMP can benefit code that might only take a fraction of a second to run in serial, whereas HTC and MPI are only good for parallelizing processes that take far longer (usually minutes or hours).

The main limitation of OpenMP is that it only scales up to the number of cores available on a single computer (or other tightly-coupled architecture). If you want to achieve greater speedup, you'll need to look for ways to perform coarser scale parallelism with HTC or MPI.

OpenMP is implemented as a set of preprocessor directives, header files, and library functions.

To use OpenMP, a C or C++ program must include the header `omp.h` and use `#pragma` directives to indicate which code segments are to be parallelized and how.

Fortran programs must contain the line

```
use omp_lib
```

and use specially formatted comments beginning with

```
!$omp
```

To compile an OpenMP program with gcc, clang, or gfortran we must use the `-fopenmp` flag. Note that on FreeBSD, **cc** is **clang**, while on Linux, **cc** is **gcc**. Hence, **cc** and **c++** should work with the default compiler on both systems.

```
mypc: gcc -fopenmp myprog.c
mypc: g++ -fopenmp myprog.cc
mypc: clang -fopenmp myprog.c
mypc: clang++ -fopenmp myprog.cc
mypc: cc -fopenmp myprog.c
mypc: c++ -fopenmp myprog.cc
mypc: gfortran -fopenmp myprog.f90
```

At the time of this writing, clang does not include a Fortran compiler. Clang is intended to be binary compatible with GCC, however, so we can use gfortran alongside clang and clang++ as long as the compiler versions are compatible.

C and C++ programs must also contain

```
#include <omp.h>
```

After an OpenMP program has split into multiple threads, each thread can identify itself by calling omp_get_thread_num(), which returns a value between 0 and N-1 for a program running N threads.

Multithreaded programs can be *task parallel* (running different code for each thread) or *data parallel* (running the same code on different data for each thread).

### 34.5.1 OMP Parallel

Using a basic omp parallel pragma, we can execute the same code on multiple cores (more or less) simultaneously.

Simple parallel program:

```
/***************************************************************************
 *  Description:
 *      OpenMP parallel common code example
 *
 *  Arguments:
 *      None
 *
 *  Returns:
 *      Standard exit codes (see sysexits.h)
 *
 *  History:
 *  Date        Name        Modification
 *  2012-06-27  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <sysexits.h>
#include <omp.h>

int     main(int argc,char *argv[])

{
/* Execute the same code simultaneously on multiple cores */
#pragma omp parallel
    printf("Hello from thread %d!\n", omp_get_thread_num());

    return EX_OK;
}
```

```
!-------------------------------------------------------------------------
!   Program description:
!       OpenMP parallel common code example
!-------------------------------------------------------------------------

!-------------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2012-06-29  Jason Bacon Created
!-------------------------------------------------------------------------

! Main program body
program openmp_hello
    use omp_lib

    ! Disable implicit declarations (i-n rule)
    implicit none

    ! Note: No space allowed between ! and $
    !$omp parallel
    print *, 'Hello from thread ', omp_get_thread_num()
    !$omp end parallel
end program
```

We can control the number of threads in an OpenMP block using the `num_treads()` parameter or by setting OMP_NUM_THREADS in the environment.

```
/***************************************************************************
 *  Description:
 *      OpenMP parallel common code example
 *
 *  Arguments:
 *      None
 *
 *  Returns:
 *      Standard exit codes (see sysexits.h)
 *
 *  History:
 *  Date        Name        Modification
 *  2012-06-27  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <sysexits.h>
```

```
#include <omp.h>

int     main(int argc,char *argv[])

{
/* Execute the same code simultaneously on multiple cores */
#pragma omp parallel num_threads(2)
    printf("Hello from thread %d!\n", omp_get_thread_num());

    return EX_OK;
}
```

Otherwise, OpenMP will match the number of threads to the number of cores present on the system.

### 34.5.2 OMP Loops

One of the most commonly used features of OpenMP is loop parallelization.

---

**Note** As always, you should optimize the serial code before trying to parallelize it. Before resorting to parallelizing a loop, make sure that the loop is as efficient as possible. You may be able to move some code outside the loop to reduce its run time, greatly reduce the number of iterations, or eliminate the loop entirely with some careful thought. See Section 19.15 for a detailed discussion.

---

If a loop can be structured in such a way that each iteration is independent of previous iterations, then in theory, all of the iterations can be executed at the same time.

Of course, the number of iterations which can execute at once is limited by the number of cores available. For example, if a parallelizable loop iterates 100 times, but the computer running it only has four cores, then only four iterations will run at once. Still, this will speed up the program by nearly a factor of four. A small amount of overhead is incurred when splitting the execution path into multiple threads and again when merging them back together after the parallel segment.

Example of a parallel loop with OpenMP:

```
/***************************************************************************
 *  Description:
 *      OpenMP parallel loop example
 *
 *  Arguments:
 *      None
 *
 *  Returns:
 *      Standard exit codes (see sysexits.h)
 *
 *  History:
 *  Date        Name        Modification
 *  2011-10-06  Jason Bacon Begin
 ***************************************************************************/

#include <stdio.h>
#include <omp.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    int     c;

#pragma omp parallel for
```

```
    for (c=0; c < 8; ++c)
    {
        printf("Hello from thread %d, nthreads %d, c = %d\n",
            omp_get_thread_num(), omp_get_num_threads(), c);
    }
    return EX_OK;
}
```

```
!-----------------------------------------------------------------------
!   Program description:
!       OpenMP parallel loop example
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2012-06-29  Jason Bacon Created
!-----------------------------------------------------------------------

! Main program body
program openmp_hello
    use omp_lib

    ! Disable implicit declarations (i-n rule)
    implicit none
    integer :: c

    ! Note: No space allowed between ! and $
    !$omp parallel do
    do c = 1, 8
        print *, 'Hello from thread ', omp_get_thread_num(), &
            ' nthreads ', omp_get_num_threads(), ' c = ', c
    enddo
end program
```

**Caution**
When several iterations of a loop are run in parallel, we cannot predict the order in which they will complete. While they are in theory executed at the "same time", there really is no such thing as exactly the same time. Even if each iteration runs the exact same number of instructions, they may finish at slightly different times, and the order is unpredictable.
Hence, when using OpenMP, you must take special care to ensure that it does not matter to your program which iterations complete first.
Performing output to a common stream such as the standard output in a parallelized section of code is generally a bad idea, except for debugging purposes.

If anything within a loop depends on results computed during previous iterations, then the loop simply cannot be parallelized. It may be possible to redesign the loop so that each iteration is independent of the rest, but there are some cases where computations must be done serially.

### 34.5.3 Shared and Private Variables

When an OpenMP process splits into multiple threads, we may want some variables to exist independently in each thread and others to be shared by all threads.

For example, if a parallelized loop computes the sum of a list of numbers, the sum variable must be shared.

On the other hand, any variable that must contain a different value for each iteration (such as the OMP thread number) should be private.

### 34.5.4 Critical and Atomic Sections

When multiple threads modify a shared variable, there is a danger that the modifications could overlap, resulting in incorrect results.

**Caution** What appears as one statement in the source code may actually be a sequence of several instructions of machine code.

For example, when assigning a new value to sum in the example below, a typical CPU must actually perform multiple steps. The following would represent the sequence of machine instructions on many CPUs:

1. Load the value of sum from memory into the CPU
2. Load the value of c from memory into the CPU
3. Add sum + c in the CPU
4. Store the result back to memory variable sum

This is commonly called a *read-modify-write* sequence.

```
/***************************************************************************
 *  Description:
 *      OpenMP private and shared variables example
 *      Run time is around 4 seconds for MAX = 200000000 on an i5
 ***************************************************************************/

#include <stdio.h>
#include <sysexits.h>

#define MAX         200000000

int     main(int argc,char *argv[])

{
    unsigned long   sum;

    sum=0;
    /* compute sum of squares */
#pragma omp parallel for shared(sum)
    for (unsigned long c = 1; c <= MAX; ++c)
    {
#pragma omp atomic
        sum += c;
    }
    printf("%lu\n", sum);
    return EX_OK;
}
```

```
!-----------------------------------------------------------------------
!   Program description:
!       OpenMP parallel loop example
!-----------------------------------------------------------------------

! Main program body
program openmp_hello
```

```
    use omp_lib

    ! Disable implicit declarations (i-n rule)
    implicit none
    integer*8 :: c, sum, max

    ! Note: No space allowed between ! and $
    !$omp parallel do private(c_squared) shared(sum)
    sum = 0
    max = 200000000
    do c = 1, max
        !$omp atomic
        sum = sum + c
    enddo
    print *, sum
end program
```

Suppose there are two threads, numbered 0 and 1.

Also suppose that thread 0 begins the read-modify-write sequence when sum = 1 and c = 2. When thread 0 finishes the sequence, sum should contain 1 + 2 = 3. We will assume that this actually happens as expected.

Now suppose thread 1 has a value c = 3 and begins the read-modify-write sequence immediately after thread 0 finishes writing 3 to sum. Thread 1 will then add sum=3 + c=3 and store the correct value of 6 in sum.

However, if thread 1 begins *before* thread 0 stores the 3 into sum, the following will occur:

1. Thread 1 will load the old value 1 from sum, not the updated value of 3.
2. Thread 0 will store 3 into sum as expected, but thread 1 will never load this value because it as already performed its load instruction.
3. Thread 1 will then add 1 + 3 and store the incorrect value 4 in sum. If the store operation in thread 1 happens after the store in thread 0, this 4 will overwrite the 3 placed there by thread 0. If the store in thread 1 happens before the store in thread 0, then the 3 from thread 0 will overwrite the 4 from thread 1.

We cannot predict exactly when each thread will perform each of its operations. This is determined by the operating system based on many variables beyond our control that affect CPU scheduling. After both threads 0 and 1 have completed, sum could contain either 3, 4, or 6. This is called a *race condition*, because it depends on which thread finishes each operation first.

Unfortunately, the programmer must take responsibility for pointing this out to the compiler by declaring statements as *critical* or *atomic*.

A statement or block marked critical can only be entered (begun) by one thread at a time. Once a thread has begun executing a critical section, no other thread can begin executing it until the current thread has completed it.

In an atomic section, only memory update portions of the code are considered critical by the compiler. This portion would include the read-modify-write sequence for sum in the example above. *Reading* the value of c, on the other hand, would not be critical, because c is not shared by multiple threads. Each thread has a private copy of c with a different value than other threads. Using atomic is usually sufficient, and sometimes faster, since it may allow the compiler to parallelize parts of the machine code that cannot be separated from the critical parts at the source code level.

The program above has an atomic section that is executed 200,000,000 times. The inability to execute these sequences in parallel can have a major impact on performance. We can get around the atomic section bottleneck by computing a private partial sum for each thread. Updating this partial sum need not be done atomically, since it is not shared by multiple threads. This replaces 200,000,000 atomic operations with operations that can be done in parallel. We must then atomically add the private sums to compute the overall sum, but the number of atomic additions here is only the number of threads, which is very small (e.g. probably 4 on a 4-core computer), so the performance hit is negligible. The end result in this example is a program that runs more than twice as fast.

```
/***************************************************************************
 *  Description:
 *      OpenMP private and shared variables example
 *      Run time is 1.9 seconds for MAX = 200000000 on an i5
 *      This is about twice as fast as the same program using a shared sum
 *      in the main loop
 ***************************************************************************/

#include <stdio.h>
#include <sysexits.h>
#include <omp.h>

#define MAX_CORES   256     // More than any current computer
#define MAX         200000000

int     main(int argc,char *argv[])

{
    unsigned long   sum, private_sums[MAX_CORES];

    for (int c = 0; c < MAX_CORES; ++c)
        private_sums[c] = 0;

    /*
     *  Here we use an array of sums, one for each thread, so that
     *  adding to the sum need not be atomic.  Hence, there is no
     *  bottleneck and we achieve much better speedup than when using
     *  a shared sum variable, which must be updated atomically.
     */

    #pragma omp parallel for
    for (unsigned long c = 1; c <= MAX; ++c)
    {
        private_sums[omp_get_thread_num()] += c;
    }

    /*
     *  Now we atomically add the private sums to find the total sum.
     *  The number of atomic operations here is only the number of
     *  threads, typically much smaller than MAX, so the bottleneck
     *  has minimal impact on run time.
     */

    sum = 0;
    #pragma omp parallel for shared(sum)
    for (int c = 0; c < omp_get_num_threads(); ++c)
    {
        #pragma omp atomic
        sum += private_sums[c];
    }

    printf("sum = %lu\n", sum);
    return EX_OK;
}
```

### 34.5.5 Self-test

1. What are the advantages of OpenMP over distributed parallel systems such as MPI?

2. Write an OpenMP program that prints the square of every number from 1 to 1000. Print the thread number alongside each square.

## 34.6 Shared Memory Parallelism with POSIX Threads

The POSIX threads library, or *pthreads*, is a standardized interface for using lightweight threads in C programs on POSIX (Portable Operating System Interface based on Unix) platforms.

Note that C libraries can easily be used in programs written in other compiled languages, so pthreads can be used directly in C++ and Fortran programs. There are also well-developed interfaces for many other languages.

The pthreads system addresses the same basic needs as OpenMP, but using a different approach. Threads are created manually using `pthread_create()` in much the same way as we would use `fork()` to create a new process.

```c
/*
 *  Example from Pthreads tutorial at:
 *  https://computing.llnl.gov/tutorials/pthreads/
 */

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   /*
    *   Code here will be run simultaneously by multiple threads.
    */

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

## 34.7 Message Passing Interface (MPI)

Avoid it if you can. Embrace it wholeheartedly if you must...

The Message Passing Interface (MPI) is a standard API (application program interface) and set of tools for building and running distributed parallel programs on almost any parallel computing architecture.

MPI includes libraries of subprograms that make it as simple as possible to start up a group of cooperating processes and pass messages between the processes. It also includes tools for running, monitoring, and debugging MPI jobs.

There are many implementations of MPI, but the emerging standard is *Open MPI*, which evolved from the best features of several earlier open source implementations. Open MPI is free and open source, so you can rest assured that projects you develop with open MPI will never be orphaned.

It is important to note that MPI programs *do not require a cluster to run*. MPI is also effective in taking advantage of multiple cores in a single computer. MPI programs can even be run on a single-core computer for testing purposes, although this won't, of course, run any faster than a serial program on the same machine, and may even take slightly longer. Other parallel programming paradigms might be easier to use or faster than MPI on a shared memory architecture, but if you may want to run across multiple computers, programming in MPI is a good investment of time.

The bottom line is that you can use the same MPI programs to utilize multiple cores on a single computer, or a cluster of any size suitable for the job. This makes MPI the most portable applications programming interface (API) for parallel programs. You can also develop and test MPI code on your own PC, and later run it on a larger cluster. Some users may find this approach preferable, since development on a local PC with their preferred tools can be faster and more comfortable, and it reduces the risk of impacting other cluster users with untested code.

### 34.7.1 Self-test

1. What does MPI stand for?
2. What tools does the MPI system include?
3. What type of parallel computer system does MPI require?

## 34.8 MPI vs Other Parallel Paradigms

There are a number of ways to decompose many problems for parallel execution.

If a problem can be decomposed into independent parts and run in an embarrassingly parallel fashion, this is usually the best route, since it is the easiest to program and the most scalable.

If the processes within a parallel job must communicate during execution, there are a variety of options, including MPI, shared-memory parallelism, and specialty hardware architectures (supercomputers) such as Single Instruction Multiple Data (SIMD) machines.

Which architecture and model will provide the best performance is dependent on the algorithms your software needs to use.

One advantage of MPI, however, is portability. An MPI program is capable of utilizing virtually any architecture with multiple processors. MPI can utilize the multiple cores on a single PC, the multiple nodes in a cluster, and in some cases (if communication needs are light) MPI programs could even run on a grid.

Shared-memory parallelism, discussed in Section 34.5, *might* provide better performance when utilizing multiple cores within a single PC. However, this is not a general rule. Again, it depends on the algorithms being used. In addition, shared-memory parallelism does not scale well, due to the fact that the cores contend for the same shared memory banks and other hardware resources. A PC with 48 cores may not provide the performance boost you were hoping for.

Since parallel programming is a complex and time-consuming endeavor, using a system such as MPI that will allow the code to run on the widest possible variety of architectures has clear advantages. Even if shared-memory parallelism offers better performance, it may still be preferable to use MPI when you consider the value of programmer time and the ability to utilize more than one computer. The performance gains of a shared-memory program using a small number of cores may not be important enough to warrant the extra programming time and effort.

### 34.8.1 Self-test

1. When should embarrassingly parallel computing be used instead of parallel programming?
2. What are the general rules that indicate the best parallel programming paradigm for a given problem?
3. What are the limitations of shared-memory parallelism?

## 34.9 Structure of an MPI Job

An MPI job consists of two or more cooperating processes which may be running on the same computer or on different computers.

MPI programs can be compiled with a standard compiler and the appropriate compile and link flags. However, MPI systems provide simple wrappers that eliminate the need to include the MPI flags.

For example, an MPI program can be run using 4 cores on a stand-alone computer with:

```
mypc: mpicc my-mpi-prog.c -o my-mpi-prog
mypc: mpirun -n 4 ./my-mpi-prog
```

One process in the job is generally designated as the root process. The root process is not required to do anything special, but it typically plays a different role than the rest. Often the root process is responsible for things like partitioning a matrix and distributing it to the other processes, and then gathering the results from the other processes.

Usually, mpirun starts up N identical processes, which then determine for themselves which process they are (by calling an MPI function that returns a different rank value to each process) and then follow different paths depending on the result.

