# C/Unix Programmer's Guide Lecture Outline and Addendum

---

## October 16, 2023

Jason W. Bacon

# Contents


0.1 Best Practices for Students and Instructors . . . 1
0.1.1 Take Ownership of your Education . . . 1
0.1.2 Note to Lecturers . . . 1
0.1.3 Making the Most of Class Time . . . 1
0.1.4 Run the Whole Race . . . 2
0.1.5 Abandon your ego . . . 2
0.1.6 Why do Quality Work? . . . 2
0.1.7 Stick to the course materials . . . 2
0.1.8 Homework, quiz, and exam format . . . 3
0.2 Course Logistics . . . 3
0.2.1 Purpose of this Course . . . 3
0.2.2 Using this Lecture Outline . . . 3
0.2.3 Practice Problem Instructions . . . 4
0.2.4 Remote Unix Servers and the Command Line Interface . . . 5
0.2.5 Programming Assignments . . . 5
0.2.6 Homework due BEFORE the First Lab . . . 7

**1 Introduction** **9**
1.1 How to Proceed . . . 9
1.1.1 Practice . . . 9
1.2 Why use C? . . . 10
1.2.1 C Advantages . . . 10
1.2.2 Language Performance . . . 10
1.2.3 Practice . . . 11
1.3 Why use Unix? . . . 13
1.3.1 Portability . . . 13
1.3.2 Stability . . . 13
1.3.3 Power, Performance, Scalability . . . 13
1.3.4 Inherent Multitasking . . . 13
1.3.5 Simplicity and Elegance . . . 14
1.3.6 Cost . . . 14

1.3.7 Unix Today . . . 14
1.3.8 Addendum: Maturity . . . 15
1.3.9 Practice . . . 15
1.4 Addendum: What is Systems Programming? . . . 15

**I Introduction to Computers and Unix 17**

**2 Binary Information Systems 18**
2.1 Why do I need to know this stuff? . . . 18
2.1.1 Practice . . . 18
2.2 Representing Information in Binary . . . 18
2.2.1 Practice . . . 18
2.3 The Usual Jargon . . . 19
2.3.1 Practice . . . 19
2.4 Binary Number Systems . . . 19
2.4.1 Practice . . . 20
2.5 Binary Fixed Point and Binary Integers . . . 20
2.5.1 Practice . . . 21
2.6 Binary Arithmetic . . . 22
2.6.1 Practice . . . 23
2.7 Signed Integers . . . 23
2.7.1 Practice . . . 25
2.8 Floating Point . . . 25
2.8.1 Practice . . . 25
2.9 Floating Point Range and Precision . . . 26
2.9.1 Practice . . . 26
2.10 Other Number Systems (Bases) . . . 27
2.10.1 Practice . . . 28
2.11 Character Representation . . . 28
2.11.1 Practice . . . 29

**3 Hardware and Software 30**
3.1 What Makes Computers Tick? . . . 30
3.2 The Main Components . . . 30
3.2.1 The CPU . . . 30
3.2.2 Electronic Memory: RAM and ROM . . . 30
3.2.3 Input/Output Devices . . . 31
3.2.4 Mass Storage Devices . . . 31
3.2.5 Practice . . . 31

3.3 Programs and Programming Languages . . . 31
3.3.1 Machine Language . . . 31
3.3.2 Assembly Language . . . 32
3.3.3 High Level Languages . . . 32
3.3.4 Practice . . . 33
3.4 The Programming Process . . . 33
3.4.1 Algorithms . . . 33
3.4.2 Top-down Designs and Stepwise Refinement . . . 33
3.4.3 Flow Charts . . . 34
3.4.4 Practice . . . 35
3.5 Engineering Product Life Cycle . . . 35
3.5.1 Specification . . . 35
3.5.2 Design . . . 35
3.5.3 Implementation and Testing . . . 35
3.5.4 Production . . . 36
3.5.5 Support and Maintenance . . . 37
3.5.6 Hardware Only: Disposal . . . 37
3.5.7 Practice . . . 37

**4 Unix Overview: Enough to Make You Dangerous 38**
4.1 What is an Operating System? . . . 38
4.1.1 Practice . . . 39
4.2 Unix Operating Systems (Was "The Unix Operating System") . . . 39
4.2.1 Addendum: Unix Commands . . . 40
4.2.2 Addendum: Processes . . . 41
4.2.3 Practice . . . 41
4.3 The Unix File-system . . . 41
4.3.1 Partitions . . . 41
4.3.2 Directories . . . 42
4.3.3 Permissions . . . 43
4.3.4 Practice . . . 44
4.4 The Shell Environment . . . 45
4.4.1 Practice . . . 45
4.5 Getting Help . . . 46
4.5.1 Practice . . . 46
4.6 Some Useful Commands . . . 47
4.6.1 Practice . . . 47
4.7 A Few Shortcuts with T-shell and Bash . . . 47
4.7.1 Command History . . . 48

4.7.2 File Specification: Globbing . . . 48
4.7.3 Practice . . . 48
4.8 Unix Input and Output . . . 49
4.8.1 Standard Streams . . . 49
4.8.2 Redirection . . . 49
4.8.3 Pipes . . . 50
4.8.4 Device Independence . . . 50
4.8.5 Practice . . . 50
4.9 Job Control . . . 50
4.9.1 Practice . . . 51
4.10 Shell Variables and Environment Variables . . . 51
4.10.1 Practice . . . 52
4.11 Shell Scripts . . . 52
4.11.1 Practice . . . 53
4.12 Advanced: Make . . . 53
4.12.1 Practice . . . 53

**II Programming in C** **54**

**5 Getting Started with C and Unix** **55**
5.1 What is C? . . . 55
5.1.1 Practice . . . 56
5.2 C Program Structure . . . 56
5.2.1 Practice . . . 57
5.3 A Word about Performance . . . 57
5.3.1 Practice . . . 58
5.4 Some Early Warnings . . . 58
5.4.1 Practice . . . 58
5.5 Coding and Compiling a C Program . . . 58
5.5.1 Coding . . . 58
5.5.2 Compiling . . . 59
Compilation Stages . . . 60
5.5.3 A First Example . . . 61
5.5.4 Handling Errors and Warnings . . . 64
5.5.5 Practice . . . 64

**6 Data Types** **66**
6.1 Introduction . . . 66
6.1.1 Practice . . . 66
6.2 Variables . . . 66
6.2.1 Practice . . . 68
6.3 C's Built-in Data Types . . . 68
6.3.1 Practice . . . 70
6.4 Constants . . . 70
6.4.1 Named Constants . . . 71
6.4.2 Practice . . . 72
6.5 Initialization in Variable Definitions . . . 73
6.5.1 Practice . . . 73
6.6 Choosing the Right Data Type . . . 73
6.6.1 Practice . . . 75
6.7 Creating New Type Names: Typedef . . . 75
6.7.1 Practice . . . 76
6.8 Addendum: Enumerated Types . . . 76

**7 Simple Input and Output** **77**
7.1 The Standard I/O Streams . . . 77
7.1.1 Practice . . . 77
7.2 Single Character I/O . . . 77
7.2.1 Practice . . . 78
7.3 String I/O . . . 78
7.3.1 Practice . . . 79
7.4 Numeric I/O . . . 80
7.4.1 Output with printf() . . . 80
7.4.2 Input with scanf() . . . 81
7.4.3 Practice . . . 83
7.5 Using fprintf() for Debugging . . . 83
7.5.1 Practice . . . 84
7.6 Addendum: Robust I/O . . . 84
7.6.1 Practice . . . 84

**8 C Statements and Expressions** **85**
8.1 Simple Expressions . . . 85
8.1.1 Practice . . . 85
8.2 C Operators . . . 86
8.2.1 Unary and Binary Operators . . . 86

8.2.2 Math Operators . . . 86
8.2.3 Practice . . . 87
8.3 Mixed Expressions . . . 87
8.3.1 Explicit Conversions: Casts . . . 89
8.3.2 Practice . . . 89
8.4 Bitwise Operators . . . 89
8.4.1 Practice . . . 90
8.5 Addendum: More Performance Tricks . . . 91
8.5.1 Polynomial Factoring . . . 91
8.5.2 Practice . . . 92

**9 Decisions with If and Switch 93**
9.1 Program Flow . . . 93
9.1.1 Practice . . . 93
9.2 Boolean Expressions . . . 93
9.2.1 Boolean Type . . . 93
9.2.2 Relations . . . 94
9.2.3 Practice . . . 94
9.3 The if-else Statement . . . 94
9.3.1 Statement Syntax . . . 94
9.3.2 Compound Statements . . . 95
9.3.3 Building Bigger Boolean Expressions . . . 97
9.3.4 De Morgan's Rules . . . 97
9.3.5 Nested `if` Statements . . . 97
9.3.6 Practice . . . 98
9.4 Switch . . . 98
9.4.1 Practice . . . 99
9.5 The Conditional Operator . . . 99
9.5.1 Practice . . . 99
9.6 Performance . . . 99
9.6.1 Programmer Psychology . . . 99
9.6.2 Minimizing Boolean Expression Evaluation . . . 100
9.6.3 Using Data Types to Reduce Boolean Expressions . . . 100
9.6.4 Practice . . . 100

**10 Repetition: Loops** **101**

10.1 Loops and Performance . . . 101
10.1.1 Loops and Performance . . . 101
The Execution Path . . . 101
10.1.2 Performance Measurement . . . 101
10.1.3 Practice . . . 102
10.2 The Universal Loop: while . . . 102
10.2.1 Practice . . . 104
10.3 The do-while Loop . . . 104
10.3.1 Practice . . . 104
10.4 The for Loop . . . 105
10.4.1 Practice . . . 106
10.5 Nested Loops . . . 106
10.5.1 Practice . . . 107

**11 Functions** **108**

11.1 Subprograms for Modularity . . . 108
11.1.1 Practice . . . 108
11.2 Reusability and Encapsulation . . . 109
11.2.1 Practice . . . 109
11.3 Writing Functions . . . 109
11.3.1 Function Definitions . . . 110
11.3.2 Function Calls . . . 112
11.3.3 Practice . . . 112
11.4 Local Variables . . . 113
11.4.1 Practice . . . 113
11.5 Arguments . . . 113
11.5.1 Privacy . . . 113
11.5.2 Argument Passing . . . 114
11.5.3 Promotions in Argument Passing . . . 115
11.5.4 Practice . . . 115
11.6 Library Functions . . . 116
11.6.1 Practice . . . 116
11.7 Documenting Functions . . . 116
11.7.1 Practice . . . 116
11.8 Top-down Programming and Stubs . . . 117
11.8.1 Practice . . . 119
11.9 Advanced: Recursion . . . 119
11.9.1 Practice . . . 120

11.10 Advanced: Scope and Storage Class . . . 120
11.10.1 Scope . . . 120
11.10.2 Storage Class . . . 120
Segments . . . 120
Static . . . 121
Auto . . . 121
Register . . . 122
Const . . . 122
Volatile . . . 123
11.10.3 Practice . . . 123
11.11 Advanced: The inline Request . . . 123
11.11.1 Practice . . . 123

**12 Programming with make** **124**
12.1 Overview . . . 124
12.1.1 Practice . . . 125
12.2 Building a Program . . . 126
12.2.1 Practice . . . 127
12.3 Make Variables . . . 127
12.3.1 Practice . . . 129
12.4 Phony Targets . . . 129
12.4.1 Practice . . . 131
12.5 Using Header Files . . . 131
12.5.1 Practice . . . 132
12.6 Makefile Generators . . . 132
12.6.1 Practice . . . 132

**13 The C Preprocessor** **133**
13.1 Macros and Constants: #define . . . 133
13.1.1 Practice . . . 135
13.2 Functions vs. Macros . . . 135
13.2.1 Practice . . . 136
13.3 Header Files: #include . . . 136
13.3.1 Practice . . . 136
13.4 Advanced: Conditional Compilation . . . 136
13.4.1 #if . . . 136
13.4.2 Preprocessor Operators . . . 137
13.4.3 #ifdef, #ifndef, and defined() . . . 137
13.4.4 Improving Portability . . . 137

13.4.5 Nesting #include Efficiently . . . 138
13.4.6 Practice . . . 138
13.5 Advanced: Other Directives . . . 138
13.6 Advanced: The Paste Operator: ## . . . 139
13.7 Advanced: Predefined Macros . . . 139
13.8 Addendum: Advanced: Variadic Macros (C99) . . . 139

**14 Pointers** **140**
14.1 Pointers: This Stuff is BIG! . . . 140
14.1.1 Practice . . . 140
14.2 Pointer Basics . . . 140
14.2.1 Practice . . . 141
14.3 Defining Pointer Variables . . . 141
14.3.1 Practice . . . 141
14.4 Using Pointers: Indirection . . . 141
14.4.1 Practice . . . 143
14.5 Pointers as Function Arguments . . . 143
14.5.1 Practice . . . 145
14.6 Typedefs and Pointers . . . 145
14.7 Addendum: C99: The `restrict` Pointer Modifier . . . 146

**15 Arrays and Strings** **147**
15.1 One-dimensional Arrays . . . 147
15.1.1 Practice . . . 149
15.2 Arrays and Pointers . . . 150
15.2.1 Pointers Instead of Subscripts . . . 150
15.2.2 Using Subscripts with Pointer Variables . . . 152
15.2.3 Practice . . . 152
15.3 Typedefs and Arrays . . . 153
15.3.1 Practice . . . 153
15.4 Advanced: More Fun with Pointers . . . 153
15.5 Arrays and Functions . . . 153
15.5.1 Practice . . . 154
15.6 Lookup Tables . . . 154
15.6.1 Practice . . . 155
15.7 Pointer Arguments and const . . . 155
15.7.1 Practice . . . 157
15.8 Multi-dimensional Arrays . . . 157
15.8.1 Practice . . . 158
15.9 Addendum: Performance and Portability . . . 158
15.9.1 Practice . . . 160

**16 Dynamic Memory Allocation** **161**

16.1 Dynamic Memory Allocation: `malloc` . . . 161

16.1.1 Practice . . . 161

16.2 Basic Usage . . . 162

16.2.1 Practice . . . 164

16.3 How `malloc()` Keeps Track: Heap Tables . . . 164

16.3.1 Practice . . . 165

16.4 Pointer Arrays . . . 165

16.4.1 Pointer Arrays and Matrices . . . 166

16.4.2 Pointer Arrays and Strings . . . 168

16.4.3 Practice . . . 169

16.5 Command-line Arguments: argc and argv . . . 169

16.5.1 Practice . . . 170

16.6 The Environment: envp . . . 171

16.6.1 Practice . . . 171

**17 Advanced: Function Pointers** **172**

17.1 Simple Function Pointers . . . 172

17.2 Function Pointer Tables . . . 172

**18 Structures and Unions** **173**

18.1 Structures . . . 173

18.1.1 Structure Templates . . . 173

Structure Templates . . . 173

18.1.2 K & R Structure Tags . . . 174

18.1.3 Copying Structures . . . 174

18.1.4 Practice . . . 175

18.2 Pointers to Structures . . . 175

18.2.1 Practice . . . 175

18.3 Structures, Functions, and OOP . . . 176

18.3.1 Practice . . . 177

18.4 Nesting Structures . . . 177

18.4.1 Practice . . . 178

18.5 Lists of Structures . . . 179

18.5.1 Arrays of Structures . . . 179

18.5.2 Linked Lists . . . 179

18.5.3 Pointer Arrays of Structures . . . 180

18.5.4 Practice . . . 180

18.6 Initializing Structures . . . 181

18.6.1 Addendum: Designated Structure Initializers (C99) . . . 181
18.6.2 Practice . . . 182
18.7 Advanced: Unions . . . 182
18.7.1 Unions for Subdividing Objects . . . 182
18.7.2 Unions for Conserving Memory . . . 183
18.7.3 Practice . . . 184
18.8 Advanced: Structure Alignment . . . 184
18.9 Advanced: Bit Fields . . . 185
18.10 Addendum: Advanced: Enforcing OOP in C, Opaque Structures . . . 185
18.11 Addendum: Advanced: Flexible Array Members (C99) . . . 189
18.12 Addendum: Advanced: void pointers . . . 189

**19 Debugging** **190**
19.1 Thinkin' it Through . . . 190
19.1.1 Practice . . . 190
19.2 Making Programs Talk: Debug Code . . . 190
19.2.1 Practice . . . 191
19.3 Unix Debuggers: Which One? . . . 192
19.3.1 Debugger Features . . . 192
19.3.2 Types of Debuggers . . . 192
19.3.3 Practice . . . 192
19.4 The GNU Debugger: `gdb` . . . 192
19.4.1 Running Programs Under `gdb` . . . 192
19.4.2 Using a Core File . . . 193
19.4.3 Practice . . . 193
19.5 Addendum: The LLVM Debugger: `lldb` . . . 193
19.5.1 Running Programs Under `lldb` . . . 193
19.5.2 Using a Core File . . . 193
19.5.3 Practice . . . 193
19.6 Addendum: Valgrind . . . 194

**III Unix Library Functions and Their Use** **195**

**20 Building Object Code Libraries** **197**
20.1 Creating Libraries . . . 197
20.1.1 Practice . . . 197
20.2 Static or Dynamic? . . . 198
20.2.1 Practice . . . 198
20.3 Creating Static Libraries . . . 198
20.3.1 Practice . . . 199
20.4 Creating Dynamic (Shared) Libraries . . . 199
20.4.1 Practice . . . 200

**21 Files and File Streams** **201**

21.1 FILE Streams . . . 201

21.1.1 Practice . . . 201

21.2 The `FILE` Structure . . . 201

21.2.1 Practice . . . 202

21.3 Basic Stream I/O Functions . . . 202

21.3.1 Opening a File: `fopen()` . . . 202

21.3.2 Closing a File: `fclose()` . . . 203

21.3.3 Reading Characters: `getc()` and `fgetc()` . . . 204

21.3.4 Writing Characters: `putc()` and `fputc()` . . . 204

21.3.5 Reading Lines: `fgets()` . . . 205

21.3.6 Writing Strings: `fputs()` . . . 205

21.3.7 Reading Numbers and Formatted Text: `fscanf()` . . . 206

21.3.8 Writing Numbers and Formatted Text: `fprintf()` . . . 206

21.3.9 Detecting End-of-file: `feof()` and `EOF` . . . 206

21.3.10 Stream I/O Examples . . . 206

21.3.11 Binary I/O: `fread()` and `fwrite()` . . . 206

21.3.12 Addendum: Writing Robust Code . . . 207

21.3.13 Practice . . . 207

**22 String and Character Functions** **208**

22.1 Basic String Manipulation . . . 208

22.1.1 Practice . . . 209

22.2 String Functions . . . 209

22.2.1 Copying Strings: `strlcpy()` . . . 209

22.2.2 Duplicating Strings: `strdup()` . . . 210

22.2.3 Finding String Length: `strlen()` . . . 210

22.2.4 Comparing Strings: `strcmp()` and strcasecmp() . . . 211

22.2.5 Concatenating Strings: `strlcat()` . . . 211

22.2.6 Searching Strings: `strstr()` and `strchr()` . . . 212

22.2.7 Building Formatted Strings: `snprintf()` . . . 212

22.2.8 Tokenizing Strings: `strsep()` . . . 212

22.2.9 Addendum: More Information . . . 213

22.2.10 Practice . . . 213

22.3 Classifying Characters: The ctype Functions . . . 213

22.3.1 Practice . . . 214

22.4 Pattern Matching Functions . . . 214

22.4.1 The Regex Functions . . . 214

22.4.2 File Specification Matching: `fnmatch()` and `glob()` . . . 214

22.4.3 Practice . . . 215
22.5 Bulk Memory Manipulation . . . 215
22.5.1 Copying Blocks: `memcpy()` and `memmove()` . . . 215
22.5.2 Comparing Blocks: `memcmp()` . . . 216
22.5.3 Practice . . . 216