```
/* Initialize data for the MPI functions */
if ( MPI_Init(&argc, &argv) != MPI_SUCCESS )
{
    fputs("MPI_Init failed.\n", stderr);
    exit(EX_UNAVAILABLE);
}

if ( MPI_Comm_rank(MPI_COMM_WORLD, &my_rank) != MPI_SUCCESS )
{
    fputs("MPI_Comm_rank failed.\n", stderr);
    exit(EX_UNAVAILABLE);
}

/*
 *  For this job, the process with rank 0 will assume the role of
 *  the "root" process, which will run different code than the
 *  other processes.
 */
if (my_rank == ROOT_RANK)
{
    // Root process code
}
else
{
    // Non-root process code
}
```

**Note** All processes in this MPI job contain code that will never be executed, but this is not considered a problem, since the size of code is generally dwarfed by the size of the data. Hence, it is not usually worth the effort to create separate programs for different processes within an MPI job.

## 34.10 A Simple MPI Program

MPI programming is complex and not necessary for all cluster users. Many cluster users will only run HTC jobs, which do not require communication between processes. The goal of this example is to provide a very simple introduction to MPI programming for those who are interested. We assume some familiarity with C or Fortran.

Users interested in pursuing MPI programming are encouraged to consult a book on the subject. Many good resources are cited on the HPC website at http://www4.uwm.edu/hpc/related_resources/.

C version:

```
/*
 *  Program description:
 *
 *  MPI Example
 *
 *  A typical MPI job will start N processes, all running the same
 *  program.  This is known as the Single-Program Multiple-Data (SPMD)
 *  model.
 *
 *  This program demonstrates how the various processes
 *  distinguish themselves from the rest using MPI library functions.
 *
 *  Return values of all MPI functions are checked for good practice,
 *  although some MPI functions will by default terminate the process without
 *  returning if an error is encountered.  This behavior can be changed, so
 *  including error checks for every MPI call is a good idea.
 */

/*
 *  Modification history:
 *  Date        Name        Modification
 *  2011-08-24  Jason Bacon Begin
 */

#include <stdio.h>      /* fputs(), printf(), etc. */
#include <stdlib.h>     /* exit() */
#include <sysexits.h>   /* EX_ exit constants */
#include <sys/param.h>  /* MAXHOSTNAMELEN */
#include <unistd.h>     /* gethostname() */
#include <mpi.h>        /* MPI functions and constants */

#define ROOT_RANK       0
#define MESSAGE_MAX_LEN 128
#define TAG             0

int     main(int argc, char *argv[])
{
    int             total_processes;
    int             my_rank;
    int             rank;
    MPI_Status      status;
    char            message[MESSAGE_MAX_LEN + 1];
    char            hostname[MAXHOSTNAMELEN + 1];

    /* Get name of node running this process */
    if ( gethostname(hostname, MAXHOSTNAMELEN) != 0 )
    {
        fputs("gethostname() failed.\n", stderr);
        exit(EX_OSERR);
    }
    /* Initialize data for the MPI functions */
    if ( MPI_Init(&argc, &argv) != MPI_SUCCESS )
    {
        fputs("MPI_Init failed.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    /* Find out how many processes are in this MPI job */
    if ( MPI_Comm_size(MPI_COMM_WORLD, &total_processes) != MPI_SUCCESS )
    {
        fputs("MPI_Comm_size failed.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    /*
     *  Each process withing the job has a unique integer "rank".
     *  This is how each process determines its role within the job.
     */
    if ( MPI_Comm_rank(MPI_COMM_WORLD, &my_rank) != MPI_SUCCESS )
    {
        fputs("MPI_Comm_rank failed.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    /*
     *  For this job, the process with rank 0 will assume the role of
     *  the "root" process, which will run different code than the
     *  other processes.
     */
    if (my_rank == ROOT_RANK)
    {
        printf("We have %d processors\n", total_processes);

        /* Send a message to all non-root processes */
        for (rank = 1; rank < total_processes; ++rank)
        {
            snprintf(message, MESSAGE_MAX_LEN, "Process %d, where are you? ", rank);
            if ( MPI_Send(message, MESSAGE_MAX_LEN, MPI_CHAR, rank, TAG,
                        MPI_COMM_WORLD) != MPI_SUCCESS )
            {
                fputs("MPI_Comm_rank failed.\n", stderr);
                exit(EX_UNAVAILABLE);
            }
        }

        /* Read the response from all non-root processes */
        for (rank = 1; rank < total_processes; ++rank)
        {
            if ( MPI_Recv(message, MESSAGE_MAX_LEN, MPI_CHAR, rank, TAG,
                        MPI_COMM_WORLD, &status) != MPI_SUCCESS )
            {
                fputs("MPI_Comm_rank failed.\n", stderr);
                exit(EX_UNAVAILABLE);
            }
            printf("%s\n", message);
        }
    }
    else
    {
        /* Wait for message from root process */
        if ( MPI_Recv(message, MESSAGE_MAX_LEN, MPI_CHAR, ROOT_RANK,
                TAG, MPI_COMM_WORLD, &status) != MPI_SUCCESS )
        {
            fputs("MPI_Comm_rank failed.\n", stderr);
            exit(EX_UNAVAILABLE);
        }
        printf("Process %d received message: %s\n", my_rank, message);

        /* Send response */
        snprintf(message, MESSAGE_MAX_LEN, "Process %d is on %s", my_rank, hostname);
        if ( MPI_Send(message, MESSAGE_MAX_LEN, MPI_CHAR, ROOT_RANK, TAG, MPI_COMM_WORLD)  ↩
            != MPI_SUCCESS )
        {
            fputs("MPI_Comm_rank failed.\n", stderr);
            exit(EX_UNAVAILABLE);
        }
    }

    /*
     *  All MPI processes must execute MPI finalize to synchronize
     *  the job before they exit.
     */

    if ( MPI_Finalize() != MPI_SUCCESS )
    {
        fputs("MPI_Finalize failed.\n", stderr);
        exit(EX_UNAVAILABLE);
    }
    return EX_OK;
}
```

Fortran version:

```
!-----------------------------------------------------------------------
!   Program description:
!
!   MPI Example
!
!   A typical MPI job will start N processes, all running the same
!   program.  This is known as the Single-Program Multiple-Data (SPMD)
!   model.
!
!   This program demonstrates how the various processes
!   distinguish themselves from the rest using MPI library functions.
!-----------------------------------------------------------------------

!-----------------------------------------------------------------------
!   Modification history:
!   Date        Name        Modification
!   2011-08-24  Jason Bacon Begin
!-----------------------------------------------------------------------

module constants
    ! Global Constants
    double precision, parameter :: &
        PI = 3.1415926535897932d0, &
        E = 2.7182818284590452d0, &
        TOLERANCE = 0.00000000001d0, &  ! For numerical methods
        AVOGADRO = 6.0221415d23         ! Not known to more digits than this
    integer, parameter :: &
        MESSAGE_MAX_LEN = 128, &
        HOSTNAME_MAX_LEN = 128, &
        TAG = 0, &
        ROOT_RANK = 0
end module constants

! Main program body
program mpi_hello
    use constants           ! Constants defined above
    use ISO_FORTRAN_ENV     ! INPUT_UNIT, OUTPUT_UNIT, ERROR_UNIT, etc.

    ! Disable implicit declarations (i-n rule)
    implicit none

    include 'mpif.h'        ! MPI constants

    ! Variable defintions
    character(MESSAGE_MAX_LEN) :: message       ! MPI message buffer
    character(HOSTNAME_MAX_LEN) :: hostname     ! Name of node
    integer :: total_processes, my_rank, rank, count, ierr, &
               message_length = MESSAGE_MAX_LEN
    integer :: status(MPI_STATUS_SIZE)

    ! Get name of node running this process
    call hostnm(hostname)

    ! Initialize data for the MPI functions
    call mpi_init(ierr)
    if ( ierr /= MPI_SUCCESS ) stop 'mpi_init failed.'

    ! Find out how many processes are in this MPI job
    call mpi_comm_size(MPI_COMM_WORLD, total_processes, ierr)
    if ( ierr /= MPI_SUCCESS ) stop 'mpi_comm_size failed.'

    ! Each process withing the job has a unique integer "rank".
    ! This is how each process determines its role within the job.
    call mpi_comm_rank(MPI_COMM_WORLD, my_rank, ierr)
    if ( ierr /= MPI_SUCCESS ) stop 'mpi_comm_rank failed.'

    ! For this job, the process with rank ROOT_RANK will assume the role of
    ! the "root" process, which will run different code than the
    ! other processes.
    if ( my_rank == ROOT_RANK ) then
        ! Only root process runs this clause

        ! Do this in root so it only prints once
        print '(a,i0,a)', 'We have ', total_processes, ' processes.'

        ! Debug code
        ! print '(a,a,a)', 'Root processing running on ', trim(hostname), '.'

        do rank = 1, total_processes-1
            write (message, '(a,i0,a)') 'Process ', rank, ', where are you?'

            ! Debug code
            ! print '(a,a,a,i0,a)', 'Sending ', trim(message), ' to process ', &
            !    rank, '.'

            ! It's stupid to send a padded string, but it's complicated
            ! for mpi_recv() to receive a message of unknown length
            ! A smarter program would save network bandwidth and time by using
            ! len_trim(message) instead of MESSAGE_MAX_LEN.
            call mpi_send(message, MESSAGE_MAX_LEN, MPI_CHARACTER, rank, &
                TAG, MPI_COMM_WORLD, status, ierr)
            if ( ierr /= MPI_SUCCESS ) stop 'mpi_send failed.'
        enddo

        do count = 1, total_processes-1
            ! Accept message from slave processes in any rank order
            ! by using MPI_ANY_SOURCE for rank in recv call
            call mpi_recv(message, MESSAGE_MAX_LEN, MPI_CHARACTER, &
                MPI_ANY_SOURCE, MPI_ANY_TAG, &
                MPI_COMM_WORLD, status, ierr)
            if ( ierr /= MPI_SUCCESS ) stop 'mpi_recv failed.'
            print *, 'Root received response: ', trim(message)
        enddo
    else
        ! All slave processes run this section

        ! Debug code
        ! print '(a,i0,a,a, a)', 'Process ', my_rank, ' running on ', &
        !    trim(hostname), '.'

        ! Wait for message from root
        call mpi_recv(message, MESSAGE_MAX_LEN, MPI_CHARACTER, ROOT_RANK, &
            TAG, MPI_COMM_WORLD, status, ierr)
        if ( ierr /= MPI_SUCCESS ) stop 'mpi_recv failed.'
        print '(a,i0,a,a)', 'Process ', my_rank, ' received: ', trim(message)

        ! Respond to message from root
        write (message, '(a,i0,a,a,a)') 'Process ', my_rank,' is on ', &
            trim(hostname), '.'
        call mpi_send(message, len(message), MPI_CHARACTER, ROOT_RANK, &
            TAG, MPI_COMM_WORLD, status, ierr)
        if ( ierr /= MPI_SUCCESS ) stop 'mpi_send failed.'
    endif

    ! All MPI processes must execute MPI finalize to synchronize
    ! the job before they exit.
    call mpi_finalize(ierr)
    if ( ierr /= MPI_SUCCESS ) stop 'mpi_finalize failed.'
end program
```

In a scheduled environment, MPI jobs are submitted like batch serial jobs. The scheduler is informed about resource requirements (cores, memory) but does not dispatch all the processes. The scheduler dispatches a single **mpirun** command and the **mpirun** command then creates all the processes to a list of nodes provided by the scheduler.

SLURM submit script:

```
#!/bin/sh -e

# Number of cores
#SBATCH --ntasks=8

mpirun ./mpi-hello
```

Programs should be compiled in the same environment in which they will run, i.e. on a compute node, under the scheduler. This will ensure that they find the same tools and libraries at run time as they did at compile time. The best way to achieve this is by using a submit script to compile:

SLURM build script:

```
#!/bin/sh -e

# Number of cores
```

```
#SBATCH --ntasks=1

mpicc -o mpi-hello mpi-hello.c
```

PBS submit script:

```
#!/bin/sh

# Job name
#PBS -N MPI-Hello

# Number of cores
#PBS -l procs=8

##########################################################################
# Shell commands
##########################################################################

# Torque starts from the home directory on each node, so we must manually
# cd to the working directory where the hello binary is located.
cd $PBS_O_WORKDIR
mpirun ./mpi-hello
```

LSF submit script:

```
#!/usr/bin/env bash

# Job name
#BSUB -J MPI-Hello

# Number of cores
#BSUB -n 8

##########################################################################
# Shell commands
##########################################################################

# LSF requires the use of wrapper scripts rather than using mpirun directly
openmpi_wrapper ./mpi-hello
```

### 34.10.1 Self-test

1. Write an MPI version of `calcpi.c` from Chapter 33. For simplicity, use a constant for the number of random points generated and use the process rank for the srandom() seed.
2. Is MPI the best solution for estimating PI? Why or why not?
3. Write an MPI matrix addition routine. The root process should distribute a row of each source matrix to each of the worker processes until all the rows are added.
4. Is this a good fit for MPI? Why or why not?

## 34.11 Best Practices with MPI

The MPI system attempts to clean up failed jobs by terminating all processes in the event that any one of the processes fails. However, MPI's automatic cleanup can take time, and cannot always detect failures. This sometimes leads to orphaned processes remaining in the system unbeknownst to the scheduler, which can cause problems for other jobs.

Hence, it is each programmer's responsibility to make sure that their MPI programs do not leave orphaned processes hanging around on the cluster. How this is accomplished depends on the particular program, however every MPI program should follow these general rules:

1. Check the exit status of *every* MPI function/subroutine call and every other statement in the program that could fail in a way that would prevent the program from running successfully. Some other examples include memory allocations and file opens, reads, writes. These are only examples, however. It is the programmer's responsibility to examine every line of code and check for possible failures. *There should be no exceptions to this rule*.
2. Whenever a statement fails, the program should detect the failure and take appropriate action. If you cannot determine a course of action that would allow the program to continue, then simply perform any necessary cleanup work and terminate the process immediately. MPI programs that do not self-terminate may leave orphaned processes running on a cluster that interfere with other users' jobs.

### 34.11.1 Self-test

1. How can an MPI program ensure that it doesn't leave orphaned processes running?

## 34.12 Higher Level MPI Features

A parallel program may not provide much benefit if only the computations are done in parallel. Disk I/O and message passing may also need be parallelized in order to avoid bottlenecks that will limit performance to serial speed.

### 34.12.1 Parallel Message Passing

Suppose we have 100 processes in an MPI job that all need the same data in order to begin their calculations. We could simply loop through them and send the data as follows:

```
for (rank = 1; rank < 100; ++rank)
{
    if ( MPI_Send(data, len, MPI_CHAR, rank, TAG, MPI_COMM_WORLD) !=
         MPI_SUCCESS )
    {
        ...
    }
}
```

The problem is, this is a serial operation that sends one message at a time, while many pathways through the network switch may be left idle. A typical network switch used in a cluster is capable of transmitting many messages between disjoint node pairs at the same time. For example, node 1 can send a message to node 5 at the same time node 4 sends a message to node 10.

If it ends up taking longer to distribute data to the processes than it does to do the computations on that data, then it's time to look for a different strategy.

A simple strategy to better utilize the network hardware might work as follows:

1. Root process transmits data to process 1.
2. Root process and process 1 transmit to processes 2 and 3 at the same time.
3. Root process and processes 1, 2, and 3 can all transmit to processes 4, 5, 6, and 7 at the same time.
4. ...and so on.

If the job uses a large number of processes, the limits of the network switch may be reached, but that's OK. If the switch is saturated, it will simply transmit the data as fast as it can. There is rarely any negative impact from a saturated network switch, other than increased response times for other processes, but this is why clusters have dedicated networks.

While this broadcast strategy is simple in concept, it can be tricky to program. Real world strategies take into account various different network architectures in an attempt to optimize throughput for specific hardware.

Fortunately, MPI offers a number of high-level routines such as MPI_Bcast(), MPI_Scatter(), and MPI_Gather(), which will provide good performance on most hardware, and certainly better than the serial loop above.

There may be cases where using routines designed for specific network switches can offer significantly better performance. However, this may mean sacrificing portability.

A big part of becoming a proficient MPI programmer is simply learning what MPI has to offer and choosing the right routines for your needs.

### 34.12.2 Self-test

1. What is the easiest way to ensure reasonably good performance from your MPI programs?
2. What problems are associated with overloading a cluster's dedicated network?
3. Write an MPI program that solves a linear system of equations using an open source distributed parallel linear systems library.

## 34.13 Process Distribution

Another thing we need to consider on modern hardware is that most cluster nodes have multiple cores. Hence, messages are often passed between two processes running on the same compute node. Such messages do not pass through the cluster network: There is no reason for them to leave the compute node and come back. Instead, they are simply placed in an operating system memory buffer by the sending process, and then read by the recipient process.

In general, local connections like this are faster than even the fastest of networks. However, memory is a shared resource and can become a bottleneck if too many processes on the same node are passing many messages.

For some applications, you may find that you get better performance by limiting the number of processes per node, and hence balancing the message passing load between local memory and the network. Jobs that perform a lot of disk I/O may also benefit from using fewer processes per node. Most schedulers allow you to specify not just how many processes to use, but also how many to place on each node.

The best way to determine the optimal distribution of processes is by experimenting. You might try 1 per node, 4 per node, and 8 per node and compare the run times. If 8 per node turns out to be the slowest, then try something between 1 and 4. If 4 turns out to be the fastest, then try 5 or 6. Results will vary on different clusters, so you'll have to repeat the experiment on each cluster. If you plan to run many jobs over the course of weeks or months, spending a day or two finding the optimal distribution could lead to huge time savings overall.

### 34.13.1 Self-test

1. When is it likely that limiting the number of MPI processes per node will improve performance? Why?

## 34.14 Parallel Disk I/O

Disk I/O is another potential bottleneck that can kill the performance of a parallel program. Disk access is about 1,000,000 times slower than memory access in the worst case (completely random), and about 10 or 20 times slower in the best case (completely sequential).

I/O is generally avoided even in serial programs, but the problem can be far worse for parallel programs. Imagine 300 processes in a parallel job competing for access to the same files.

Using local disks on the compute nodes *might* improve performance, especially if the processes are spread out across as many nodes as possible. However, this will require distributing the data beforehand. Another issue is that modern nodes have multiple cores, and there may be other processes competing for disk even though you chose to spread out your own job. You could request exclusive access to nodes if I/O is really an issue, but leaving 7 cores idle on 300 8-core machines is not good utilization of resources.

Modern clusters usually have a high-speed shared file system utilizing Redundant Arrays of Parallel Disks (RAIDs). In theory, a group of 20 disks grouped together in a RAID can be read 20 times faster than a single disk of the same type. Performance gains for write access are not as simple, but can also be significant.

Depending on how much I/O your jobs must do, you may be better off using a high-speed shared RAID, or you may be better off distributing data to local disks before computations begin. Only experimentation with each job on each cluster will reveal the optimal strategy.

# Part V

# Systems Management

# Chapter 35

# Systems Management

## 35.1 Guiding Principals

Decisions should be based on objective goals, such as

- Improving performance
- Improving reliability (which should also be viewed as part of performance)
- Reducing maintenance cost
- Making all hardware expendable. What end-users ultimately need is access to the programs that do what they need. If a computer they are using to run those programs becomes inaccessible for any reason, it should be easy for them to use another one. Package managers, discussed in Chapter 39, help us achieve this sort of independence. All too often, however, people end up in a panic, unable to get work done, because of a hardware failure. This situation is almost always a symptom of poor systems management.

Apply the KISS principal (Keep It Simple, Stupid) to avoid wasted time and effort on unnecessary complexity.

Unfortunately, many IT professionals are driven by ego or other irrational motives and decisions are based on emotional objectives such as

- Using their favorite tool (solutions looking for problems)
- Favoring the complex solution to make themselves look smart

Top-notch systems managers aim to make everything easily reproducible. All hardware then becomes expendable, because the functionality it provides can be quickly replicated on another machine. This means automating configurations using shell scripts or other tools, and keeping back-ups of important data. Using proprietary tools that may not be around in the future can be a grave mistake. Make sure your automation and backup tools will be readily available as long as you need them.

It's normal to struggle with something the first time you do it. It's incompetent to struggle with it the second time.

Top systems managers also understand how their systems work in detail, so when something does go wrong, they know exactly what to do and can fix it instantly.

Apply the principles of the engineering life cycle, discussed in Section 13.8. Start by throwing out all assumptions about design and implementation of IT solutions, such as which language or operating system will be used. First examine the specification: What does the end-user need to do? Will it be done once, twice, or many times? Then consider ALL viable alternatives from counting on your fingers, to scribbling on paper, to using a supercomputer. Which is the cleanest, simplest, most cost-effective way to enable it?

## 35.2 Attachment is the Cause of All Suffering

If you're averse to reinstalling your OS from scratch, get over it.

Trying to keep an existing installation running too long is a bad idea for a variety of reasons.

- All hardware fails eventually. Therefore, you should always have your important files backed up in another location and should always be prepared to rebuild your setup on a new disk and restore from backup.
- File systems more than a couple years old can suffer from "bit rot", where the disk is still functional, but some of the bits have faded to the point of being uncertain. For this reason, it's a bad idea to perform OS upgrade after OS upgrade. Instead, the disk should be wiped clean at least once every few years and everything reinstalled from scratch, to "freshen the bits".
- Practice makes perfect. If you avoid doing fresh installs, you will lack the skills needed when it becomes necessary and struggle to recover from hardware failures. If, on the other hand, you do fresh installs frequently, a disk or system failure will not be a big deal to you. Replace the failed hardware and be back in action in an hour or two in most cases.

# Chapter 36

# Popular Unix Platforms in Science

## 36.1 FreeBSD in Science

The computer field is full of annoying evangelists pushing their favorite operating systems and languages without regard for individual users' needs. I am not one of them. The purpose of this section is not to promote FreeBSD, but to make people aware of its potential advantages. If these advantages make FreeBSD a good fit for your needs, then you should use it. If other factors make a different operating system more suitable for you, then you should use that. My only goal is to help people make informed choices. Most people don't. They are more likely to blindly follow the crown or advice from friends and colleagues, than to properly investigate the alternatives and make a rational choice. This section is here to inform you, not persuade you. I hope you find the information useful.

FreeBSD is a popular platform for scientific computing, for good reasons, though many people are unaware of this. It has a number of features that make it an outstanding platform for scientists.

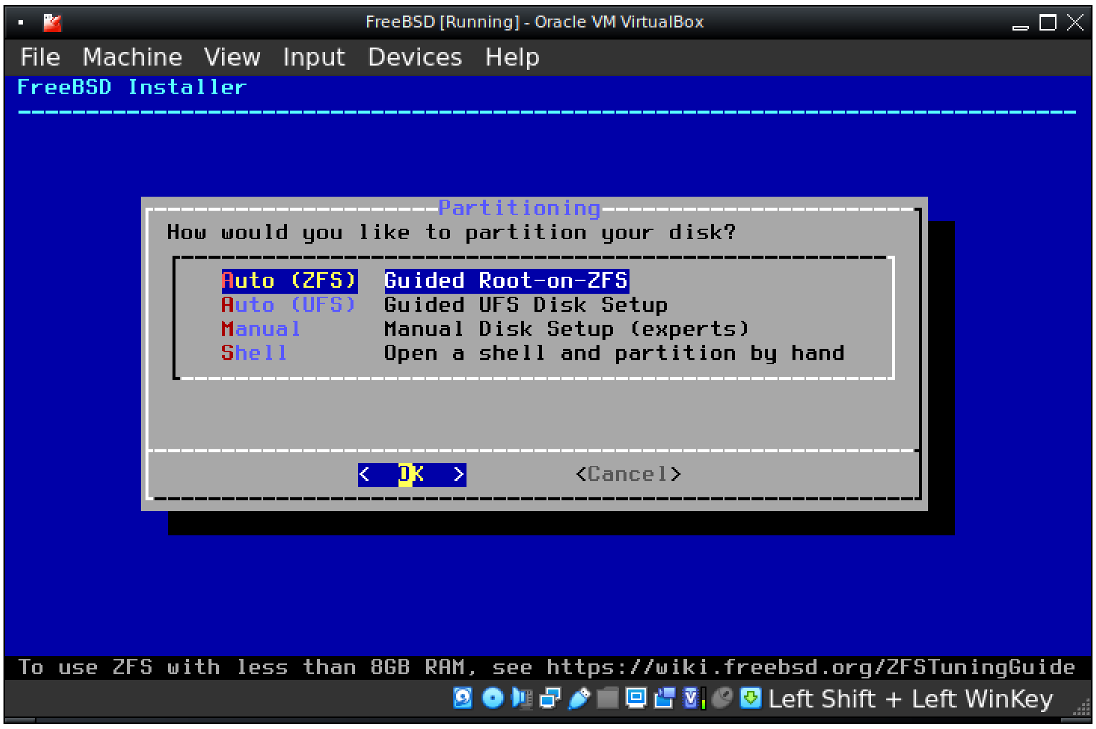
A minimal FreeBSD installation can be completed in under 5 minutes. This may be all you need for a server installation. Then just configure your security measures, install the packages you need, and get to work.

For desktop PCs, laptops, and workstations, you can install any of the popular desktop environments using the *desktop-installer* package in the FreeBSD ports system (https://github.com/outpaddling/desktop-installer).

```
shell-prompt: pkg install desktop-installer
shell-prompt: desktop-installer
```

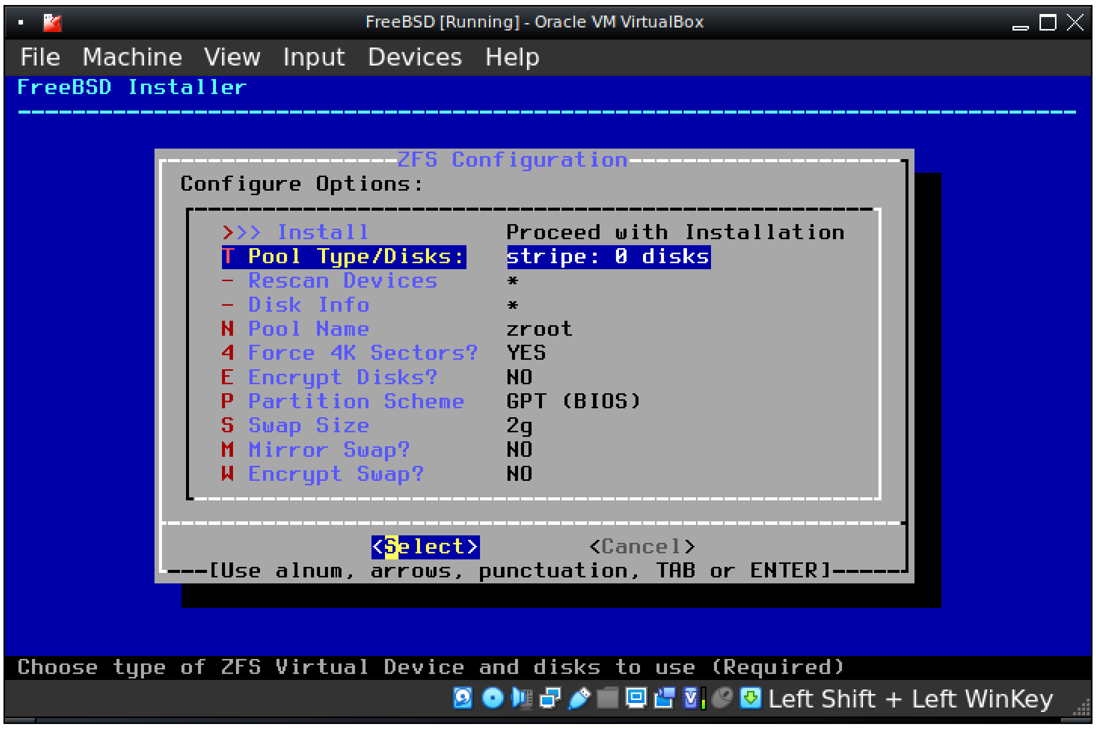
Unix beginners might prefer *GhostBSD*, a FreeBSD derivative with a fully graphical installer and systems management tools. (https://www.ghostbsd.org/) GhostBSD provides an experience very similar to Ubuntu Linux.

- Unparalleled reliability/robustness. Jobs can run for weeks or months without worry of a system crash or freeze. A FreeBSD HPC cluster I built in 2012 never experienced a single compute node crash in more than eight years of service. It experienced many extreme loads during that time. A few head node crashes early on were traced to a Dell firmware bug affecting single-CPU configurations of the dual-socket PowerEdge server. After upgrading the firmware, the head node never crashed again either.
- FreeBSD is renowned for its network performance and reliability. For this reason, many popular networking devices, including pfSense, OPNsense, TrueNAS, and Juniper network products, are based on FreeBSD. This makes FreeBSD a solid platform for big data processing and HPC clusters, which move huge amounts of data between machines.
- FreeBSD has a fully-integrated ZFS file system, that can be easily configured during OS installation. Installing FreeBSD to boot from a ZFS RAID is as easy as installing most operating systems to a single disk. If you have a PC with two or more disks and no hardware RAID controller, you can easily configure your disks in any of the standard ZFS RAID levels during the FreeBSD install process. The entire install process will take about five minutes. The screen shots below show the disk configuration steps.


FreeBSD [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Left Shift + Left WinKey

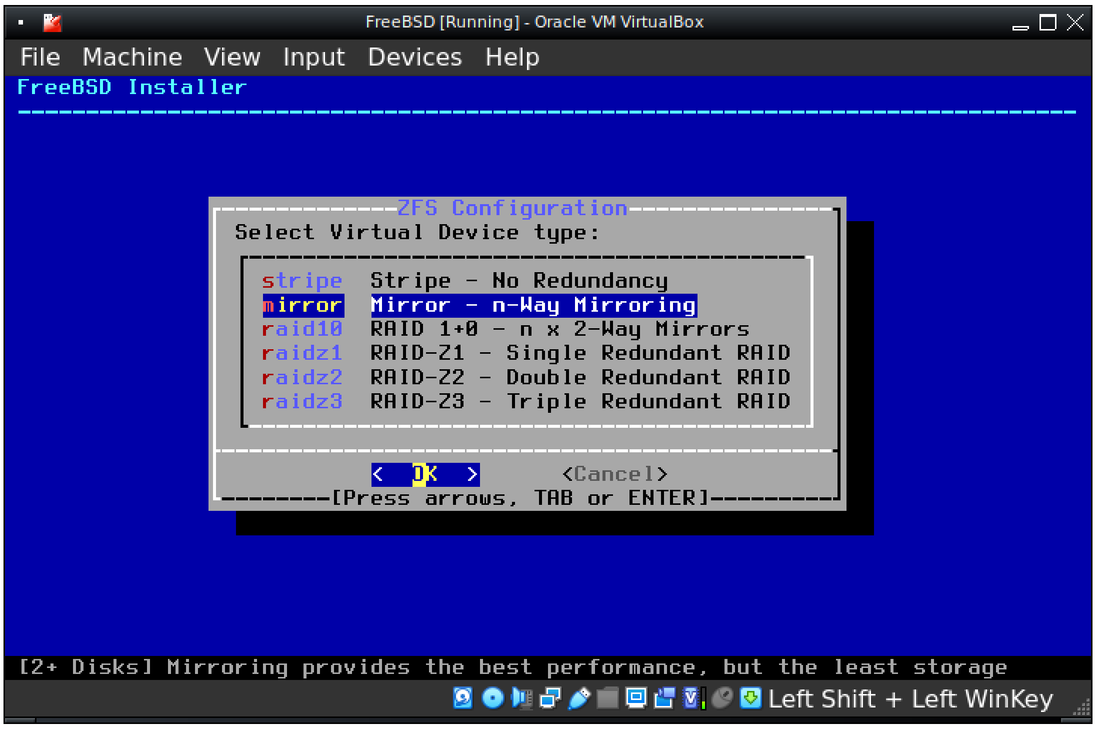
```
FreeBSD Installer

                         Partitioning
How would you like to partition your disk?