**23 Odds and Ends** **217**
23.1 Math Functions . . . 217
23.1.1 Practice . . . 217
23.2 Data Conversion Functions . . . 217
23.2.1 Practice . . . 218
23.3 Random Numbers . . . 218
23.3.1 Practice . . . 219
23.4 Basic Process Control . . . 219
23.4.1 Normal Termination: `exit()` . . . 219
23.4.2 Last Requests: `atexit()` . . . 219
23.4.3 Creating a Core File: `abort()` . . . 219
23.4.4 Practice . . . 219
23.5 Manipulating the Environment . . . 220
23.5.1 Reading the Environment: `getenv()` . . . 220
23.5.2 Writing to the Environment . . . 220
23.5.3 Practice . . . 220
23.6 Sorting and Searching . . . 220
23.6.1 Sorting: `qsort()` . . . 220
23.6.2 Searching: `bsearch()` . . . 221
23.6.3 Practice . . . 221
23.7 Functions with Variable Argument Lists . . . 221
23.7.1 ANSI Form: `stdarg.h` . . . 221
23.7.2 Unix Form: `varargs.h` . . . 221
23.8 Addendum: Advanced: `tgmath.h` (C99) . . . 221

**24 Working with the Unix Filesystem** **222**
24.1 File Information: `stat()` and `fstat()` . . . 222
24.1.1 Practice . . . 222
24.2 Changing File Information . . . 223
24.2.1 Changing Ownership: `chown()` . . . 223
24.2.2 Changing Permissions: `chmod()` . . . 223
24.2.3 Practice . . . 223
24.3 Accessing Directories . . . 224
24.3.1 Reading Directories . . . 224
24.3.2 Creating Directories . . . 224

**25 Low-Level I/O** **225**

25.1 Why Use Low-level I/O . . . 225

25.1.1 Cat: A Bad Example . . . 225

25.1.2 Practice . . . 226

25.2 Basic Input and Output . . . 226

25.2.1 Opening Files: `open()` . . . 226

25.2.2 Reading Files: `read()` . . . 227

25.2.3 Writing Files: `write()` . . . 228

25.2.4 Closing Files: `close()` . . . 228

25.2.5 Moving within a File: `lseek()` . . . 228

25.2.6 Cat: A Better Example . . . 228

25.2.7 Choosing the Right Buffer Size . . . 229

25.2.8 Handling Multiple Files: `select()` . . . 229

25.2.9 Controlling File Descriptors: `fcntl()` . . . 230

25.2.10 Practice . . . 230

**26 Controlling I/O Device Drivers** **231**

26.1 Termios . . . 231

26.1.1 Input Flags . . . 231

26.1.2 Output Flags . . . 231

26.1.3 Control Flags . . . 231

26.1.4 Local Flags . . . 231

26.1.5 Control Characters . . . 231

26.1.6 The Termios Functions . . . 231

26.1.7 Alternatives to the Termios Interface . . . 231

Other Low-evel Methods . . . 231

High-level Libraries . . . 231

**27 Unix Processes** **232**

27.1 Creating Processes . . . 232

27.1.1 Creating Processes: `fork()` . . . 232

27.1.2 Transforming Processes: `execve()` . . . 233

27.1.3 Waiting for Godot: `wait()` . . . 234

27.1.4 A Complete Example . . . 234

27.1.5 Addendum: The POSIX Spawn Interface . . . 235

27.1.6 Practice . . . 236

27.2 Redirection . . . 237

27.2.1 Simple Redirection . . . 237

27.2.2 Redirection and Restoration . . . 237

27.2.3 Practice . . . 238

**28 Interprocess Communication (IPC) 239**

28.1 The Environment . . . 239
28.1.1 Practice . . . 239
28.2 Signals . . . 239
28.2.1 Practice . . . 241
28.3 Pipes . . . 241
28.3.1 Practice . . . 242
28.4 Sockets . . . 243
28.4.1 Practice . . . 245
28.5 Shared Memory . . . 245
28.5.1 System V Shared Memory . . . 246
28.5.2 Process Synchronization: Semaphores . . . 247
28.5.3 Practice . . . 247

**29 Threads 248**

29.1 Overview . . . 248
29.1.1 Practice . . . 249
29.2 Addendum: POSIX Threads . . . 249
29.2.1 Practice . . . 249
29.3 Addendum: OpenMP . . . 249
29.3.1 Practice . . . 251

**30 Unix Graphics: X Windows 252**

30.1 How X11 Works . . . 252
30.1.1 The X Server . . . 252
30.1.2 X Clients . . . 252
30.1.3 Addendum: DRI . . . 252
30.2 Programming with Xlib . . . 252
30.3 Programming with the Xt Toolkit . . . 252
30.4 Addendum: OpenGL 3D Graphics . . . 252
30.5 Addendum: QT, GTK . . . 252

**IV The C++ Programming Language 253**

**31 Introduction to C++ 254**

**32 Index 255**

# List of Figures

1 Using SSH to log into a Remote Server . . . 6
2 GlobalProtect-OpenConnect VPN Client . . . 6
3 APE IDE . . . 7

3.1 Music box (wonderhowto.com) . . . 32
3.2 Top-down Design for Selection Sort . . . 33
3.3 Flow Chart . . . 34

4.1 Standard Streams . . . 49
4.2 Redirection . . . 49

5.1 APE Build Menu . . . 59
5.2 Compilation . . . 60

10.1 The top Command . . . 102

16.1 Pointer Array . . . 166
16.2 Arguments to cc . . . 170

# List of Tables

1.1 Selection Sort of 200,000 Integers . . . 12
1.2 Partial List of Unix Operating Systems . . . 14

2.1 Range of Common Unsigned Binary Integer Systems . . . 22
2.2 Range of Common Unsigned Binary Integer Systems . . . 24
2.3 IEEE Floating Point Range and Precision . . . 26
2.4 Binary/Octal Conversion . . . 27
2.5 Binary/Hex Conversion . . . 28

3.1 A Hypothetical Machine Instruction . . . 31

6.1 C Data Types . . . 69
6.2 Execution Time of a Loop with Various Types . . . 74

7.1 Format specifiers for printf() . . . 80
7.2 Format specifiers for scanf() . . . 82

8.1 Basic Math Operators in C . . . 86
8.2 Data Type Ranks . . . 88
8.3 Bitwise Operators in C . . . 89

9.1 Relational Operators . . . 94
9.2 Boolean Operators . . . 97

11.1 Memory Map . . . 112
11.2 Argument Passing . . . 114
11.3 Machine Language Program Segments . . . 120
11.4 Memory Map of Local Variables . . . 122

12.1 Standard Make Variables . . . 129

14.1 Memory map . . . 142
14.2 Variable contents at the start of swap() . . . 144
14.3 Variable contents at the end of swap() . . . 144

14.4 Variable contents at the start of swap() . . . . . 144
14.5 Variable contents at the end of swap() . . . . . 145

15.1 The Memory Hierarchy . . . . . 159

22.1 RE Patterns . . . . . 214

25.1 Mode bits for open() . . . . . 227

October 16, 2023

## 0.1 Best Practices for Students and Instructors

### 0.1.1 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.

### 0.1.2 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.

### 0.1.3 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.

### 0.1.4 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.

### 0.1.5 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.

### 0.1.6 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.

### 0.1.7 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.

### 0.1.8 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.

## 0.2 Course Logistics

### 0.2.1 Purpose of this Course

This is a course about Unix systems programming in C. You will learn about Unix, the only existing set of standards for portable operating systems. You will then learn the C programming language with the aim of maximizing performance and portability. Finally, you will learn how to write C programs utilizing Unix standard libraries, which can be run on virtually any modern hardware and operating system.

Students are expected to have some experience with programming, such as one or two semesters of general programming at the college level. It is assumed that students have never touched a Unix system and do not know what binary is. As such, they should not be expected to get into advanced topics such as sockets during this semester. Learning the Unix command line, the basics of number systems such as binary, octal, and hexadecimal, and the C programming language will fill most of the semester. There should be some time (at least a couple weeks) remaining at the end to introduce Part III of this text, the standard libraries, and teach the concepts of creating processes and threads and communicating via pipes, or maybe shared memory.

### 0.2.2 Using this Lecture Outline

First, be aware that this document is updated frequently. Do not print it, as it will likely be obsolete before the print job finishes. Get the latest revision every time you read to ensure that you have the latest material and revisions.

The chapters and sections in this outline use the same names as corresponding chapters and sections in the book. They are also in the same order to make things easy to find. This outline is your guide to which material from the book you are expected to know for quizzes and exams. Material here is presented in an abbreviated format more detailed than a slide presentation, but far less verbose than the book.

---

**Note** This is only an outline. You must read the corresponding sections in the book in order to learn the required material.

---

This outline is also an addendum to the book wherever something has changed since the book's publication, or to clarify things that have not changed but could have been explained better in the book. C and Unix are extremely stable products, so there are very few differences of interest between the publication date in 1999 and today. The few important differences are noted here.

This outline does not contain a complete record of what happened in lecture and are not a substitute for attending class or reading the book. Students are expected to know all material covered in class, which may include material and details found neither in this outline nor in the book.

Students are strongly advised to read this outline and the book sections BEFORE each class. This will enable you to get much more out of lecture as well as contribute to the conversation. Thoroughly study this outline and your hand-written notes from lecture to prepare for quizzes and exams.

Use the examples in the book as practice. Don't just read them: DO them. Type them in and run them, modify them, write similar programs just for curiosity's sake.

Homework questions include material covered in lecture and material that is only in the book. Read each section immediately before doing the homework for that section, so you can easily find the right answers. This learning process is much like the code implementation process: Write a little code, then *test it* before moving on.

### 0.2.3 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.

### 0.2.4 Remote Unix Servers and the Command Line Interface

There are two general categories of user interfaces:

- Command driven (the user types in commands), also known as a *CLI (Command Line Interface)*
- Menu driven (the user selects from among items on the screen). This includes *Graphical User Interfaces (GUIs)*, where menu items may be icons selected with a mouse rather than just text.

CLIs always have been and always will be an important part of computer science for the following reasons:

- Development of a GUI is an order of magnitude more costly and hence not feasible in many areas, such as scientific computing, where funding and programmer talent are limited.
- GUIs are not usually feasible for using remote servers over slow network connections such as WiFi, home Internet connections, or long distances. Access to virtual machine instances on cloud servers is often entirely CLI-based.
- GUIs are preferable for simple systems such as an ATM (automatic teller machine), but cumbersome for complex systems with too much functionality to display on one screen. The user must then navigate a system of submenus to find the functions they need. For this reason, *CAD (Computer Aided Design)* programs, which are intensively graphical by nature, include a CLI. Experienced CAD users prefer the CLI because it allows them to manipulate the design much faster.

As part of this course, you will learn to use the Unix command line, possibly via *SSH* (*Secure Shell*). This experience will be essential to anyone working with *HPC* (*High Performance Computing*) clusters or certain types of cloud services in the future. While there are various user interfaces for working with remote computers, the command line is the only available interface for many tasks. Mastery of the Unix command line will open many additional doors during your career as a computer scientist. A small sample of potential careers are listed below.

- Unix Systems Programmer: Involves knowledge of Unix systems management, shell scripting, possibly other languages such as Perl, Python, and C. Many jobs available in both private sector and academia. Academic jobs tend to pay less, but are usually more interesting and offer more flexibility. Potential employers include Google and Yahoo (largely Linux-based), Netflix (largely FreeBSD-based), most universities, national laboratories, etc.
- Bioinformatics: Considered the wild west of scientific computing, this is a rapidly growing field dealing with big data issues. There are many opportunities in academic research as well as biotech companies. Expect a high salary and very challenging work.
- HPC Systems Manager: Requires advanced skills in Unix systems management and knowledge of high-speed networking technologies. Opportunities exist in academia and the private sector.
- HPC Facilitator: Facilitators train and assist end-users of HPC resources. They are experts in *using* HPC clusters, though not necessarily in managing them.

### 0.2.5 Programming Assignments

There will be several programming assignments throughout the semester, which will require putting multiple concepts together. The weekly homework will help you master these basic concepts. If you do the homework properly, then the programming assignments will simply be exercises in integrating those concepts.

Programs are graded on strict quality measures (well-chosen identifier names, code indentation and other formatting, meaningful comments, etc.) and execution (correct output for correct input, graceful handling of incorrect input).

To use a remote Unix server, you will need a terminal emulator and an **ssh** client to access the remote server from a PC (see Figure 1).

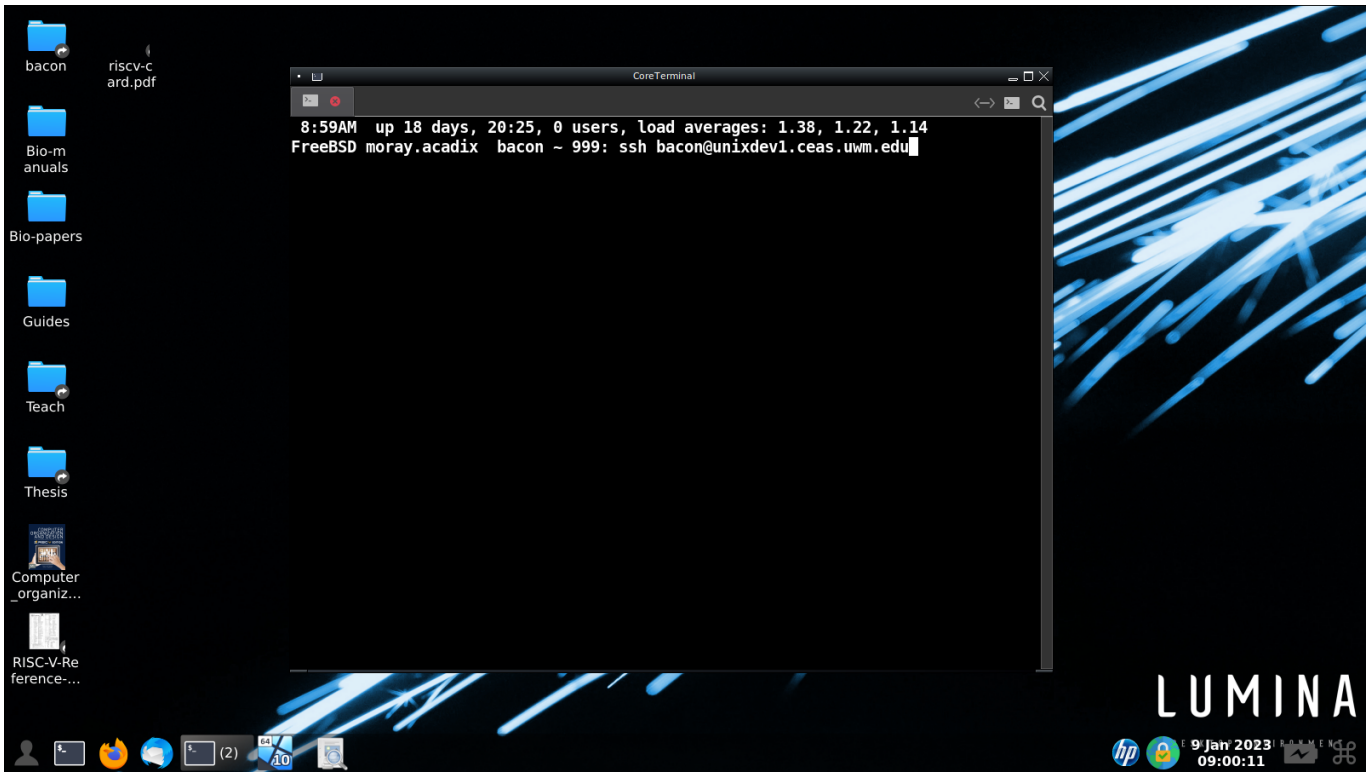



Figure 1: Using SSH to log into a Remote Server

Access to the remote server from off campus may require a VPN client. You can use the commercial VPN client provided by your IT department, or the open source **OpenConnect** or GlobalProtect OpenConnect, which can be installed via Debian packages and FreeBSD ports (see Figure 2), and possibly other package managers.

---

**Note** GlobalProtect OpenConnect *daemonizes* a program called **gpservice** to maintain the connection to the VPN server. As a daemon, **gpservice** continues to run after the GUI **gpclient** is terminated. If left running for a long time, the connection can go *stale*, resulting in failed connections. You may need to manually terminate the daemon by running **pkill gpservice** as root, so that a new VPN connection is established next time you run **gpclient**.

---


Figure 2: GlobalProtect-OpenConnect VPN Client

Note that testing on multiple platforms is a good idea and should always be done for professional programming. However, testing

on additional platforms almost always reveals more bugs. If you do not have time to debug your programs twice, develop them on the remote server provided for this course.

A text-based editor such as emacs or a text-based IDE (Integrated Development Environment) such as APE (Another Programmer's Editor, Figure 3) will work well when programming remotely over an SSH connection. Some graphical editors such as Eclipse will likely be sluggish.

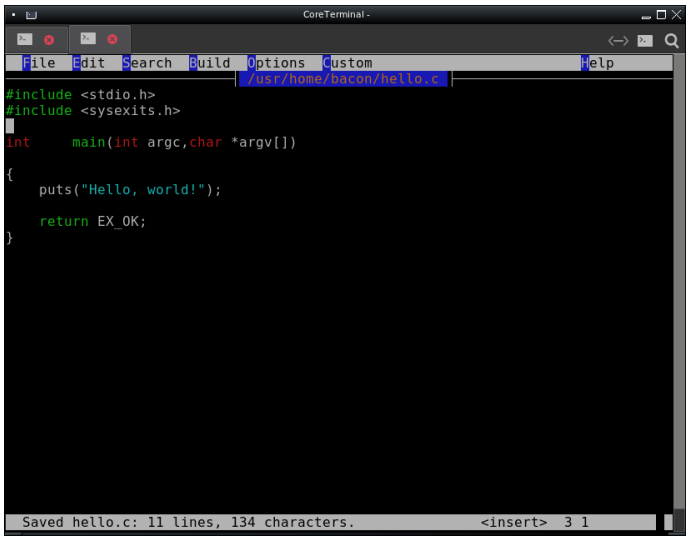



Figure 3: APE IDE

### 0.2.6 Homework due BEFORE the First Lab

For courses utilizing a remote Unix server, students should have a working terminal emulator and SSH client on their laptop BEFORE the first lab. Your successful login to the Unix server will be checked by the instructor during the first lab session.

**Caution**
THIS MUST BE DONE BEFORE THE FIRST LAB. It cannot be done during lab, since multiple students downloading packages at the same time would overwhelm the WiFi access point. The task is very simple, but involves downloading a significant volume of program files for Windows users.
Do this assignment IMMEDIATELY and ask for help if you run into any problems. Instructors will be available to help.

Students MUST have a working terminal emulator and SSH client on their laptop BEFORE the first lab session. Before your first lab, verify that you can open a terminal window on your laptop and run the **ssh** command. Just type "ssh" into the terminal window and you should see something like the following:

```
shell-prompt: ssh
usage: ssh [-46AaCfGgKkMNnqsTtVvXxYy] [-B bind_interface]
           [-b bind_address] [-c cipher_spec] [-D [bind_address:]port]
           [-E log_file] [-e escape_char] [-F configfile] [-I pkcs11]
           [-i identity_file] [-J [user@]host[:port]] [-L address]
           [-l login_name] [-m mac_spec] [-O ctl_cmd] [-o option] [-p port]
           [-Q query_option] [-R address] [-S ctl_path] [-W host:port]
           [-w local_tun[:remote_tun]] destination [command [argument ...]]
```

If you see something like "ssh: command not found", then you need to install an SSH package.

Most BSD and GNU/Linux distributions, and macOS come with terminal emulator and SSH preinstalled. Windows users can install them as part of Cygwin in about 15 minutes, following the instructions in the "Connecting to the Remote Server" section of the lab manual provided on Canvas (c-unix-lab.pdf).

Also BEFORE the first lab, be sure to verify that you can connect to WiFi on campus.

This assignment should not take much more than 15 minutes, but no task is too simple to fail, so do it LONG BEFORE YOUR LAB SESSION so that you will have time to look for a solution or ask for help before it is too late.

---

**Note** All assignments for CS 337 are to be done individually, except this one. The assignment in this case is simply to have a terminal emulator and **ssh** command on your laptop before the first lab session. If someone helped you do this, that's fine as far as credit for the assignment, though it may be a sign that you need to up your game. This should be easy for any computer science student, following the instructions in the lab manual provided on Canvas (c-unix-lab.pdf). Better to ask an instructor or a friend for help than to fail to complete the assignment, though.