  Auto (ZFS)   Guided Root-on-ZFS
  Auto (UFS)   Guided UFS Disk Setup
  Manual       Manual Disk Setup (experts)
  Shell        Open a shell and partition by hand

          <  OK  >          <Cancel>

To use ZFS with less than 8GB RAM, see https://wiki.freebsd.org/ZFSTuningGuide
```

Figure 36.1: FreeBSD Installer Disk Partition Screen


FreeBSD [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Left Shift + Left WinKey

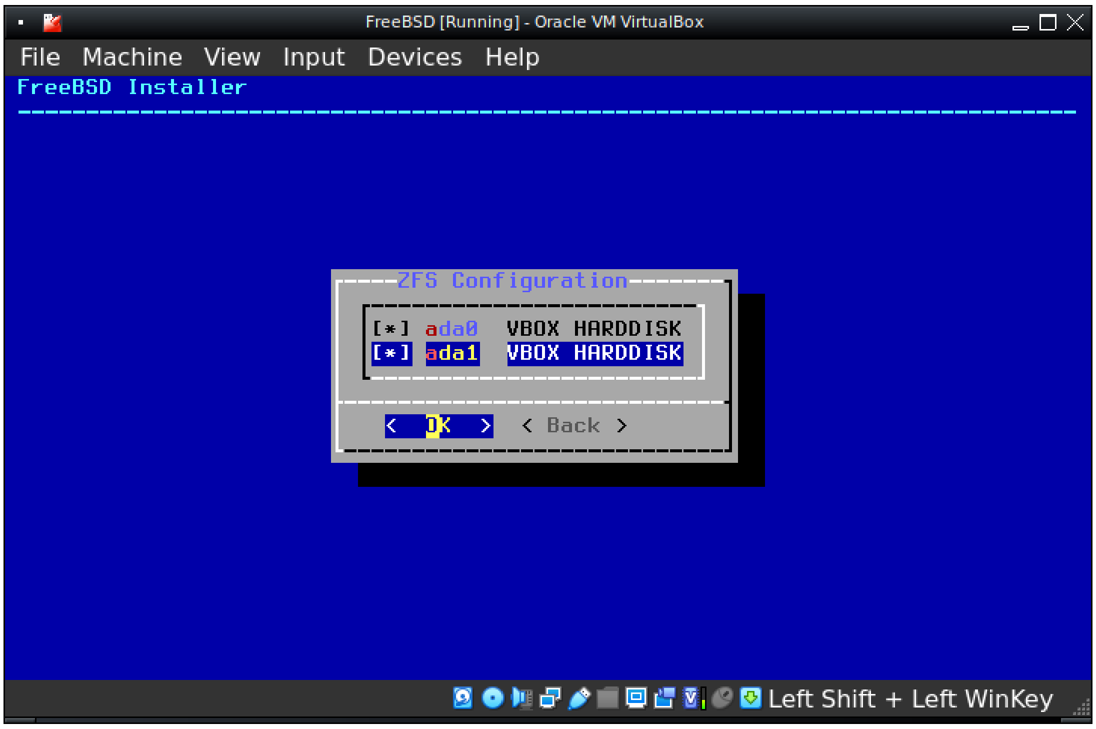
```
FreeBSD Installer

                    ZFS Configuration
Configure Options:

  >>> Install             Proceed with Installation
  T Pool Type/Disks:      stripe: 0 disks
  - Rescan Devices        *
  - Disk Info             *
  N Pool Name             zroot
  4 Force 4K Sectors?     YES
  E Encrypt Disks?        NO
  P Partition Scheme      GPT (BIOS)
  S Swap Size             2g
  M Mirror Swap?          NO
  W Encrypt Swap?         NO

          <Select>            <Cancel>
---[Use alnum, arrows, punctuation, TAB or ENTER]-----

Choose type of ZFS Virtual Device and disks to use (Required)
```

Figure 36.2: FreeBSD Installer ZFS Options



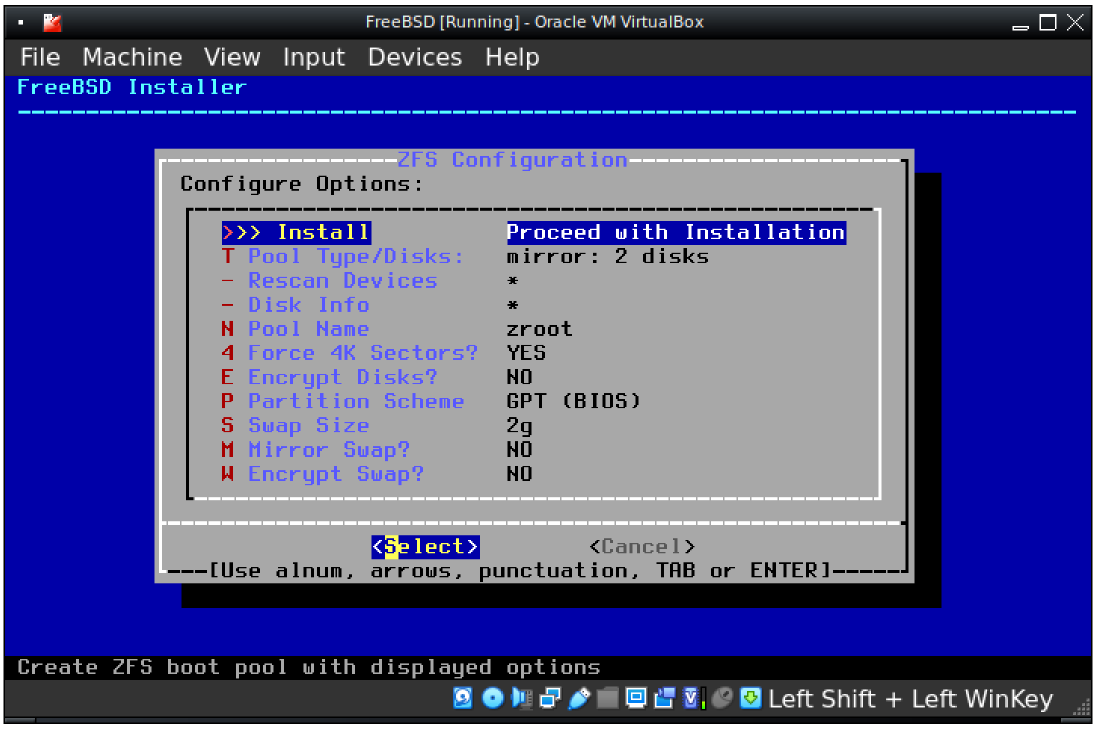
```
FreeBSD [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
FreeBSD Installer
---------------------------------------------------------------------------

                 ---------ZFS Configuration---------------
                 Select Virtual Device type:
                  ----------------------------------------
                 | stripe   Stripe - No Redundancy        |
                 | mirror   Mirror - n-Way Mirroring      |
                 | raid10   RAID 1+0 - n x 2-Way Mirrors  |
                 | raidz1   RAID-Z1 - Single Redundant RAID|
                 | raidz2   RAID-Z2 - Double Redundant RAID|
                 | raidz3   RAID-Z3 - Triple Redundant RAID|
                  ----------------------------------------
                 ------------------------------------------
                          <  OK  >        <Cancel>
                 --------[Press arrows, TAB or ENTER]------

[2+ Disks] Mirroring provides the best performance, but the least storage
                                                Left Shift + Left WinKey
```

Figure 36.3: FreeBSD Installer ZFS RAID Level Selection



```
FreeBSD [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
FreeBSD Installer
---------------------------------------------------------------------------

                        ---ZFS Configuration-------
                         ------------------------
                        | [*] ada0   VBOX HARDDISK |
                        | [*] ada1   VBOX HARDDISK |
                         ------------------------
                        ----------------------------
                          <  OK  >   < Back >
                        ----------------------------

                                                Left Shift + Left WinKey
```

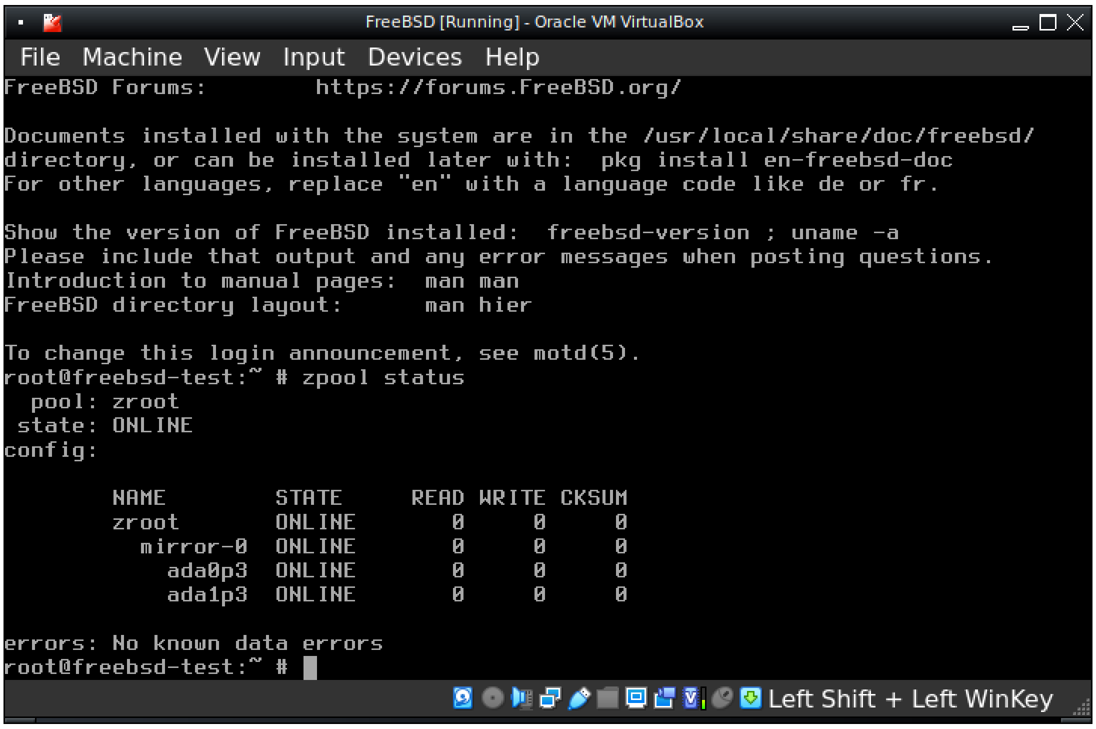
Figure 36.4: FreeBSD Installer ZFS Disk Selection


FreeBSD [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Left Shift + Left WinKey

```
FreeBSD Installer
-------------------------------------------------------------------------------

              ZFS Configuration
 Configure Options:

   >>> Install             Proceed with Installation
   T Pool Type/Disks:      mirror: 2 disks
   - Rescan Devices        *
   - Disk Info             *
   N Pool Name             zroot
   4 Force 4K Sectors?     YES
   E Encrypt Disks?        NO
   P Partition Scheme      GPT (BIOS)
   S Swap Size             2g
   M Mirror Swap?          NO
   W Encrypt Swap?         NO

              <Select>          <Cancel>
---[Use alnum, arrows, punctuation, TAB or ENTER]-----

Create ZFS boot pool with displayed options
```

Figure 36.5: FreeBSD Installer ZFS Commit

A few minutes later...


FreeBSD [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Left Shift + Left WinKey

```
FreeBSD Forums:        https://forums.FreeBSD.org/

Documents installed with the system are in the /usr/local/share/doc/freebsd/
directory, or can be installed later with:  pkg install en-freebsd-doc
For other languages, replace "en" with a language code like de or fr.

Show the version of FreeBSD installed:  freebsd-version ; uname -a
Please include that output and any error messages when posting questions.
Introduction to manual pages:  man man
FreeBSD directory layout:      man hier

To change this login announcement, see motd(5).
root@freebsd-test:~ # zpool status
  pool: zroot
 state: ONLINE
config:

        NAME        STATE     READ WRITE CKSUM
        zroot       ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            ada0p3  ONLINE       0     0     0
            ada1p3  ONLINE       0     0     0

errors: No known data errors
root@freebsd-test:~ #
```

Figure 36.6: FreeBSD ZFS Status

Booting from mirrored disks will protect your data from a single disk failure as well as increase performance. You should still back up data off-site to protect against other incidents such as fire, theft, or accidentally file deletion.

You can also manually configure other options outside the installer as well.

- The FreeBSD ports system allows users to quickly, easily, and reliably manage over 30,000 software packages. According to repology.org, FreeBSD ports has one of the largest collections, and also one of the highest percentage of packages that are up-to-date (offering the latest release).

  Unlike most competing package systems, FreeBSD ports allows you to build and install any package from source as easily as you would install the precompiled (binary) package. This allows building programs with additional, non-portable optimizations to take better advantage of your high-end CPUs, building with optional features, building with debug flags to help track down bugs in the software, etc.

  ```
  # Install the binary packages
  pkg install rna-seq

  # Install from source: Take longer, but no more effort from you
  cd /usr/ports/biology/rna-seq && make install
  ```

  Of the more than 30,000 packages in the collection, more than 2,000 are in the science categories.

  ```
  shell-prompt: count-science-ports
  astro         141
  biology       238
  cad           146
  math         1200
  science       505
  Total        2230
  ```

- The CentOS-based Linux compatibility actually makes FreeBSD more suitable for running commercial Linux applications than many Linux distributions. Most commercial Linux software is built on Redhat Enterprise Linux (RHEL), and much of it does not run properly on Linux platforms with newer tools and libraries. FreeBSD's Linux compatibility is based on CentOS, which is RHEL-compatible. FreeBSD users thus have access to newer tools provided by FreeBSD and a RHEL-compatible Linux environment at the same time, with no virtual machine or container overhead.

  FreeBSD's Linux compatibility is not an emulation layer. It is a kernel module that supports Linux system calls that differ from FreeBSD's. There is no additional overhead when running Linux binaries. In fact, Linux binaries sometimes run slightly faster on FreeBSD than they do on Linux.

## 36.2 Linux in Science

### 36.2.1 RHEL

Redhat Enterprise. Designed for running commercial software. Response to criticism about unstable and rapidly changing early Linux distributions. Snapshot of Fedora Linux. Tools for a major release of RHEL are never upgraded. They only receive fixes to improve reliability and maintain binary compatibility for closed-source applications. Libraries and tools much older than bleeding-edge Linux distributions that are more popular among individual users. Trade modernity for stability and compatibility.

Most HPC clusters run RHEL derivatives.

### 36.2.2 Ubuntu

Ubuntu, which is based on Debian, is popular among individual users due to its ease of installation and management via graphical tools. GhostBSD provides a similar experience on a FreeBSD platform. The Debian family also has the largest collection of quality-controlled packages, with FreeBSD ports not too far behind, and with a higher percentage of packages running the latest release. (repology)

Unfortunately, many scientists still perform caveman installs of scientific software on Linux systems rather than use a package manager.

## 36.3 macOS in Science

Based on FreeBSD and Mach kernel (also derived from BSD). NeXTSTEP OS for Next computer started by Steve Jobs after being pushed out of Apple, was based on FreeBSD and Mach. When Jobs returned to Apple, the original macOS was replaced by a derivative of the NeXTSTEP operating system.

MacOS is the most popular Unix platform by far among individual computer users. More than 100 million Macs in use today. Server market share is quite different.

MacOS allows users to run Unix-based open source software alongside commercial products like MS Office, without the use of VMs.

# Chapter 37

# Platform Selection

## 37.1 General Advice

No matter what operating system you use, you are going to have problems.

What you need to decide is what kinds of problems you can live with.

System crashes are the worst kind of problem for scientific computing, where analyses and simulations may takes days, weeks, or even months to run. If a system crash occurs when a job has been running for a month, someone's research may be delayed by a month (unless their software uses checkpointing, allowing it to be resumed from where it left off).

Reliability must be considered as a major factor when assessing the performance of a system. Long-term *throughput* (work completed per unit time) is heavily impacted by systems outages that cause jobs to be restarted.

It doesn't really matter why a system needs to be rebooted. It could be due to system freezes, *panics* (kernel detecting unrecoverable errors), or security updates so critical that they cannot wait. Systems that need to be rebooted frequently for any of these reasons should be considered less reliable.

*Uptime*, the time a system runs between reboots, should be monitored to determine reliability. The average uptime for popular operating systems varies from days to months.

System crashes are also the worst for IT staff who manage many machines. Suppose you manage 30 machines running an operating system that offers and average up time of a month or two. This means you have to deal with a system crash every day or two on average (unless you reboot machines for other reasons in the interim).

This is exactly the situation I experienced while supporting fMRI research using cutting-edge Linux distributions, such as Redhat (not Redhat enterprise, but the original Redhat, which evolved into Fedora), Mandrake, Caldera, SUSE (again, the original, not SUSE Enterprise).

Some of our Linux workstations would run for months without a problem while others were crashing every week. NFS servers running several different distributions would consistently freeze under heavy load. Systems would freeze for a few minutes at a time while writing DVD-RAMs. These were pristine installations with no invasive modifications. It's not anything we did to the systems, but just the nature of these cutting-edge distributions.

This is a fairly common issue. Some research groups resort to scheduled reboots in order to maximize likely up times from the moment an analysis was started. The HTCondor scheduler has an option to reboot a compute host after a job finishes for similar reasons.

This is in no way a criticism of cutting-edge Linux distributions. They play an important role in the Unix ecosystem, namely as a platform for testing new innovations. We need lots of people using new software systems in order to shake out most of the bugs and make it enterprise-ready, and cutting-edge Linux distributions serve this purpose very well. Many people want to try out the latest new features and don't need a system that can run for months without a reboot. In fact, most of them probably upgrade and reboot their systems every week or so, and as a result, rarely experience a system crash.

However, no operating system is the best at everything, and cutting-edge Linux distributions are not the best at providing stability. Some glitches should be expected from anything on the cutting edge.

For the average user maintaining one or two systems for personal use or development, the stability of a cutting-edge Linux system is generally more than adequate.

For scientists running simulations that take months or IT staff managing many systems, it could be a major nuisance.

One solution is to run an Enterprise Linux distribution, such as Redhat Enterprise, is described in Section 37.3, or SUSE Enterprise.

Another is to run a different Unix variant, such as FreeBSD, described in Section 37.4. This is the route we chose in our fMRI research labs, and it solved almost all of our stability issues. FreeBSD has always been extremely reliable and secure. System crashes are extremely rare. Almost every system crash I've experienced has been traced to a hardware problem or a configuration error on my part. Critical security updates, in my experience, occur less frequently than other systems such as Windows and Linux. If you're looking for a "set and forget" operating system to make your sysadmin duties easy, FreeBSD is a great option.

In addition to choosing an operating system that focuses on reliability, you may want to invest in a UPS and a RAID to protect against power outages and disk failures. If you're really worried, some systems also offer fault-tolerant RAM configurations, using some RAM chips for redundancy, akin to RAIDs.

## 37.2 Choosing Your Unix the Smart Way

Ultimately, the only thing that matters with respect to which Unix system you use is how well it runs the programs you need.

Many people make the mistake of choosing an operating system based on how "nice" it looks, what their friends (who often are not very computer savvy) are using, or how easy it is to install.

What's really important, though, is what happens *after* the system is up and running. The effort required to maintain it over the course of a couple years is by far the lion's share of the total cost of ownership, so get informed about what that cost will be for your particular needs before deciding.

Each operating system has its own focus, which may be very different from the rest.

The free operating systems include several systems based on BSD (Berkeley Systems Distribution), a free derivative of the original AT&T Unix from the University of California, Berkeley. It is the basis for FreeBSD, Mac OS X, NetBSD, OpenBSD, and a few others.

FreeBSD is the most popular among the free BSD-based systems. FreeBSD is known for its speed, ease of setup, robust network stack, and most of all its unparalleled stability. It is the primary server operating system used by Netflix, and WhatsApp. Netflix alone accounted for more than 1/3 of streaming Internet traffic in North America in 2015. (See http://appleinsider.com/articles/16/01/20/netflix-boasts-37-share-of-internet-traffic-in-north-america-compared-with-3-for-apples-itunes) FreeBSD is also the basis of advanced file servers such as FreeNAS, Isilon, NAS4Free, NetApp, and Panasas, and network equipment from Juniper Networks, and the open source pfSense firewall.

The FreeBSD ports system makes it trivial to install any of more than 30,000 packages, including most mainstream scientific software. FreeBSD ports can be installed automatically either from a binary package, or from source code if you desire different build options than the packages provide.

Mac OS X is essentially FreeBSD with Apple's proprietary user interface, so OS X users already have a complete Unix system on their Mac. In order to develop programs under Mac OS, you will need to install a compiler. Apple's development system, called Xcode, is available as a free download. The free and open source MacPorts system offers the ability to easily install thousands of software packages directly from the Internet. The MacPorts system is one of the most modern and robust ports systems available for any operating system. There are also other package managers for Mac OS X, such as Fink, Homebrew, and Pkgsrc.

There are many free Linux distributions, as well as commercial versions such as Red Hat Enterprise and SUSE Enterprise. The most popular free distributions for personal use are currently Mint and the Ubuntu line (Ubuntu, Kubuntu, and Xubuntu), which are based on Debian Linux. These systems are known for their ease of installation and maintenance, and cutting-edge new Linux features. Systems based on Debian Linux support the Debian packages system, which offers more than 40,000 packages available for easy installation.

**Caution** Be careful about judging the popularity of different operating systems based on package counts. The Debian packages collection is somewhat inflated by the fact that they tend to split a single software distribution into several packages. For example, libraries are typically provided by at least three packages, a base package for only the run time components (shared libraries), a -devel package for building your own programs with the library (header files), and a -doc package containing man/info pages, HTML, PDF, etc. Some libraries are further split into single/double precision libs, core and optional components, etc. Many other package systems, such as FreeBSD ports and pkgsrc, tend to provide all library components in a single package.

Gentoo Linux is a Linux system based heavily on ports. The Gentoo system installation process is very selective, and results in a compact, efficient system for each user's particular needs. Gentoo is not as easy to install as other Linux systems, but is a great choice for more experienced Linux users who want to maximize performance and stability. Like FreeBSD and Debian, Gentoo's ports system, known as portage, offers automated installation of nearly 20,000 software packages at the time of this writing.