---

# Chapter 1

# Introduction

**Note** The introduction in the book contains many dated examples, e.g. OS/2, Windows 95, Windows NT, Borland C. However, the concepts illustrated by these examples still apply to their more modern equivalents. In most other parts of the book, I took care to avoid using dated material, and very little has changed since the book's publication.

## 1.1 How to Proceed

Unix "man pages" (demonstrate) contain a wealth of detailed information on Unix commands and C functions.

They are written to serve as references, rather than tutorials, so a book is a much better way to get started. After reading the book, you should be ready to tackle the man pages.

Components of each chapter:

- What to read first (prerequisite knowledge)
- Essentials
- Related performance discussion
- Related portability discussion
- Advanced topics (only some will be covered in this class)

### 1.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Why do we need a book when we can just learn Unix and C functions from the man pages?
2. What is the best way to utilize the examples in the book?
3. Where does proficiency come from?

## 1.2 Why use C?

### 1.2.1 C Advantages

- C is Fast: It is the fastest among portable high-level languages, nearly as fast as hand-optimized assembly language.

  Addendum to book: C++ code performance has improved since publication of the book. It now rivals the speed of C and outperforms Fortran and Pascal in many cases.

  Table 1.1 shows the relative performance of numerous languages

- C is powerful: Literally any code can be written in C, including code that could only otherwise be done in assembly language, which is not portable. We can also write complex application programs in C.

  C is sometimes criticized for being too "low-level", suggesting that we have to write more code to achieve the same thing as a high-level language. This is not generally true, however. High-level features in other languages can usually be replaced by a simple library function call in C. Whether a feature is part of the language or part of a library makes no difference to how easy it is to use.

```
% In Matlab, matrix multiplication is built into the language
product = matrix1 * matrix2;
```

```
// In C, matrix multiplication is not supported by the
// language, but there are many prewritten libraries
// containing functions that we can use
matrix_multiply(product, matrix1, matrix2);
```

- C is Portable: C code will run on more hardware platforms than any other language, from small embedded controllers to massive supercomputers.

- C is small and flexible. The design philosophy was not to include any feature in the language itself that can be reasonably well provided by a library function. As a result, the C language is very small and easy to master. Growth as a C programmer then shifts to learning about the vast number of available libraries used in your field.

  C popularized the *small and extensible* philosophy, which aims at keeping software systems simple, while allowing the capabilities to be easily extended by separate modules. In the case of C, the separate modules are library functions. Other software may use *plugins*, but the concept is the same.

- C is popular: C is among the most popular languages in terms of the number of programs written in each. Most modern languages are based on C. Hence, learning C gets you started with C++, Java, etc.

- C is stable: Code written in many languages ceases to function after several years as language features are deprecated and eventually removed. C, on the other hand, has been stable for decades and is unlikely to change in the future in any way that will break existing programs. It is a WORF (write once, run forever) programming language.

You *can* do object-oriented programming in C. In fact, an object-oriented design can be implemented in *any* language, and it can be done easily in any language with structures and type definitions. We will see how after we cover structures. Most features of object-oriented languages (e.g. overloading, inheritance, templates, etc.) are conveniences rather than necessities for OOP.

### 1.2.2 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 1.1 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.

### 1.2.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How much of the original Unix operating system was written in C?
2. List 5 advantages of C over other high-level languages.
3. Is C too low-level for application programming?
4. Can you do object-oriented programming in C?

| 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 1.1: Selection Sort of 200,000 Integers

5. How fast is a C program compared to the same program design implemented in an interpreted language such as Python, Perl, or R?
6. Why is C so much simpler than other high-level languages?
7. Is it enough that a program runs fast enough on your computer? Why or why not?
8. What is the relationship between C and C++?
9. Does C++ enforce object-oriented programming?

## 1.3 Why use Unix?

### 1.3.1 Portability

Unix is the only set of open standards to which operating systems conform. It began as an operating system at AT&T Bell Labs around 1970, but quickly became a model for all operating systems that followed. Today, the term "Unix" refers to every operating system you are likely to use except Microsoft Windows.

Code written for Unix systems will run on virtually any platform, including Windows with the aid of a compatibility layer such as Cygwin, or a Virtual machine (VM) running a Unix-compatible system. Code written for Windows will generally only run on Windows. You can run Windows in a VM on a Unix system, but this does require purchasing a Windows license for each VM.

Unix systems are nearly 100% *source compatible* with each other. The Unix kernel encapsulates the hardware, so most Unix programs are not even aware of the type of CPU on which they are running.

POSIX (Portable Operating System Standards based on Unix), is an official standard to which Unix-like systems conform. Code written to the POSIX standard will generally run without modification on any Unix-like system. Most modern Unix-like operating systems are more than 99.9% POSIX-compliant.

### 1.3.2 Stability

Since the Unix kernel encapsulates the hardware, user programs cannot access most hardware directly and therefore cannot cause a system crash or interfere with the operation of other programs. As a result, Unix systems rarely need to be rebooted. FreeBSD is particularly robust, and systems have been known to run for years without a reboot. ( This is not a good idea, though, since it means the kernel is not being updated. )

Since it is so easy to switch from one Unix platform to another, Unix OS developers have to compete with each other for your business based on objective features such as performance and reliability. This is another reason that Unix systems all tend toward reliability.

### 1.3.3 Power, Performance, Scalability

Unix now runs on everything from small embedded devices and cell phones to supercomputers. Android OS is based on Linux, while macOS and iOS are based on FreeBSD.

In the 1980s, Unix was too big to run on a typical PC, so DOS was created to provide something conceptually similar, but much smaller. When 32-bit PCs became popular in the 1990s, PC-based Unix became feasible. Today, the roles have reversed. Windows 10 and 11 require far more memory and disk than most Unix systems.

---

**Note** I don't recall the last time I paid more than $150 for a computer. FreeBSD runs faster on 5 or 10 year old PC than modern Windows does on brand new hardware, so I just buy used PCs online for all my personal needs.

---

### 1.3.4 Inherent Multitasking

Unix was designed as a multiuser, multitasking system from the start, so security and resource sharing are fully integrated. These features were afterthoughts in MS Windows, which evolved from the single-user, single-process MS DOS.

### 1.3.5 Simplicity and Elegance

Unix designers followed the KISS principle: Keep it simple, stupid. Not every Unix feature will seem intuitive at first, but you will likely find that it is more elegant than what you would have done.

### 1.3.6 Cost

The majority of Unix systems running today are either BSD or Linux based. Most BSD and Linux systems are *FOSS* (*Free, Open Source Software*), so we can install and use them free of charge.

### 1.3.7 Unix Today

The BSD family of operating systems have direct lineage from the original AT&T Unix, though most of the code in them today came from other sources. The most popular pure BSD operating system today is FreeBSD. It is heavily used in networking and storage applications, including the Netflix content delivery network, Juniper networking products, pfSense and OPNsense firewalls, TrueNAS and XigmaNAS storage appliances, and many others. https://en.wikipedia.org/wiki/List_of_products_based_on_FreeBS Apple's macOS and iOS are also derived from FreeBSD.

Linux is a kernel (not an operating system) that was developed mostly independently as a free Unix clone in the 1990s. The name is derived from the project founder's name (Linus Torvalds) + Unix. Linux-based operating systems usually combine the Linux kernel and GNU project user tools (userland), including basic Unix commands, compilers, etc. Some common GNU/Linux operating systems include Debian, Redhat Enterprise, and Ubuntu. There are dozens more and the landscape is constantly changing. Linux is also the basis of many products, including Android OS.

Table 1.2 provides a partial list of Unix systems in use today.

| 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 1.2: Partial List of Unix Operating Systems

### 1.3.8 Addendum: Maturity

Modern FOSS (Free Open Source Software) Unix platforms have come of age. They now provide everything a typical computer user needs for personal use and much more, all completely free of charge. Free web browsers such as Firefox are in many ways more capable than closed source browsers such as Apple's Safari and Microsoft Edge. LibreOffice can easily replace Microsoft office for typical users. GNU Image Manipulation Program (GIMP) rivals PhotoShop's capabilities. Thunderbird email client can be a drop-in replacement for MS Outlook. There are now thousands of high-quality open source applications for most common computer uses. Commercial operating systems are now only required by those with esoteric needs that are only served by commercial applications that most people will never use.

All of these powerful applications can be run on numerous free operating systems that can be installed in about an hour by someone with relatively modest computer skills.

This was not true in the late 1999 when the first edition of the book was published. In those days, running Unix at home was only feasible for computer professionals.

Poll: How many students in the class have heard of Firefox? LibreOffice? GIMP? Thunderbird? How many students are running an open source operating system for personal use?

### 1.3.9 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Is Unix an operating system?
2. Which mainstream operating systems are Unix compatible?
3. If you write a program on a Linux system with an Intel x86 processor, how hard will it be to port the program to FreeBSD on an ARM processor?
4. What is POSIX?
5. What types of devices run Unix?
6. How much does a Unix license typically cost?
7. What is the major benefit of using Unix and C together?
8. How does the Unix kernel make it possible for programs written on a PC to run on a RISC workstation or server?

## 1.4 Addendum: What is Systems Programming?

There is no clear definition of systems programming, but in general the term serves to contrast *applications programming*. In applications programming, we write software used directly by the end-user, who may or may not be a computer expert.

Systems programming generally refers to writing software that is not used directly by average users. It may include special tools used by computer experts, libraries and tools used by application programs but unknown to the application user, or parts of the operating system.

Learning systems programming generally involves a few major topics:

- Learning C, which is the primary systems programming language for most POSIX (Portable Operating System standards based on UnIX) systems.
- Learning how C and other languages work under the hood, so that we can write highly efficient and correct systems software. Efficiency is almost always important, but is paramount in systems programming, since systems code may be utilized by many or all applications. We also don't want systems code competing for resources with applications. Systems should leave as much CPU and memory as possible available to the applications that need them.

- Learning how to build and deploy systems programs and libraries in a clean and reproducible manner, usually using established tools for a given operating system. Systems code may be used on millions of installations and therefore must be easy to deploy and reliable.
- Learning how to interface with operating system services, such as file operations, process creation, interprocess communication, networking, etc.
- Learning how to document the tools we create using standard documentation tools and formats, so that application developers have easy access to the information they need.