The NetBSD project is committed to simplicity and portability. For this reason, NetBSD runs on far more hardware configurations than any other operating system.

The OpenBSD is run largely be computer security experts. Core security software such as OpenSSL and OpenSSH are developed by the OpenBSD project and used by most other operating systems.

Redhat Enterprise and SUSE Enterprise Linux are more conservative Linux-based systems designed to provide the stability required in enterprise environments. They are popular in corporate and academic data centers, where they support critical services, often running commercial software applications. They do not include the latest cutting edge Linux features, as doing so would jeopardize the stability they are meant to provide. They are based on older Linux kernels and system tools, which have been well-tested and debugged by users of cutting-edge Linux systems over several years.

## 37.3 RHEL/CentOS Linux

Redhat Enterprise Linux is a Linux distribution designed for reliability and long-term binary compatibility. Redhat, Inc. took a lot of heat during the 1990s for the inadequate stability of their product. In response, they invented Redhat Enterprise Linux (RHEL) in 2000.

Community Enterprise Linux (CentOS), essentially a free version of RHEL, had its first release in 2004. These systems are created by taking a snapshot of Fedora and spending a lot of time fixing bugs, without upgrading the core tools, which might introduce new bugs. Hence, they run older kernels, compilers, and core libraries like libc.

RHEL, CentOS and their derivatives are used on the vast majority of HPC clusters.

They are also used in data centers around the world to provide all kinds of services needed to keep business, governments, and other organizations running.

One of their major advantages is full support for many commercial scientific software, most of which are supported only on Windows, Mac, and Enterprise Linux.

Enterprise Linux is also more stable than cutting-edge Linux systems. Many Linux users are unaware of this fact, because it is not relevant to them. In my own experience, most Linux systems will provide average up times of a month or two, which is far more than the average computer user needs. Many people will install updates and reboot about once a week anyway, so they will rarely experience a system crash.

One of the disadvantages of Enterprise Linux is that they use older kernels, compilers, standard libraries, and other tools. This makes it difficult to build and run the latest open source software on Enterprise Linux.

The pkgsrc package manager, discussed in Section 39.4.3 can be a big help overcoming this limitation.

## 37.4 FreeBSD

Stability and performance are the primary goals for the FreeBSD base system.

FreeBSD seems to often be the target of false criticism from people who have little or no experience with it. If someone tells you that FreeBSD is "way behind", "not up to snuff", etc., take the Socratic approach: Ask them to describe some of its shortcomings in detail and watch them demonstrate their lack of knowledge.

In reality, FreeBSD is a very powerful, enterprise-class operating system, used in some of the most demanding environments on the planet. A short list of FreeBSD-based products and services you may be familiar with is below. See https://en.wikipedia.org/-wiki/List_of_products_based_on_FreeBSD for a more complete list.

- Netflix content servers, which alone are responsible for a large portion of all the Internet traffic in North America
- Large cloud services companies such as New York Internet and Webair
- Dell Compellent, FreeNAS, Isilon, NAS4Free, NetApp, and Panasas high-performance storage systems
- Juniper network equipment
- mOnOwall, OPNsense, Nokia IPSO, pfSense firewalls
- Trivago and Whatsapp servers
- CellOS (Playstation 3), and Orbis OS (Playstation 4)

You may hear that FreeBSD is not as cutting-edge as some of the Linux distributions popular for personal use. This may be true from certain esoteric perspectives, but the reality is that only a tiny fraction of programs require cutting-edge features that FreeBSD lacks and FreeBSD is capable of running virtually all the same programs as any Linux distribution, with little or no modification.

A reliable platform on which to run them is far more important in scientific computing and there is no general-use operating system more reliable than FreeBSD.

Enterprise Linux offers comparable reliability, but FreeBSD offers newer compilers and libraries than Enterprise Linux, making it easier to build and run the latest open source software.

The FreeBSD ports collection offers one of the largest available collections of cutting-edge software packages that can be installed in seconds with one simple command. Users can also choose between using the latest packages or packages from a quarterly snapshot of the collection for the sake of stability in their add-on packages as well as the operating system itself. The quarterly snapshot's receive bug fixes, but not upgrades, much like Enterprise Linux distribution.

FreeBSD ports can be easily converted to pkgsrc packages for deployment on Enterprise Linux and other Unix-compatible systems.

FreeBSD has a Linux compatibility system based on CentOS. It can run most closed-source software built on RHEL/CentOS, although complex packages (e.g. Matlab) may be tricky to install. Ultimately, though, FreeBSD is actually more binary-compatible with RHEL than most Linux distributions. It uses tools and libraries straight from the CentOS Yum repository. The RPMs there are easily converted to FreeBSD ports for quick, clean deployment on FreeBSD systems.

Note that the compatibility system is not an emulation layer. There is no performance penalty for running Linux binaries on a FreeBSD system, and in fact some Linux executables may run faster on FreeBSD than they do on Linux. The system consists of a kernel module to support system calls that exist only in Linux, and the necessary run time tools and libraries to support Linux executables. The system only requires a small amount of additional RAM for the kernel module and disk space for Linux tools and libraries.

Hence, if you are running mostly open source and one or two closed-source Linux applications, FreeBSD may be a good platform for you. If you are running primarily complex closed-source Linux applications (Matlab, ANSYS, Abaqus, etc.), you will likely be better off running an Enterprise Linux system.

ZFS is fully-integrated into the FreeBSD kernel, and is becoming the primary file system for FreeBSD. The FreeBSD installer makes it easy to configure and boot from a ZFS RAID.

The UFS2 file system is still fully supported, and a good choice for those who don't want the high memory requirements of ZFS. UFS2 has many advanced features, such as an 8 ZiB file system capacity, soft updates (which ensure file system consistency without the use of a journal), an optional journal for quicker crash recovery, and backgrounded file system checks (which allow the file system to be checked and repaired while in-use, eliminating boot delays even if the journal cannot resolve consistency issues).

There are many other advanced features and tools such as FreeBSD jails (a highly developed container system), bhyve, qemu, VirtualBox, and Xen for virtualization, multiple firewall implementations, network virtualization, and mfiutil for managing LSI MegaRAID controllers, to name a few.

FreeBSD is a great platform for scientific computing in its own right, especially for running the latest open source software. It’s also a great sandbox environment for testing software that may later be run on RHEL/CentOS via pkgsrc.

## 37.5 Running a Desktop Unix System

Most mainstream operating systems today are Unix compatible. Microsoft Windows is the only mainstream operating system that is not Unix compatible, but there are free compatibility systems available for Windows to provide some degree of compatibility and interoperability with Unix.

The de facto standard of Unix compatibility for Windows is Cygwin, http://cygwin.com, which is free and installs in about 10 minutes. There are alternatives to Cygwin, but Cygwin is the easiest to use and offers by far the most features and software packages.

**Note** None of the Unix compatibility systems for Windows are nearly as fast as a genuine Unix system on the same hardware, but they are fast enough for most purposes. If you want to maximize performance, there are many BSD Unix and Linux systems available for free.

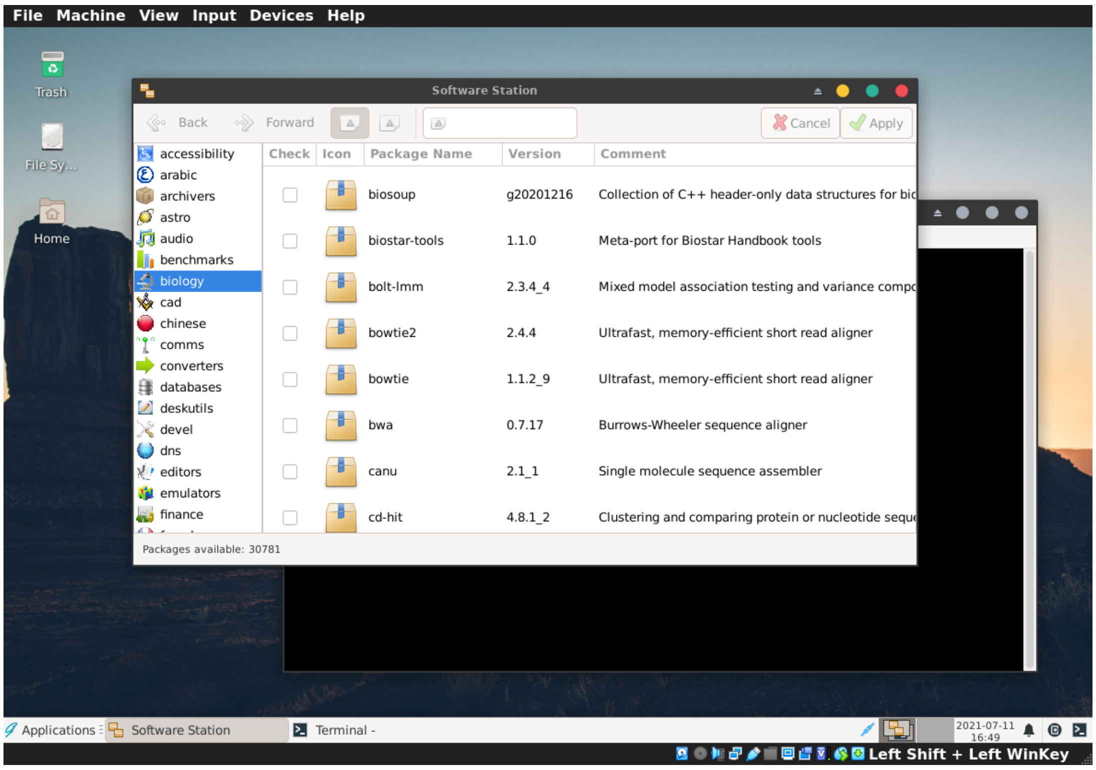
Another option for running Unix programs on a Windows computer is to use a *virtual machine (VM)*. This is discussed in Chapter 40.

Lastly, many Windows programs can be run directly under Unix, without a virtual machine running Windows, if the Unix system is running on x86-based hardware. This is accomplished using WINE, a Windows API emulator. WINE attempts to emulate the entire Windows system, as opposed to virtual machines, which emulate hardware. Emulating Windows is more ambitious, but eliminates the need to install and maintain a separate Windows operating system. Instead, the Windows applications run directly under Unix, with the WINE compatibility layer between them and the Unix system.

While it is possible to create a Unix-like environment under Windows using a system such as Cygwin, such systems have some inherent limitations in their capabilities and performance. Installing a Unix-compatible operating system directly has many benefits, especially for those developing their own code to run on the cluster.

Many professional quality Unix-based operating systems are available free of charge, and with no strings attached. These systems run a wide variety of high-quality free software, as well as many commercial applications. Hence, it is possible for researchers to develop Unix-compatible programs at very low cost that will run both on their personal workstation or laptop, and a cluster or grid.

One of the easiest Unix systems to install and manage is GhostBSD, a free, open source derivative of FreeBSD with a simple graphical installer, "Control Panel", and software manager:



| Check | Icon | Package Name | Version | Comment |
|---|---|---|---|---|
| ☐ | | biosoup | g20201216 | Collection of C++ header-only data structures for bi |
| ☐ | | biostar-tools | 1.1.0 | Meta-port for Biostar Handbook tools |
| ☐ | | bolt-lmm | 2.3.4_4 | Mixed model association testing and variance comp |
| ☐ | | bowtie2 | 2.4.4 | Ultrafast, memory-efficient short read aligner |
| ☐ | | bowtie | 1.1.2_9 | Ultrafast, memory-efficient short read aligner |
| ☐ | | bwa | 0.7.17 | Burrows-Wheeler sequence aligner |
| ☐ | | canu | 2.1_1 | Single molecule sequence assembler |
| ☐ | | cd-hit | 4.8.1_2 | Clustering and comparing protein or nucleotide sequ |

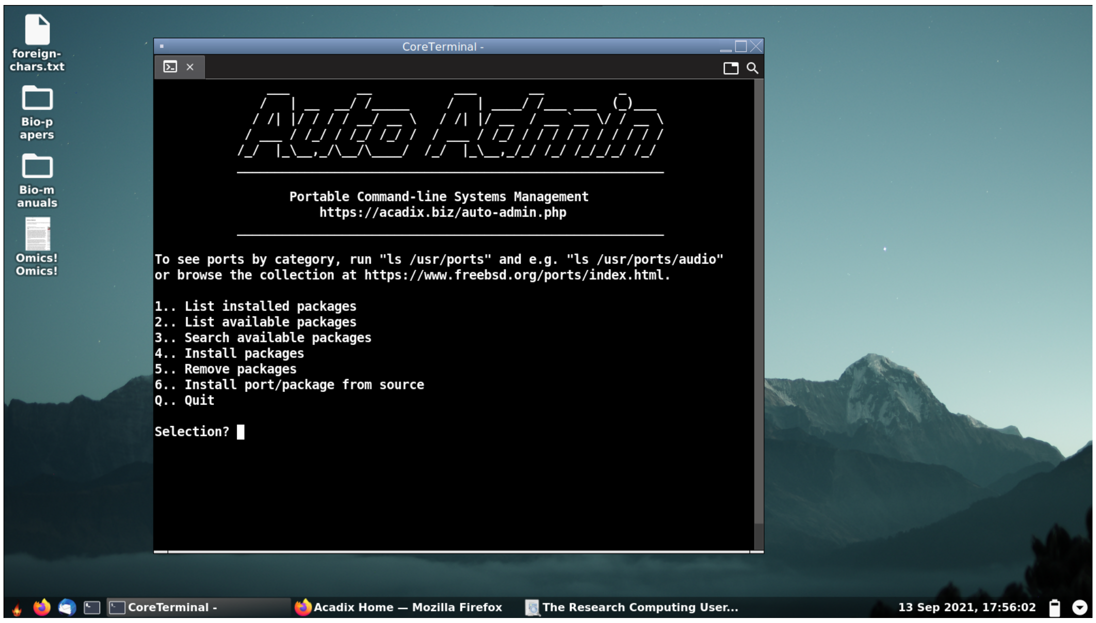
Packages available: 30781

*A GhostBSD system running XFCE desktop.*

GhostBSD is extremely easy to install and manage, as well as extremely reliable. If you want to try out Unix while encountering as few hurdles as possible, GhostBSD is probably your best bet.

Similar to GhostBSD are the Ubuntu family of Linux systems (Ubuntu, Kubuntu, Xubuntu, Edubuntu, ...). Each of these Linux distributions is built on Debian Linux, with a different desktop environment. ( Ubuntu uses Gnome, Kubuntu KDE, Xubuntu XFCE, etc.)

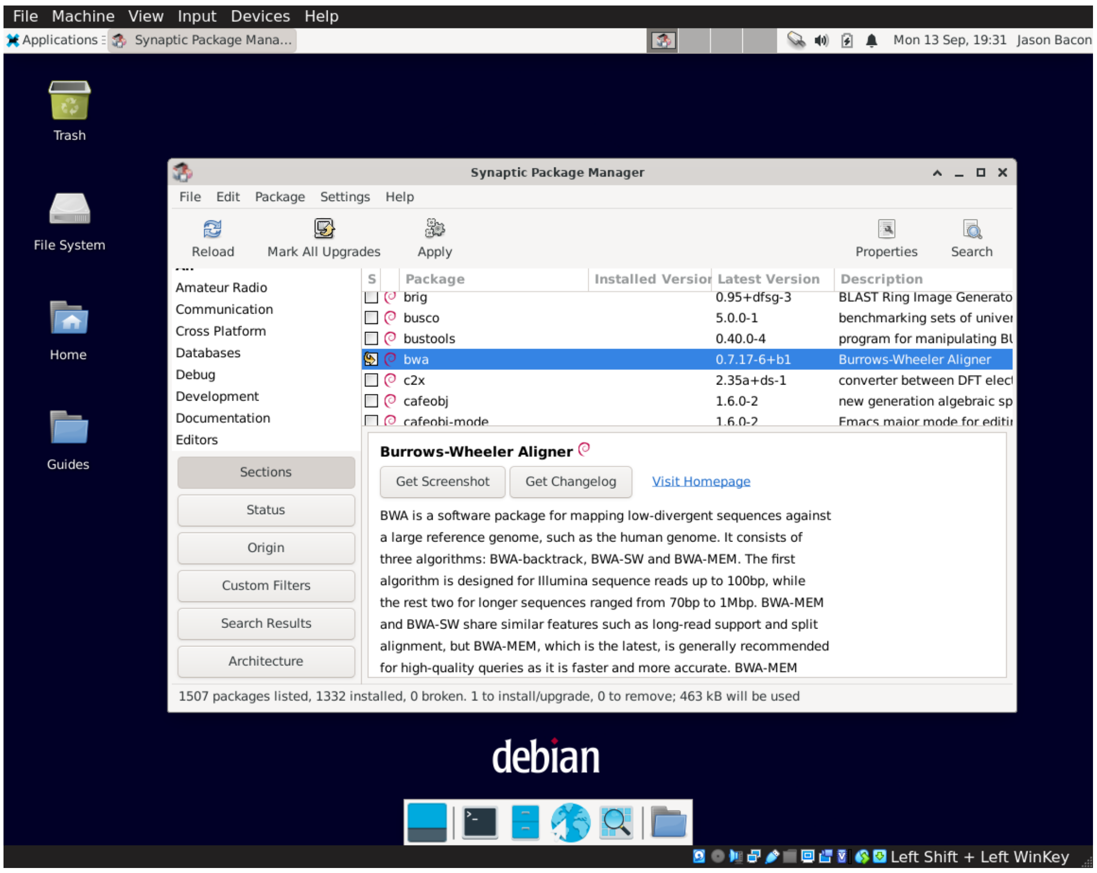
Another alternative for the slightly more computer-savvy is to do a stock FreeBSD installation and then install and run the sysutils/desktop-installer port. This option simply helps you easily configure FreeBSD for use as a desktop system using standard tools provided by the system and FreeBSD ports. The whole process can take as little as 15 minutes on a fast computer with a fast Internet connection. Just run desktop-installer from a terminal and answer the questions.



```
          Portable Command-line Systems Management
              https://acadix.biz/auto-admin.php

To see ports by category, run "ls /usr/ports" and e.g. "ls /usr/ports/audio"
or browse the collection at https://www.freebsd.org/ports/index.html.

1.. List installed packages
2.. List available packages
3.. Search available packages
4.. Install packages
5.. Remove packages
6.. Install port/package from source
Q.. Quit

Selection?
```

*A FreeBSD system running Lumina desktop.*

The Debian system itself has also become relatively easy to install and manage in recent years. It lacks some of the bells and whistles of Ubuntu, but may be a bit faster and more stable as a result.



| S | Package | Installed Version | Latest Version | Description |
|---|---|---|---|---|
| ☐ | brig | | 0.95+dfsg-3 | BLAST Ring Image Generato |
| ☐ | busco | | 5.0.0-1 | benchmarking sets of univer |
| ☐ | bustools | | 0.40.0-4 | program for manipulating BU |
| | bwa | | 0.7.17-6+b1 | Burrows-Wheeler Aligner |
| ☐ | c2x | | 2.35a+ds-1 | converter between DFT elect |
| ☐ | cafeobj | | 1.6.0-2 | new generation algebraic sp |
| ☐ | cafeobj-mode | | 1.6.0-2 | Emacs major mode for editi |

**Burrows-Wheeler Aligner**

BWA is a software package for mapping low-divergent sequences against a large reference genome, such as the human genome. It consists of three algorithms: BWA-backtrack, BWA-SW and BWA-MEM. The first algorithm is designed for Illumina sequence reads up to 100bp, while the rest two for longer sequences ranged from 70bp to 1Mbp. BWA-MEM and BWA-SW share similar features such as long-read support and split alignment, but BWA-MEM, which is the latest, is generally recommended for high-quality queries as it is faster and more accurate. BWA-MEM

1507 packages listed, 1332 installed, 0 broken. 1 to install/upgrade, 0 to remove; 463 kB will be used

*A Debian system running XFCE desktop.*

All of these systems have convenient methods for installing security updates and minor software upgrades.

When it comes time for a serious upgrade of the OS, don't bother with upgrade tools. Back up your important files, reformat the disk, do a fresh install of the newer version, and restore your files.

Many hours are wasted trying to fix systems that have been broken by upgrades or were broken before the upgrade. It would have been faster and easier in many cases to run a backup and do a fresh install. You will need to do fresh installs sometimes anyway, so you might as well become good at it and use it as your primary method.

## 37.6 Unix File System Comparison

Windows file systems become fragmented over time as file are created and removed. Windows users should therefore run a defragmentation tool periodically to improve disk performance.

Unix file systems, in contrast, do continuous defragmentation, so performance will not degrade significantly over time.

Overwrite performance on some file systems is slower than initial write. Hence, removing files before overwriting them may help program run times.

Most Unix systems offer multiple choices for file systems. Most modern file systems use *journaling*, in which data that is critical to maintaining file system integrity in the event of a system crash is written to the disk immediately instead of waiting in a memory buffer.

To save time, this data is queued to a special area on the disk known called the journal. Writing to a journal is faster than saving the data in it's final location, since it requires fewer disk head movements.

Journaling reduces write performance, since data is first written to a journal and later moved to its final location. This takes more time and more disk head movements than storing data in a memory buffer until it is written to its final location. However, the performance penalty is marginal if done intelligently. All modern Unix file systems use advanced journaling methods to minimize the performance hit and disk wear.

Popular file systems:

- EXT is the most commonly used file system on Linux systems. EXT3 was the first to including journaling, basically as a feature added to EXT2. EXT2 was notorious for incredibly slow file system checks and repairs. The journaling features added by EXT3 greatly reduced the need for repairs, but EXT3 is not the best performer overall and is also hard on disks due to excessive head movements.

  EXT4 represents a vast improvement over EXT3 due to major redesign of key components. Performance and reliability are solid.

- HFS is the file system used by Mac OS X. Features and performance are generally positive. One potential problem for Unix users is the lack of true case-sensitivity. HFS is *case-preserving*, but not *case-sensitive*. This means that if you create a file named "Tempfile", the "T" will be remembered as a capital. However, it is not distinguished from a lower-case "t", so the file may be referred to as "tempfile". Also, you cannot have two files in the same directory called "Tempfile" and "tempfile", because these two file names are considered the same.

- UFS (Unix File System) evolved from the original Unix system 7 file system and is now used by most BSD systems as well as some commercial systems such as SunOS/Solaris and HP-UX.

  FreeBSD's UFS2 includes a unique feature called *soft updates*, which protects file system integrity in the event of a system crash without using a journal. This allows UFS2 to exhibit better write performance and less disk wear.

- XFS is a file system developed by SGI for it's commercial IRIX operating system during the 1990s, which were popular for high-end graphics. SGI IRIX machines were used to develop and featured in the movie Jurassic Park.

  XFS has been fully integrated into Linux and is now used as an alternative to EXT4 where high performance and very large partitions are desired.

- ZFS is a unique combination of a file system combined with a *volume manager*, developed by Sun Microsystems.

  ZFS is widely regarded as the most advanced file system to date. One of its most unique features is the fact that it does not require partitioning the disk in order to separate file systems. With other file systems, if you want home and /var to be separated and utilize different settings, you must divide the disk into separate partitions. Choosing the optimal size for each partition is almost impossible since we cannot predict the space requirements of the future. With ZFS, you can create multiple file systems, each with its own settings, all of which allocate blocks from the same pool. Thus, you never run out of space in one file system while having unutilized space in others. ZFS also offers advanced software RAID that generally outperforms hardware RAID systems, and many other advanced features such as compression and encryption.

  ZFS has been fully integrated into FreeBSD and is now the default file system for high-end FreeBSD servers as well as the GhostBSD desktop system. ZFS does require a lot of RAM, however, so UFS2 is still a better choice for low-end hardware such as net books and embedded FreeBSD systems.

## 37.7 Network File System

Network File System, or NFS, is a standard Unix network protocol that allows disk partitions on one Unix to be directly accessed from other computers. In concept, NFS is similar to Apple's AFS and Microsoft's SMB/CIFS.

Access to files across an NFS link is generally somewhat slower than local disk access, due to the overhead of network communication. Speed may be limited either by the local disk performance on the NFS server or by the bandwidth of the network. For example, of an NFS server has a RAID that can deliver 500 megabytes per second locally and a 1 gigabit (~100 megabyte per second) network, then the disk performance seen by NFS clients will be limited by the network to about 100 megabytes per second.

Unix systems also allow other computers to access their disks using non-Unix protocols like AFS and SMB/CIFS if necessary. For example, Samba is an open source implementation of the SMB/CIFS protocol that allows Windows computers to access data on Unix disks.

# Chapter 38

# System Security

## 38.1 Securing a new System

- Configure firewall or TCP wrappers to allow incoming traffic from only specific hosts.
- Create ONE account with administrator rights and use it only for system updates and software installations.
- Do not share login accounts. Create SEPARATE accounts for each user, without administrator rights, and use them for all normal work.
- NEVER share your password with ANYONE. PERIOD. NOBODY should ever ask you for your password. Other users have no right to mess with your login account. IT staff with rights to manage a machine do not need your password, so be suspicious if they ask for it.
- Store passwords in KeePassX or a similar encrypted password vault. Use a strong password for each KeePassX database.
- If you set up a computer to allow remote access, use ONLY systems that encrypt ALL traffic. If you are not sure your remote access software encrypts everything, DO NOT ENABLE IT. Talk to a professional about how to securely access the computer remotely before allowing it.

## 38.2 I've Been Hacked!

If you suspect that your computer has been hacked, unplug it from the network (or disable WiFi), but do not turn it off. Call your local computer security experts, and do not touch the computer until they arrive.

Once a computer has been hacked, that operating system installation is finished. Don't even think about trying to patch your way out of it. The only way to clean a hacked system is by backing up your files, reformatting the hard disk, reinstalling, and changing every password that was ever typed on the computer, whether it was a local password or a password on another computer someone connected to fro the hacked computer.

Antivirus and other antimalware software only detects known malware. If a hacker installs a custom program of their own design, it will not be detected.

There are many sites listing the steps you need to take, but most are incomplete. Below is a fairly comprehensive list.

1. Unplug the computer from the network to cut off the hacker's access immediately.
2. Stop using the computer. Especially, do not use the computer to log into any other computers over the network, as you will likely be giving away your passwords to those machines as you type them.
3. USING A DIFFERENT COMPUTER, immediately change your passwords on every other computer that you have ever connected to from the hacked computer. Every password that has ever been typed on the hacked machine must be changed, as the hacker may have been monitoring all of your keystrokes for a long time before the intrusion was detected. That includes local passwords on the PC as well as passwords entered on the PC to log into remote machines.

4. If you have IT staff trained in computer security, contact them. They may want to do a forensic analysis on the machine to determine who hacked it and how.
5. Back up your data files. Note that they may have been corrupted by the hacker, so check them carefully before relying on them.
6. Do not back up any programs, scripts, installation media, or configuration files. They may be infected with malware and restoring them to the newly installed system will allow the hacker right back in. Antivirus and other antimalware programs do not detect all malware. Don't think for a minute the your computer is clean just because your virus scan didn't find anything. This is foolish wishful thinking that will only cause more problems for you and others around you.
7. Reformat all disks in the computer and reinstall the operating system from trusted install media. ( Do not use install media that was stored on the hacked computer! )
8. Do not use any of the same passwords on the new installation. Create new passwords for every user and every application on the computer.
9. Restore your data files from backup.
10. Reinstall all programs from trusted installation media.

# Chapter 39

# Software Management

## 39.1 The Stone Age vs. Today

There are many thousands of quality open source applications and libraries available for Unix systems.

Just knowing what exists can be a daunting task. Fortunately, software management systems such as the FreeBSD ports have organized documentation about what is available. You can browse software packages by category on the ports website: http://www.freebsd.org/ports/index.html. Even if you don't use FreeBSD, this software listing is a great resource just for finding out what's available.

Installing various open source packages can also be a daunting task, especially since the developers use many different programming languages and build systems.

Many valuable man-hours are lost to stone-age software management, i.e. manually downloading, unpacking, patching, building, and installing open source software.

Free software isn't very free if it takes 20 hours of someone's time to get it running. An average professional has a total cost to their employer on the order of $50/hour. Hence, 20 hours of their time = $1,000. If 1,000 professionals around the world spend an average of 20 hours installing the same software package, then $1,000,000 worth of highly valuable man-hours have gone to waste. Even worse, many people eventually give up on installing software entirely, so there are no gains to balance this loss.

In most cases, the software could have been installed in seconds using a software management system (SMS) and all that time could have been spent doing something productive.

When choosing a Unix system to run, a good *ports* or *packages* system is an important consideration. A ports or packages system automatically downloads and installs software from the Internet. Such systems also automatically install additional *prerequisite* ports or packages required by the package you requested.

For example, to install Firefox, you would first need to install dozens of libraries and other utilities that Firefox requires in order to run properly. ( When you install Firefox on Windows or Max OS X, you are actually installing a bundle of all these packages. )

The ports or packages system will install all of them automatically when you choose to install Firefox. This allows you install software in seconds or minutes that might otherwise take days or weeks for an inexperienced programmer to manually download, patch, and compile.

## 39.2 Goals

Complete execution well before deadline.

Minimize man-hours.

Maximizing execution speed of every program is a foolish waste of resources.

Focus on big gains, 80/20 rule (Pareto principal). 20% of effort typically yields 80% of gains. Don't waste time or hardware trying to squeeze out marginal gains unless it's really necessary. If it won't mean meeting a deadline that would otherwise be missed, or free up saturated resources, then it's a waste.

## 39.3 The Computational Science Time Line

The figure below represents the time line of a computational science project.

| Development Time | Deployment Time | Learning Time | Run Time |
|---|---|---|---|
| Hours to years | Hours to months (or never) | Hours to weeks | Hours to months |

Table 39.1: Computation Time Line

### 39.3.1 Development Time

Not relevant to most researchers.

Learn software development life cycle, efficient coding and testing techniques.

Understand objective language factors; compiled vs interpreted speed, portability, etc.

### 39.3.2 Deployment Time

Deployment time virtually eliminated by package managers, described in Section 39.4.

### 39.3.3 Learning Time

Largely up to end-user.

IT staff can help organize documentation and training.

### 39.3.4 Run Time

Software efficiency (algorithms, language selection) should always be the first focus. Often software can be made to run many times faster simply by changing the inputs. Is the resolution of your fluid model higher than you really need? Are you analyzing garbage data along with the useful data? Is your algorithm implemented in an interpreted language such as Matlab, Perl, or Python? If so, it might run 100 times faster if rewritten in C, C++, or Fortran. See Section 13.4.

System reliability (system crashes cause major setbacks, especially where check pointing is not used). Operating system, (FreeBSD, ENTERPRISE Linux), UPS.

Some scientific analyses take a month or more to run. FSL, single-threaded. Average up time of 1 month is not good enough.

From the researcher's perspective, this may mean restarting simulations or analyses, losing weeks worth of work if check pointing is not possible.

From the sysadmin's perspective, if managing 30 machines with an average up time of 1 month, you average 1 system crash per day.

Some choose scheduled reboots to maximize likelihood of completing jobs. Better to do your homework and find an operating system with longer up times.

Parallelism is expensive in terms of both hardware and learning curve. It should be considered a last resort after attempting to improve software performance.

## 39.4 Package Managers

### 39.4.1 Motivation

A package manager is a system for installing, removing, upgrading or downgrading software packages. They ensure that proper versions of dependency software are installed and keep track of all files installed as part of the package.

A *caveman installation* is an installation performed by downloading, patching, building and installing software manually or via a custom script that is not part of a package manager. This is a temporary, isolated solution.

A package added to a package manager is a permanent, global solution.

A package need only be created once, and then allows the software to be easily deployed any number of times on any number of computers worldwide.

1,000 people spending 2 hours each doing caveman (ad hoc) installations = 2000 man-hours = 1 year's salary.

1 person spending 2 hours creating a package + 999 spending 2 seconds typing a package install command = 2.55 man-hours.

There is a significant, but one-time investment in learning to package software. Once learned, creating a package usually takes LESS time than a caveman install.

Have you ever been in a panic because your server went down and you're approaching a deadline to get your analysis or models done? If you deploy the software with a package manager, no problem... Just install it on another machine and carry on. If you've done a caveman install, you might be dead in the water for a while until you can restore the server or duplicate the installation on another.

Some packages managers allow the end-user to build from source with many combinations of options, compilers, alternate libraries (BLAS, Atlas, OpenBLAS). FreeBSD ports, Gentoo Portage, MacPorts, pkgsrc.

This provides the user more flexibility.

Binary packages distributed by any package manager must be portable and thus use only minimal optimizations. We can do optimized builds from source (make.conf, mk.conf or command-line additions) such as -march=native.

Building from source takes longer, but is no more difficult for you. It also provides many advantages, such as:

- The ability to build an optimized installation. Binary (precompiled) packages must be able to run on most CPUs still in use, so they do not take advantage of the latest CPU features (such as AVX2 as of this writing in February 2019). When compiling a package from source, we can add compiler flags such as `-march=native` to optimize for the local CPU. The package produced may not work on older CPUs.
- The ability to install to a different prefix. This can be useful for doing multiple installations with different build options, or installing multiple versions of the same software.
- The ability to easily install software whose license does not allow redistribution in binary form
- The ability to easily test patches.

It also makes it easy to use the package manager to systematically deploy work-in-progress packages that are not yet complete and for which no binary packages have been built.

Most package managers work only on one platform or possible a few closely related platforms. Some work on multiple platforms with some limitations. The pkgsrc package manager is unique in that it provides general purpose package management for any Unix compatible platform.

Table 39.2 provides some information about popular package managers.

A direct comparison of which collection is "biggest" is not really feasible. A raw count of Debian packages will produce a higher number than the others, but this is in part due to the Debian tradition of creating separate packages for documentation (-doc packages) and header files (-dev packages). FreeBSD ports and many other package managers traditionally include documentation and headers in the core package. For example, below are listings of FFTW (fast Fourier transform) packages on Debian and FreeBSD:

| Name | Platforms | Build From Source? |
|---|---|---|
| Conda | Linux, Mac, Windows | No |
| dports | Dragonfly BSD (derived from FreeBSD ports) | Yes |
| Debian Packages | Debian-based (Debian, Ubuntu, etc) | No |
| FreeBSD Ports | FreeBSD, Dragonfly BSD* | Yes |
| MacPorts | OS X | Yes |
| pkgsrc | Any POSIX | Yes |
| Portage | Gentoo Linux | Yes |
| Yum/RPM | Redhat Enterprise Linux, CentOS | No |

Table 39.2: Package Manager Comparison

Software in the RHEL Yum repository also tends to be older versions than you will find in Debian, FreeBSD, or Gentoo packages.

This is in no way a criticism of Red Hat Enterprise Linux, but simply illustrates that it was designed for a different purpose, namely highly stable enterprise servers running commercial software. The packages in Yum are intended to remain binary compatible with commercial applications for 7 years, not to provide the latest open source applications.

The Pkgsrc software management system can be used to maintain more recent open source software on enterprise Linux systems, separate from the commercial software and support software installed via Yum. Pkgsrc offers over 17,000 packages, most of which are tested on CentOS.

Pkgsrc has the additional benefit of being the only cross-platform SMS. It can currently be used to install most packages on NetBSD, Dragonfly BSD, Linux, Mac OS X, and Interix (a MS Windows Unix layer).

Even if you have to manually build and install your own software, you can probably install most of the prerequisites from an SMS. Doing so will save you time and avoid conflicting installations of the same package on your system.

Better yet, learn how to create packages for an SMS, so that others don't have to duplicate your effort in the future. Each SMS is a prime example of global collaboration, where each user leverages the work of thousands of others. The best way to support this collaboration is by contributing to it, even if in a very small way.

### 39.4.2 FreeBSD Ports

The FreeBSD ports system represents one of the largest package collections available, and it runs on a platform offering enterprise stability and near-optimal performance.

29,483 packages as of Feb 2 2018.

Port options allow many possible build option combinations for some ports (R is a good example). Some other package managers would require separate binary packages to provide the same support.

Most core scientific libraries are well-tested and maintained. (BLAS, LAPACK, Eigen, R, Octave, mpich2, openmpi, etc.)

Easy to deploy latest open source software, easy to convert to pkgsrc for deployment on other POSIX systems. Great scientific computing platform and sandbox environment.

Advanced development tools (ports-mgmt category), portlint, stage-qa, poudriere.

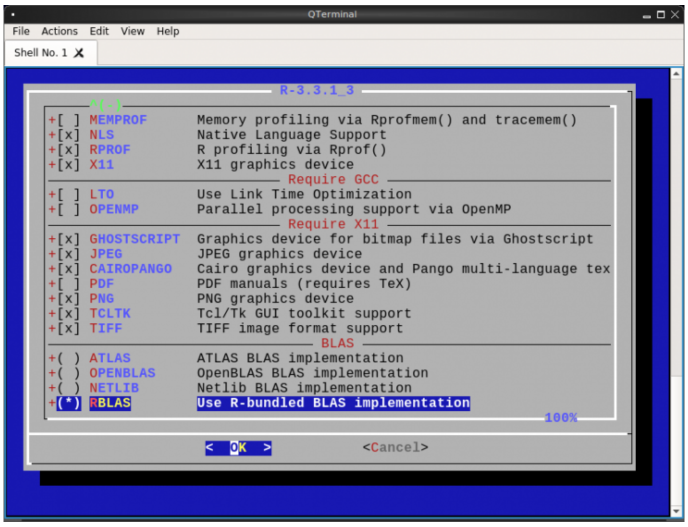
```
DEVELOPER=yes
```

Security checks

```
shell-prompt: pkg install R
```

```
shell-prompt: cd /usr/ports/math/R
shell-prompt: make rmconfig
shell-prompt: make install
```

Port options dialog for R:



```
                          R-3.3.1_3
    ^(-)
+[ ] MEMPROF       Memory profiling via Rprofmem() and tracemem()
+[x] NLS           Native Language Support
+[x] RPROF         R profiling via Rprof()
+[x] X11           X11 graphics device
                         Require GCC
+[ ] LTO           Use Link Time Optimization
+[ ] OPENMP        Parallel processing support via OpenMP
                         Require X11
+[x] GHOSTSCRIPT   Graphics device for bitmap files via Ghostscript
+[x] JPEG          JPEG graphics device
+[x] CAIROPANGO    Cairo graphics device and Pango multi-language tex
+[ ] PDF           PDF manuals (requires TeX)
+[x] PNG           PNG graphics device
+[x] TCLTK         Tcl/Tk GUI toolkit support
+[x] TIFF          TIFF image format support
                             BLAS
+( ) ATLAS         ATLAS BLAS implementation
+( ) OPENBLAS      OpenBLAS BLAS implementation
+( ) NETLIB        Netlib BLAS implementation
+(*) RBLAS         Use R-bundled BLAS implementation
                                                         100%

               <  OK  >            <Cancel>
```

Porter's Handbook

Example: http://acadix.biz/hello.php

Install freebsd-ports-wip: https://github.com/outpaddling/freebsd-ports-wip

Add the following to ~root/.porttools:

```
EMAIL="your-email@some.domain"
FULLNAME="Your Full Name"
```

```
shell-prompt: pkg install porttools
shell-prompt: wip-update
shell-prompt: wip-reinstall-port port-dev
shell-prompt: cd /usr/ports/wip
shell-prompt: port create hello

The port directory name given here should usually be all lower-case,
except for ports using perl CPAN and a few other cases.  The PORTNAME
is usally lower-cased as well, but there is not general agreement on this.
It's not that important as "pkg install" is case-insensitive.

shell-prompt: cd hello
shell-prompt: wip-edit

See /usr/ports/Mk/bsd.licenses.db.mk for list of valid licenses.
You can comment out LICENSE_FILE= until after the distfile is downloaded
and unpacked if that's more convenient that figure out it's location
via the web.  It should usually be prefixed with ${WRKSRC}, e.g.