# Part I

# Introduction to Computers and Unix

# Chapter 2

# Binary Information Systems

## 2.1 Why do I need to know this stuff?

Software controls hardware. Understanding the limitations of hardware allows us to write programs that produce correct results and run as fast as possible.

Computer hardware represents all information in *binary*, combinations of information packets with only *two* states.

The limitations of hardware can only be understood by understanding binary information systems.

This topic is covered in more depth in a computer architecture or assembly language course. It is introduced more briefly here so that we can fully understand the features of the C language.

### 2.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Why is it important for programmers to understand number systems?

## 2.2 Representing Information in Binary

Any information can be represented in binary, as a combination of elements with two possible states. Conceptually, we use the digits 0 and 1. Computer hardware uses two different voltages, such as 0V for 0, and 3V for 1.

The decimal number system uses elements with 10 possible states, 0 through 9. The English alphabet uses elements with 26 possible states, A through Z.

### 2.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is binary?
2. How do we represent binary information on paper?
3. How is binary represented inside a computer?
4. Why do computers use binary rather than decimal?

## 2.3 The Usual Jargon

- Bit = Binary digit (a bit of a misnomer, since not all bits are actually part of a number)
- Byte = 8 bits
- Nybble = 4 bits
- Word = maximum number of bits a computer can process at once. Usually 16, 32, or 64.
- Book addendum: Long word was usually 32 bits for both 16 and 32-bit computers, which was pretty much all computers when the book was published. Usually means 64 bits on 64-bit computers.
- Short word usually means 16 bits, regardless of word size.

### 2.3.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Define each of the following:
   (a) Bit
   (b) Byte
   (c) Word
   (d) Longword
   (e) Shortword
   (f) Nybble

## 2.4 Binary Number Systems

Binary number systems are basically the same as decimal number systems, but use a base of 2 rather than 10. Both binary and decimal use the "Arabic" system of weighted digits.

There are several ways to define an Arabic numeral system with a limited number of digits:

*Fixed point* number systems have a fixed number of whole digits and a fixed number of fractional digits. A decimal system with two whole digits and three fractional digits can represent numbers from 00.000 to 99.999. It is called fixed point because the position of the decimal (or binary) point is fixed, i.e. it cannot move.

An *integer* system is simply a fixed point system with no fractional digits. Integer number systems are highly useful in computer programming due to their efficiency. (Review your notes from 1st grade to refamiliarize yourself with integer arithmetic.)

A *floating point* system allows the decimal (or binary) point to move. This makes the system far more flexible and increases both range and precision.

Floating point numbers are represented internally in a form similar to scientific notation, $\texttt{mantissa * radix}^{\texttt{exponent}}$. As a result, floating point operations take about 3 times as long as integer operations. For example, floating point addition requires the same three steps as scientific notation addition:

- Equalize the exponents
- Add the mantissas
- Normalize the result

### 2.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What are the differences between binary numbers and decimal numbers?
2. What do binary and decimal number systems have in common?
3. What is the main difference between the abstract number sets we use in mathematics and the number sets represented in computers?
4. What is a fixed point number system?
5. What is an integer number system?
6. What is a floating point system?
7. What is a major disadvantage to floating point as compared to integers in computer systems?
8. How are floating point systems implemented in computers?
9. What are the three parts of a floating point number?
10. What steps are necessary to add two floating point numbers?

## 2.5 Binary Fixed Point and Binary Integers

The Arabic system is a weighted-digit system with each digit multiplied by a power of the base (radix). The digits just left of the period always has a power of zero. We can use a subscript or a '_' to indicate the base, e.g. $56.1_{10}$ or 56.1_10.

```
972.81_10 = 9 * 10^2 + 7 * 10^1 + 2 * 10^0 + 8 * 10^-1 + 1 * 10^-2

101.01_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^-1 + 1 * 2^-2
         = 4 + 0 + 1 + 0 + 1/4
         = 5.25_10
```

Digits in any Arabic system range from 0 to base-1.

Binary numbers are particularly easy to read, since we are always multiplying by either 0 or 1. In fact, it's a waste of time to even write "1 *" or the whole expression "0 * 2^x":

```
101.01_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^-1 + 1 * 2^-2
         = 2^2 + 2^0 + 2^-2
         = 4 + 1 + 1/4
```

Converting from binary to decimal by hand is intuitive, since we use decimal in our heads to do the math.

To convert from decimal to binary, we need to figure out what powers of 2 add up to the number.

```
37.75_10 = 32 + 4 + 1 + .5 + .25
         = 2^5 + 2^2 + 2^0 + 2^-1 + 2^-2
         = 100101.11_2
```

To do this formally, we divide by successive powers of 2, starting with the largest one less than the number. The integer quotient is our next digit. We continue at least until we have divided by 2^0. If there are fractional digits, we continue until the remainder is 0, or until we see that the digits will repeat forever.

```
          100101.11
        +-------
    32  | 37.75
          32
         +-----
      16 |5.75
          0
         +----
       8 |5.75
          0
         +----
       4 |5.75
          4
         +----
       2 |1.75
          0
         +----
       1 |1.75
          1
         +----
      .5 |0.75
           .5
         +----
     .25 |0.25
           .25
          ----
           0
```

The rightmost digit in any Arabic number has the lowest power of 2 (the lowest weight), so it is called the *least significant digit*. Likewise, the leftmost digit is the *most significant digit*. In binary, a digit is called a bit, so we use the terms *least significant bit (LSB)* and *most significant bit (MSB)*.

Bits are often referred to by their position, which is the same as the exponent in an unsigned binary number. E.g., the rightmost bit is bit 0, the next one bit 1, etc.

Computer fixed point systems are generally integer systems with 8, 16, 32, or 64 bits.

The largest integer in any system is the one filled with the largest digit. E.g., the largest 5-digit decimal integer is 99999, and the largest 8-bit integer is 11111111.

An interesting property of maximum integers is that they are always equal to base$^{\#digits}$ - 1. If we add 1 to them, the sum digits are all 0 and the carry propagates to the far left.

```
            Decimal             Binary

Carry      11111               11111111

            99999               11111111
        +       1           +          1
        ---------           ------------
           100000 = 10^5       100000000 = 2^8
```

Hence, the largest N-digit decimal value is always $10^N$ - 1, and the largest N-digit binary value is always $2^N$ - 1. Table 2.1 shows the range of the common unsigned integer types.

Book addendum: PCs now mostly use 64-bit processors, extended versions of the 32-bit processors in use when the book was published.

### 2.5.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

| Bits | Range | Decimal Range |
|---|---|---|
| 8 | 0 to 2^8 - 1 | 0 to 255 |
| 16 | 0 to 2^16 - 1 | 0 to 65,535 |
| 32 | 0 to 2^32 - 1 | 0 to 4,294,967,295 |
| 64 | 0 to 2^64 - 1 | 0 to 18,446,744,073,709,551,615 |

Table 2.1: Range of Common Unsigned Binary Integer Systems

1. What is the decimal value of 1101.1_2?
2. What is the binary value of 49.125_10?
3. What are LSB and MSB?
4. What is the LSB in 100010?
5. What is the MSB in 010111?
6. What is the range of a 7-digit unsigned decimal number system?
7. What is the range of a 10-bit unsigned binary number system? Show your answer in binary, in powers of 2, and in decimal.
8. How many kinds of people are there in the world?

## 2.6 Binary Arithmetic

Binary arithmetic is easier than decimal (or other bases, which we will discuss later), because we are only working with 0s and 1s. Just remember that the largest possible digit is 1, so $2_{10}$ is $10_2$ and $3_{10}$ is $11_2$.

---

**Note** There are 10 kinds of people in the world: those who understand binary, and those who don't.

---

The process of adding, subtracting, multiply, or dividing binary is exactly the same as in decimal. We need only remember to express the results in binary, e.g. 10 instead of 2.

```
Carry  1   1

         1001
    +    1101
    ---------
       1 0110
```

The carry out of the MSB indicates an *overflow* if we are using a 4-bit unsigned binary integer system. The mathematical sum requires 5 bits, but our integer system only has 4.

In most CPUs, the carry bit is saved in a special register (storage cell) in the CPU so that we can check for overflow if necessary.

A 32-bit computer can add 64-bit numbers, but it will require two steps. First, add the lower 32 bits. Then add the upper 32 bits + the carry from the lower 32. This is called *multiple precision arithmetic*. Obviously, it takes twice as long as single precision arithmetic, where the computer can perform the operation in one step.

```
# Multiple precision illustration adding 8-bit values 4 bits at a time
# The lower (rightmost) 4 bits are added first.  This is one ADD instruction
# for the computer.  The upper 4 bits are then added using a different
# instruction, ADDC (add with carry), which adds two values, plus the carry
# from the previous instruction.

        ADDC  ADD
```

```
          1        Carry from previous ADD instruction
      0010 1001
  +   1001 1011
  -------------
      1100 0100
```

Hence, using 64-bit integers on a 32-bit CPU, or 128-bit integers on a 64-bit CPU will slow down the program.

### 2.6.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. (a) What is 1100_2 + 0111_2 in 4-bit binary?
   (b) What is the decimal value of the 4-bit sum?
   (c) Is there an overflow? Explain in terms of the resulting bits and in terms of the resulting value.
2. Explain multiple precision addition.
3. What is the down side of multiple precision arithmetic?
4. Represent +8 and -3 in 8-bit twos complement.
5. What are the decimal values of the 8-bit twos complement numbers 11111111 and 00001111?
6. How many different unsigned integers can we represent with N bits?
7. How many different signed integers can we represent with N bits?
8. How many different floating point values can we represent with N bits?

## 2.7 Signed Integers

Computer hardware represents all information as patterns of two voltages, which we call '0' and '1' for convenience. It does not have another state to represent '-'. We must somehow distinguish positive and negative numbers using just 0s and 1s.

There are many systems for representing negative numbers, which are covered in courses on computer architecture or assembly language. Here we only introduce the most commonly used system, called *two's complement*.

A positive value in two's complement always has a 0 in the leftmost bit and looks exactly the same as an unsigned integer with the same value.

```
# Unsigned
00001001_2      = 2^3 + 2^0 = 9

# Two's complement
00001001_twos   = +(2^3 + 2^0) = +9
```

A negative number has a 1 in the leftmost bit. However, the leftmost bit is *not* an independent "sign bit" as is often suggested. The fact that negative numbers have a 1 there is a consequence of the negation process that affects *all* bits.

To negate *any* two's complement value, whether positive or negative, we invert all the bits and then add 1. The inverted form of a binary value X is referred to as X', so in two's complement, -(X) = X' + 1.

```
-(00001001_twos) = 00001001' + 1 = 11110110 + 1 = 11110111_twos
```

Intuitively, we would think that reversing the process involves subtracting 1 and then inverting all the bits. This will work, but it is not necessary to have this separate process. The same process X' + 1 works in both directions.

```
-(11110111_twos) = (11110111 - 1)' = 11110110' =  00001001_twos
-(11110111_twos) = 11110111' + 1 = 00001000 + 1 = 00001001_twos
```

This is important in hardware design, since it means we only need one circuit, not two, to perform negation.

What are the decimal values of 01001111_twos and 11001000_twos?

```
01001111 is positive, so
      +(2^6 + 2^3 + 2^2 + 2^1 + 2^0)
    = +(64 + 8 + 4 + 2 + 1)
    = +143

11001000 is negative, so
      11001000
    = -(00110111 + 1)
    = -00111000
    = -(2^5 + 2^4 + 2^3)
    = -(32 + 16 + 8)
    = -56
```

Represent +7 and -12 in 8-bit two's complement.

```
+7 = 4 + 2 + 1 = 2^2 + 2^1 + 2^0 = 00000111_twos
-12 = -(8 + 4) = -(2^3 + 2^2) = -(00001100) = 11110011 + 1 = 11110100_twos
```

With N bits, there are exactly $2^N$ different values that can be represented, regardless of how the bits are interpreted.

With two's complement, half the patterns are used to represent negative values, and half are non-negative (positive or zero).

```
Binary      0000                0111 1000              1111
Unsigned    0                   7    8                 15
Two's comp  0                   7    -8                -1
```

Hence, the positive range is half of what it would be for an unsigned integer system with the same number of bits.

Generally, the non-negative values in two's complement range from 000..0 to 011..1 (0 to $2^{N-1}$ - 1) and the negative values range from 100..0 ($-2^{N-1}$) to 111..1 (-1). Table 2.2 shows the ranges of common computer signed integer sizes.

| Bits | Range | Decimal Range |
|---|---|---|
| 8 | -2^7 to +2^7 - 1 | -128 to +127 |
| 16 | -2^15 to +2^15 - 1 | -32768 to +32767 |
| 32 | -2^31 to +2^31 - 1 | -2,147,483,648 to +2,147,483,647 |
| 64 | -2^63 to +2^63 - 1 | -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 |

Table 2.2: Range of Common Unsigned Binary Integer Systems

---

**Note** It is important to know these ranges when selecting an integer data type in any language. We must choose a type that has sufficient range for our data, including the highest intermediate value computed while evaluating an expression, not just the end result.

---

One of the beauties of two's complement is that addition works exactly as it does in unsigned binary. We can ignore the sign of both numbers and just add them as if they were unsigned integers. Overflow detection is different, however. We detect overflow in two's complement addition by noting that the sign of the result is wrong. I.e., when adding two positives we get a negative, or vice versa. It is not possible to get an overflow when adding a positive and a negative.

```
Carry   1000                  1 1100

        0111     7, +7           1011    11, -5
    +   0100     4, +4       +   1110    14, -2
    --------                 --------
        1011    11, -5         1 1001     9, -7

    No OV for unsigned       OV for unsigned
    OV for two's comp        No OV for two's comp
```

### 2.7.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the range of a 12-bit twos complement system? Show your answer in binary, as powers of 2, and as decimal values.
2. Which of the common integer sizes are sufficient to represent Avogadro's constant?
3. What's the difference between unsigned addition and twos complement addition?
4. Add the following 8-bit twos comp values 01001000 and 01010011. Show results in twos comp and in decimal. Indicate whether an overflow occurs. Explain in terms of the bits computed and the value of the result.

## 2.8 Floating Point

Floating point allows us to represent fractional values as well as much larger integers than an integer system with the same number of bits.

Floating point stores values in a format similar to scientific notation, `mantissa * radix`$^{\texttt{exponent}}$. Modern systems use the IEEE floating point standards. The 32-bit standard uses a sign bit, an 8-bit exponent, a 24-bit mantissa of the form 1.F, and a radix of 2. Only the fractional part of the mantissa is stored, since the whole digit is always 1. The 64-bit standard uses an 11-bit exponent and a 53-bit mantissa. Since the leftmost bit of the mantissa is always 1, it need not be stored, and only the fractional bits are part of the format:

```
32-bit   S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
64-bit   S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Value = +/- 1.F * $2^E$.

A floating point system offers less precision than an integer system with the same number of bits, because some of the bits are used for the exponent. Only the mantissa contains significant digits. The 32-bit format has 6 to 7 decimal digits of precision while the 64-bit format has 15 to 16. A 32-bit integer handles up to 10 decimal digits and a 64-bit integer up to 20. ( See Table 2.1. )

The exponent essentially spreads out the values we can represent. We can represent larger integers, but the gap between numbers we can represent grows as the exponent grows. E.g., if incrementing the mantissa from 1.00000000000000000000000 to 1.00000000000000000000001 increases the *mantissa* by $2^{-23}$. However, if the exponent in the floating point value is 30, this change increases the value of the floating point number by $2^{-23} * 2^{30} = 2\text{^}7$, or 128. There are 127 integer values between that cannot be represented.

### 2.8.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Does floating point allow us to represent more different values than integer systems with the same number of bits? Explain.
2. What are two advantages of floating point over integers?
3. What are two disadvantages of floating point vs integers?

## 2.9 Floating Point Range and Precision

There are 5 properties of floating point values we must understand in order to use them properly:

- Largest positive value
- Smallest positive value
- Largest negative value (what does "largest" mean for a negative number?)
- Smallest negative value (what does "smallest" mean for a negative number?)
- Precision (maximum significant digits that can be stored)

---

**Note** Accuracy and precision are two different things. Accuracy indicates how close a value is to reality. It is a property of a particular value. Precision indicates how many digits we can represent. It is a property of the number system or the device. A scale that consistently reports weight to 3 decimal places, but is not calibrated (so it reads 0.000 when nothing is on it) is precise, but not accurate.

---

Book addendum: The book doesn't always distinguish between accuracy and precision.

When a result exceeds the largest positive or smallest negative value we can represent, we have an overflow. I.e. the magnitude is too large.

When a result is between the largest negative and smallest positive values we can represent, we have an *underflow*. I.e. the magnitude is too small.

When the true result requires more significant digits than our system can store, we have *round off* or *truncation* error. Truncation always results in a smaller value than the true result. Round off results in the nearest value that can be represented.

| Bits | Largest magnitude | Smallest magnitude | Precision |
|---|---|---|---|
| 32 | ~ +/- $10^{38}$ | ~ +/- $10^{-38}$ | 23 bits (~7 decimal digits) |
| 64 | ~ +/- $10^{308}$ | ~ +/- $10^{-308}$ | 52 bits (~16 decimal digits) |

Table 2.3: IEEE Floating Point Range and Precision

### 2.9.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What are the properties of floating point that programmers need to understand?
2. Which has a greater magnitude, the smallest negative value or the largest negative value?
3. What is precision and of what is it a property?
4. What is accuracy and of what is it a property?

5. Define overflow.
6. Define underflow.
7. Define round-off error.
8. Define truncation error.
9. Which of the common floating point sizes can represent Avogadro's constant?

## 2.10 Other Number Systems (Bases)

Binary has the advantage of being the simplest number base for implementing in hardware.

The down side is, the lower the base, the more digits a value has.

$65{,}535_{10} = 1111111111111111_2$.

Converting between binary and decimal is laborious due to the large number of bits, and does not always work perfectly. For example, the decimal number 1/10 cannot be represented in binary fixed point or floating point. It is like trying to represent 1/3 in decimal. It requires an infinite number of digits.

The solution to these problems stems from an interesting property of number bases. If $B = A^N$, then a digit in a base B number is exactly N digits in the base A number. E.g., a base 8 (octal) number is exactly 3 bits, since $8 = 2^3$, and a base 16 (hexadecimal) digit is exactly 4 bits, since $16 = 2^4$.

Octal digits are 0 to 7. Hexadecimal (hex) digits are 0 to 15, with 10 through 15 represented by A through F.

When converting between binary (base 2) and any base $2^N$, we can convert one digit at a time, rather than multiply and add or divide and subtract as we would when converting to/from decimal. Table 2.4 and Table 2.5 are provided for convenience.

| Binary | Octal |
|---:|---:|
| 000 | 0 |
| 001 | 1 |
| 010 | 2 |
| 011 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |

Table 2.4: Binary/Octal Conversion

```
    F093651C.4_16 = 1111 0000 1001 0011 0110 0101 0001 1100 . 0100

    74672343.5_8 = 111 100 110 111 010 011 100 011 . 101
```

When converting from binary, be careful to group the bits starting at the binary point and work outward. Pad any groups of fewer than 3 bits appropriately. 10 = 010 != 100. .01 = .010 != .001.

```
    10001010101.1001_2

=   10 001 010 101.100 1
=   010 001 010 101.100 100
=   2125.44_8

=   100 0101 0101.1001
=   0100 0101 0101.1001
=   455.9_16
```

| 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 2.5: Binary/Hex Conversion

When converting between binary, octal, and hexadecimal, we completely disregard the binary format of the number. We don't care whether it is an unsigned integer, two's complement, floating point, or any other system. Octal and hexadecimal are used to represent raw binary, not specific information formats.