LICENSE_FILE=   ${WRKSRC}/COPYING

shell-prompt: port-check
shell-prompt: port-remake
```

Thorough port testing:

```
shell-prompt: port-poudriere-setup
ZFS pool []: (just hit enter)
Configuration file opens in vi, accept defaults.
shell-prompt: wip-poudriere-test hello all
```

The port-poudriere-setup script will create a basic poudriere setup and a FreeBSD jail for building and testing ports on the underlying architecture and operating system. It also offers the option to create additional jails for older operating systems and lower architectures (i386 if you are running amd64).

The wip-poudriere-test script runs "poudriere testport" on the named port in the wip collection.

Other useful poudriere commands:

```
shell-prompt: poudriere ports -u

Updates the ports tree used by poudriere.  This will obsolete any binary
packages saved from previous builds if the corresponding port is upgraded.
Hence, your next poudriere build may take much longer.

shell-prompt: poudriere bulk wip/hello

This will build a binary package for the named port, which you can deploy
with "pkg add" on other systems.
```

Run "poudriere" or "poudriere <command>" or "man poudriere" for help.

Example 2: https://github.com/cdeanj/snpfinder

```
shell-prompt: cd /usr/ports/wip
shell-prompt: port create snpfinder

USE_GITHUB=yes
GH_ACCOUNT=cdeanj
DISTVERSION=1.0.0

shell-prompt: cd snpfinder
shell-prompt: wip-edit
shell-prompt: port-patch-vi work/snpfinder-1.0.0
shell-prompt: port-check
shell-prompt: port-remake
```

### 39.4.3 Pkgsrc

Pkgsrc was forked from FreeBSD ports in 1997 by the NetBSD project.

Like everything in the NetBSD project, the primary focus is portability. Pkgsrc aims to support all POSIX environments. Top-tier support for NetBSD, Linux, SmartOS. Strong support for Mac OS X, other BSDs.

Tools analogous to FreeBSD ports, but often less developed. pkglint, stage-qa, pbulk.

http://netbsd.org/~bacon/

url2pkg, fbsd2pkg

```
PKG_DEVELOPER=yes
```

Pkgsrc Guide (both user and packager documentation)

Example: http://acadix.biz/hello.php

Log into a system using pkgsrc (NetBSD, Linux, Mac, etc.)

Install pkgsrc-wip: https://www.pkgsrc.org/wip/ Use auto-admin?

```
shell-prompt: cd /usr/pkgsrc/wip/pkg-dev
shell-prompt: bmake install
```

Install FreeBSD ports and wip on your pkgsrc system: (ports collection is mirrored on Github if you prefer git)

```
shell-prompt: pkgin install subversion
shell-prompt: svn co https://svn.FreeBSD.org/ports/head /usr/ports
shell-prompt: cd /usr/ports
shell-prompt: svn co https://github.com/outpaddling/freebsd-ports-wip.git wip
```

Convert the FreeBSD port to pkgsrc:

```
shell-prompt: cd /usr/pkgsrc/uwm-pkgsrc-wip/fbsd2pkg
shell-prompt: bmake install
shell-prompt: cd ..
shell-prompt: fbsd2pkg /usr/ports/wip/hello your-email-address

You can run the above command repeatedly until the package is done.

shell-prompt: cd hello
shell-prompt: pkg-check
shell-prompt: pkglint -e
shell-prompt: pkglint -Wall
```

Create the package from scratch using url2pkg:

```
shell-prompt: mkdir hello
shell-prompt: cd hello
shell-prompt: url2pkg http://acadix.biz/Ports/distfiles/hello-1.0.tar.xz
```

## 39.5 What's Wrong with Containers?

Absolutely nothing. Containers are great and play many important roles in computing, especially in development and security.

There are problems with the way some people use them, however. As is often the case, containers have become a solution looking for problems. Many people use them because they think it's "cool" or because it's a path of least resistance.

In scientific computing, containers are often used to isolate badly designed or badly implemented software that is otherwise difficult to install outside a container. For example, software build systems that bundle share libraries can cause conflicts with other versions of the same shared library.

**Aside** There is no problem that cannot be "solved" by adding another layer of software. This is never a solution, however, and is generally short-sighted.

Isolating such software in a container will get it up and running and work around conflicts, but with some major down sides:

- It eliminates the motivation to clean up the software, contributing to the de-evolution of software.
- It adds overhead to running the software. ( Many modern containers advertise their low overhead for this reaason. )

Misusing containers in this creates more disorder and complexity where there is already too little IT talent available to manage things well.

# Chapter 40

# Running Multiple Operating Systems

You don't necessarily need to maintain a second computer in order to run Unix in addition to Windows. All mainstream Unix operating systems can be installed on a PC alongside Windows on a separate partition, or installed in a virtual machine (VM), such as Oracle VirtualBox, which is also available for free.

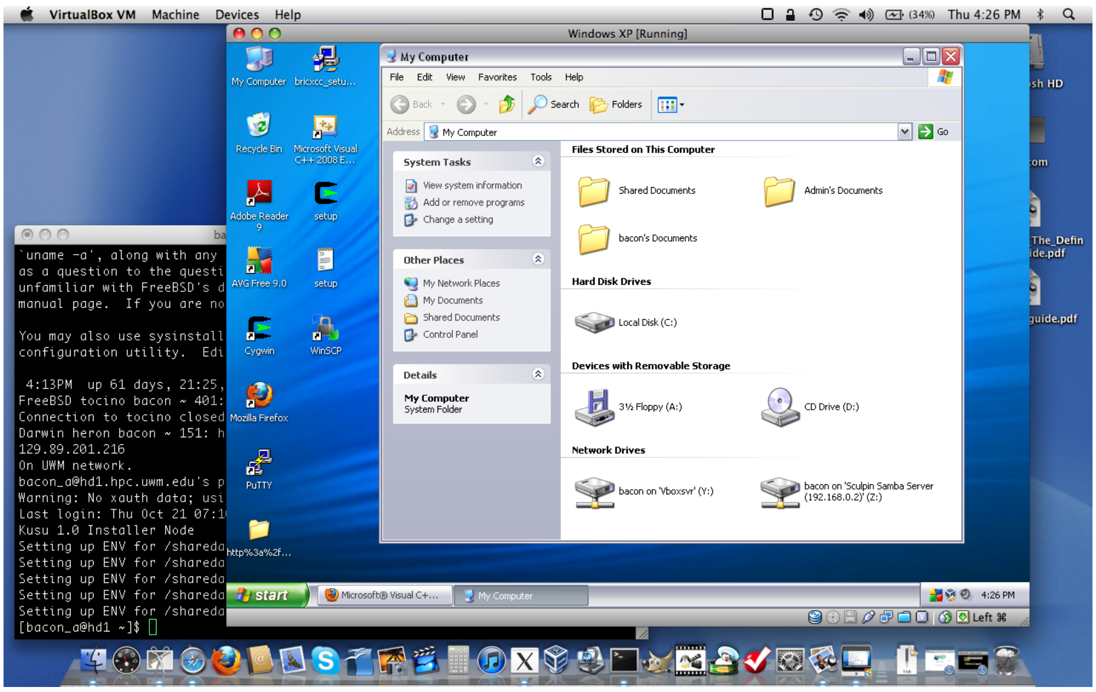
VMs are software packages that pretend to be computer hardware. You can install an entire operating system plus the software you need on the VM as if it were a real computer. The OS running under the VM is called the *guest* OS, and the OS running the VM on the real hardware is called the *host*.

Computational code runs at the same speed in the guest operating system as it does in the host. The main limitation imposed on guest operating systems is graphics speed. If you run applications requiring fast 3D rendering, such as video players, they should be run on the host operating system.

There are many VMs available for x86-based PC hardware, including VirtualBox, http://www.virtualbox.org/, which is free and open source, and runs on many different host platforms including FreeBSD, Linux, Mac OS X, Solaris, and Windows.

Running a Unix guest in a VM on Windows or Windows as a guest under Unix will provide a cleaner and more complete Unix experience than can be achieved with a compatibility layer like Cygwin. The main disadvantage of a VM is the additional disk space and memory required for running two operating systems at once. However, given the low cost of today's hardware, this doesn't usually present a problem on modern PCs.

Virtual machines are most often used to run Windows as a guest on a Unix system, to provide access to Windows-only applications to Unix (including Mac) users without maintaining a second computer. This configuration is best supported, and offers the most seamless integration between host and guest. An example is shown in Figure 40.1.

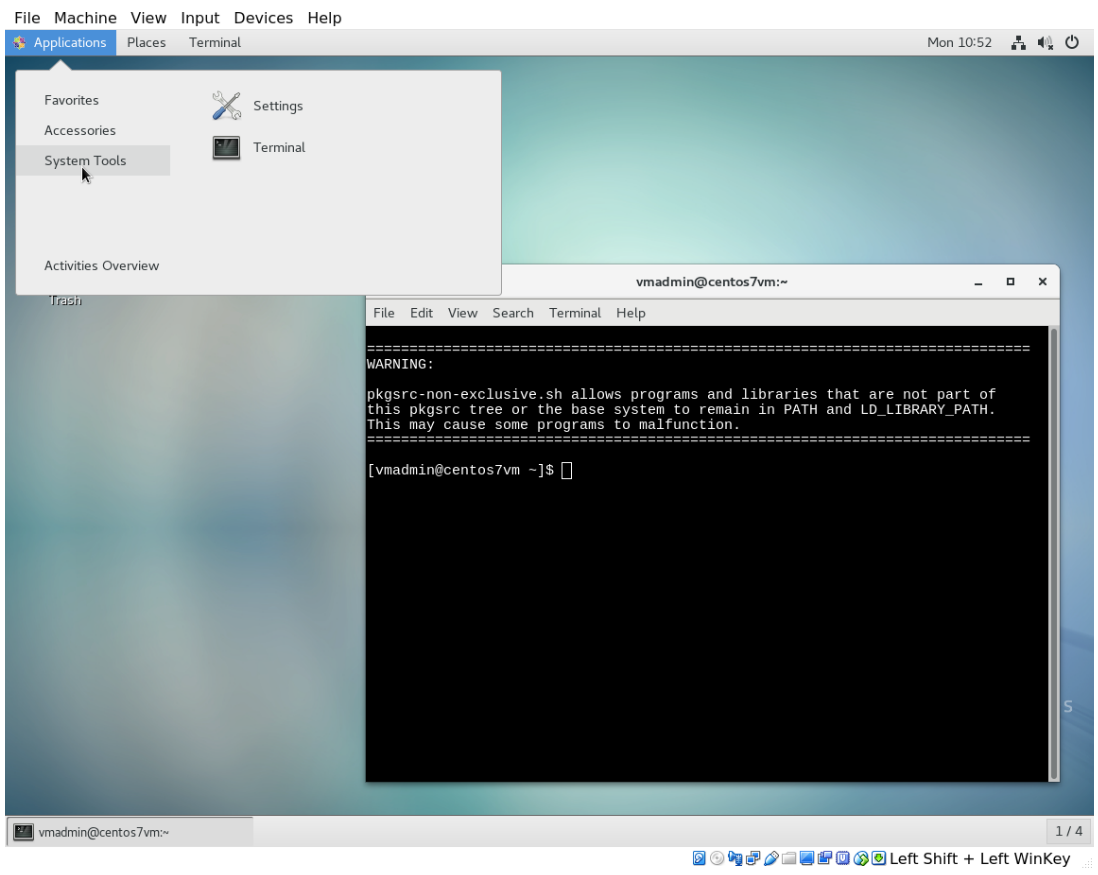


```
`uname -a', along with any
as a question to the questi
unfamiliar with FreeBSD's d
manual page.  If you are no

You may also use sysinstall
configuration utility.  Edi

 4:13PM  up 61 days, 21:25,
FreeBSD tocino bacon ~ 401:
Connection to tocino closed
Darwin heron bacon ~ 151: h
129.89.201.216
On UWM network.
bacon_a@hd1.hpc.uwm.edu's p
Warning: No xauth data; usi
Last login: Thu Oct 21 07:1
Kusu 1.0 Installer Node
Setting up ENV for /shareda
Setting up ENV for /shareda
Setting up ENV for /shareda
Setting up ENV for /shareda
Setting up ENV for /shareda
[bacon_a@hd1 ~]$
```

Figure 40.1: Windows as a Guest under VirtualBox on a Mac Host

Another issue is that Windows systems need to be rebooted frequently, often several times per week, to activate security updates. Most Unix systems, on the other hand, can run uninterrupted for months at a time. (FreeBSD systems will typically run for years, if your power source is that stable.) There are far fewer security updates necessary for Unix systems, and most updates can be installed without rebooting. Rebooting a host OS requires rebooting all guests as well, but rebooting a guest OS does not affect the host. Hence, it's best to run the most stable system as the host.

If necessary, it is possible to run Unix as a guest under Windows. FreeBSD and many Linux distributions are fully supported as VirtualBox guest operating systems.



```
==============================================================================
WARNING:

pkgsrc-non-exclusive.sh allows programs and libraries that are not part of
this pkgsrc tree or the base system to remain in PATH and LD_LIBRARY_PATH.
This may cause some programs to malfunction.
==============================================================================

[vmadmin@centos7vm ~]$ 
```

Figure 40.2: CentOS 7 with Gnome Desktop as a Guest under VirtualBox

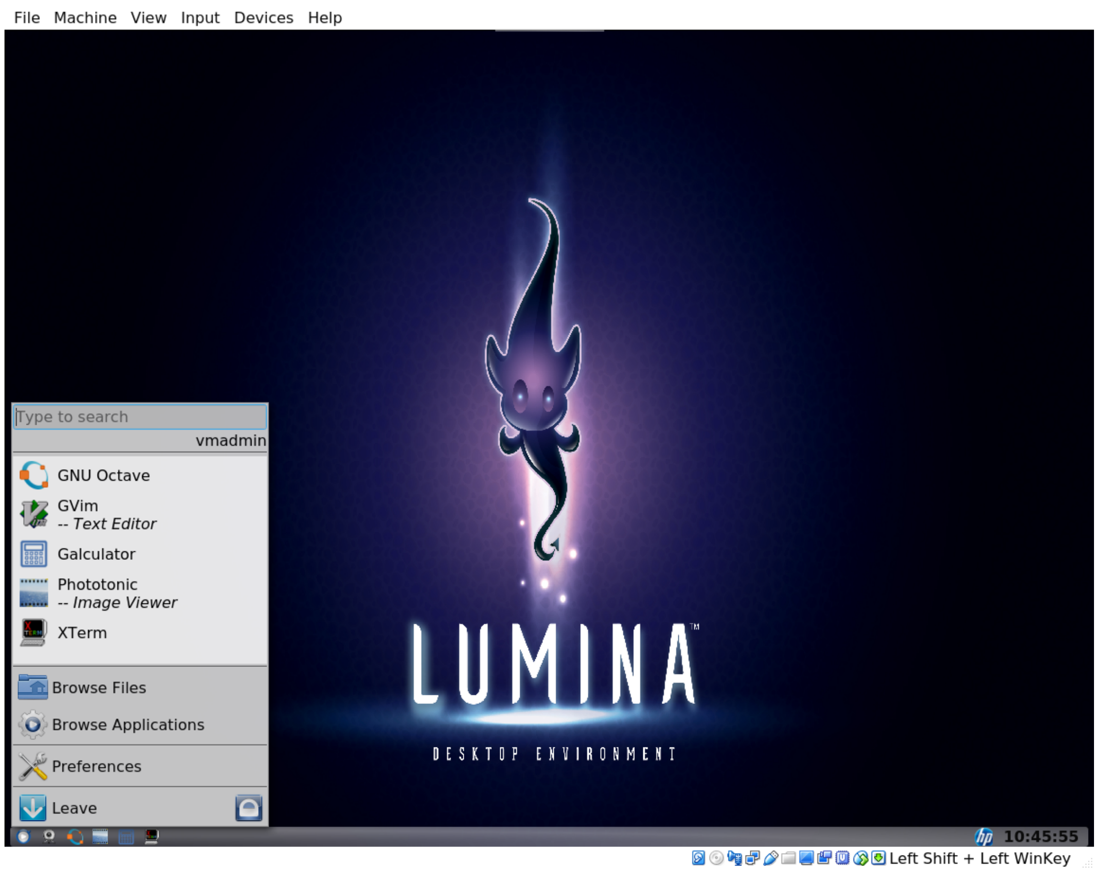


Figure 40.3: FreeBSD with Lumina Dekstop as a Guest under VirtualBox

# Chapter 41

# Index