```
1001010011010011_2      = 94D3_16
1001010011010011_twos   = 94D3_16
```

### 2.10.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Convert F841.5_16 to binary.
2. Convert 100100101001111.001_2 to hexadecimal.
3. Convert 673.1_8 to binary.
4. Convert 1011001011010011.01_2 to octal.
5. Convert 789.52_8 to binary.

## 2.11 Character Representation

The original standard character set is ASCII, which uses 7-bit codes to represent 128 characters including English letters, digits, etc.

ISO, the International Standards Organization, extended the ASCII set to create multiple 8-bit ISO standards including characters from non-English languages and additional graphic characters. E.g. the ISO-Latin1 standard covers western European languages. 16-bit extensions also exist to include non-phonetic languages, such as Chinese, which have far more than 256 characters, the limit for 8 bits.

Character codes are not numbers, though we often write them in decimal for convenience. E.g. the code for 'A' is 01000001, which can be treated as a number and written as $65_{10}$. The code for 'a' is 01100001. Note that 'a' differs from 'A' by 1 bit (bit 5). We can convert between upper and lower case by toggling this bit.

### 2.11.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What are the limitations of ASCII?
2. What is ISO?
3. How are ISO character sets related to ASCII?
4. What is EBCDIC?
5. Are ASCII and ISO codes numbers?

# Chapter 3

# Hardware and Software

## 3.1 What Makes Computers Tick?

A computer is like a pasta machine that turns programs into actions.

A computer scientist is a machine that turns pizza into programs.

## 3.2 The Main Components

### 3.2.1 The CPU

CPU = Central Processing Unit.

The CPU is controlled by machine instructions, and converts those instructions to electronic signals that drive memory, disk, and other devices.

Addendum: CPU has become a vague term that could mean a *physical* chip, or a *logical* unit of operation, for which the preferred term is now *core*. As of 2023, one CPU chip usually has 4 or more cores.

Different CPU architectures use different machine instructions. The x86 family of architectures (mostly Intel and AMD processors, including AMD Epyc, AMD Ryzen, Intel Core series, Intel Xeon), use completely different machine instructions than a Sun Sparc, an ARM, or a RISC-V processor. Most PCs use x86 processors, while most cell phones and tablets use ARM. The latest Apple Macintosh computers also use ARM.

RISC-V (RISC Five) is the world's first mainstream open source architecture. Unlike other CPU architectures, there is no patent, no licensing fees, and no NDA (non-disclosure agreement) for producers of RISC-V processors. RISC-V is rapidly taking market share from less open architectures.

### 3.2.2 Electronic Memory: RAM and ROM

RAM = Random Access Memory. It is *volatile* storage, meaning it's contents disappear when power is lost. (From chemistry, meaning something that evaporates quickly.)

ROM = Read-Only Memory. ROM works exactly like RAM, except that it cannot be overwritten and is non-volatile. A portion of the memory space in all computers must be ROM, so that there is a something for the CPU to run when the power is first turned on. The program code in this ROM is called *firmware* rather than software.

Both are randomly accessible, i.e. we can access any location directly. As a counterexample, a tape drive is not randomly accessible. We must read all locations sequentially before the one we want in order to find it.

The difference between RAM and ROM is that RAM can be both written and read. RAM would be better named RWM. Modern ROMs such as flash memory can be rewritten, but not as easily as RAM and only a small number of times (typically 100,000, then it's burned out).

### 3.2.3 Input/Output Devices

I/O devices include anything used to communicate with people, such as keyboards, mice, displays, and printers.

### 3.2.4 Mass Storage Devices

Mass storage is non-volatile storage that is cheaper and slower than electronic memory. It includes magnetic disks, tapes, and flash memory devices such as SSDs (Solid State Drives) and USB thumb drives.

Mass storage is typically tens of thousands to millions of times slower than RAM and ROM.

Machine instructions and data that need to be accessed multiple times are usually loaded into RAM rather than read from mass storage repeatedly.

---

**Note** If data need only be read once, it is foolish to load large amounts of it into RAM (such as an array or list). Doing so only increases resource requirements of the program and slows it down. More about this in Chapter 15.

---

### 3.2.5 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is a CPU? What is a core? What is the relationship between them?
2. What are volatile and non-volatile storage?
3. What is ROM? Is it volatile or non-volatile? What is it used for?
4. What is RAM? What might be a better name for it and why?
5. What are I/O devices for?
6. What is mass storage for? What are three common types of mass storage?

## 3.3 Programs and Programming Languages

### 3.3.1 Machine Language

It is often said that machine language is the only language that a CPU "understands". This is misleading. It is more accurate to say that machine instructions cause a CPU to generate electronic signals.

A machine language instruction is a package of bits indicating what operation to perform and the operands on which to perform it. The *opcode* indicates the operation. The operands are either *registers* (memory locations within the CPU, of which there are very few) or memory locations in RAM. Table 3.1 represents a hypothetical instruction that adds the contents of registers 1 and 5, placing the result in register 1.

| Opcode | Destination | Source #1 | Source #2 |
|---|---|---|---|
| 1100 | 0001 | 0101 | 0001 |

Table 3.1: A Hypothetical Machine Instruction

The bits of a machine instruction act like the pins on the drum of a music box. These pins are a binary "program" that is fed through the machine, flipping metal strips of different lengths, causing them to emit notes, as shown in Figure 3.1.

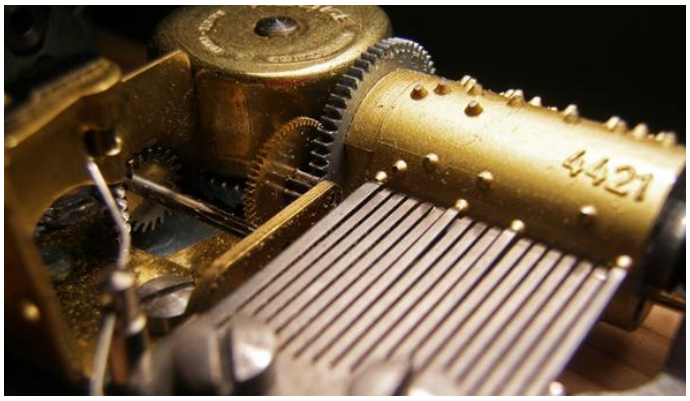



Figure 3.1: Music box (wonderhowto.com)

### 3.3.2 Assembly Language

Machine language is hard for humans to read, so we use *assembly language*, which is basically a mnemonic (symbolic) form of machine language plus a few added conveniences. The assembly language form of the machine instruction above might be written as:

```
            add     r1, r5, r1
```

Assembly language must be *assembled* to machine language by a program called an *assembler*. An assembler is an almost trivial program that reads one line of assembly *source code* as text and usually converts it to one corresponding binary machine instruction. Some assembly instructions may translate to a few machine instructions to provide some added convenience, but generally the assembly and machine instructions sets are about the same.

Assembly language is easier to read than machine language, but like machine language, it is specific to one CPU architecture, i.e. it is not portable. An assembly language program for x86 will have to be rewritten to run on ARM.

Assembly language programs are also very long, since machine/assembly instructions are very simple. Evaluating a polynomial may require a sequence of dozens of simple add and multiply instructions.

### 3.3.3 High Level Languages

High level languages are portable (not specific to one CPU architecture) and are much shorter and more intuitive than assembly language, thanks to constructs such as algebraic expressions, if statements, loops, and subprograms. Evaluating a polynomial can be done in one intuitive line:

```
            y = 3.4 * x * x - 5.1 * x + 0.2;
```

A *compiler* converts high-level language source code to a sequence of machine instructions, called *machine code* or *object code*.

A compiler is a highly complex program that must analyze groups of statements forming constructs such as conditionals, loops, and subprograms before it can output the equivalent sequence of machine instructions.

---

**Note** The Java "compiler" is not a true compiler. It converts Java source code to Java byte code, not machine code. The Java byte code does not control any CPU architecture directly. It must be interpreted by a program called an *interpreter*, which is part of the Java Runtime Environment (JRE).

---

The true compiled languages most commonly used today include C, C++, and Fortran. These three account for the vast majority of compiled open source programs.

### 3.3.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is machine language?
2. What is assembly language?
3. What are two major drawbacks to programming in assembly language?
4. What does an assembler do? How complex is it and why?
5. What is a high-level language?
6. What does a compiler do? How complex is it and why?
7. What does an interpreter do? How does it differ from a compiler.
8. Which will run faster, and by how much, a compiled program or an equivalent interpreted program?

## 3.4 The Programming Process

### 3.4.1 Algorithms

*Algorithms* are hypothetical recipes for solving problems. A program is not an algorithm. A program is an *implementation* of an algorithm, much like a car is an implementation of a CAD model.

Selection sort is an algorithm, where we find the smallest (or largest) element in a list, swap it with the first, and repeat for the remaining elements. Quicksort is a similar algorithm that is much more efficient in most cases. These algorithms can be implemented in any language and using different features of a language, such as arrays, lists, and recursion.

### 3.4.2 Top-down Designs and Stepwise Refinement

*Stepwise refinement* is the process of starting with a high level view of an algorithm, with no detail, and breaking it down into *a few* components. The process is repeated for each component until it is trivial to implement each component.

A *top-down design* represents *all* levels of the stepwise refinement process. It is a tree structure with the high-level statement as the trunk. Figure 3.2 shows a top-down design for a selection sort. Each node in this tree can be easily implemented as a subprogram in any language. The "Selection sort" node is the main program.

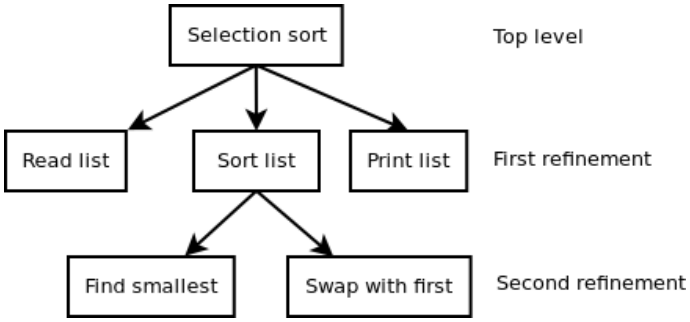



Figure 3.2: Top-down Design for Selection Sort

Diagrams like Figure 3.2 become very wide very quickly if we have more than a few levels of refinement. On paper, different levels of refinement can be represented with increasing levels of indentation. This format fits a typical document much better than a tree diagram.

```
Highest level
    First refinement
        Second refinement

Sort a file
    Read file into list
    Sort list
        Find smallest element
        Swap smallest with first
        Repeat for remaining elements
    Output sorted list
```

Top-down implementation is the process of writing the main program first, and gradually adding subprograms at increasing levels, while testing the partially complete program frequently along the way. This will be covered in more detail in Chapter 11.

### 3.4.3 Flow Charts

A flow chart typically represents *one* level of a top-down design graphically, to aid understanding.

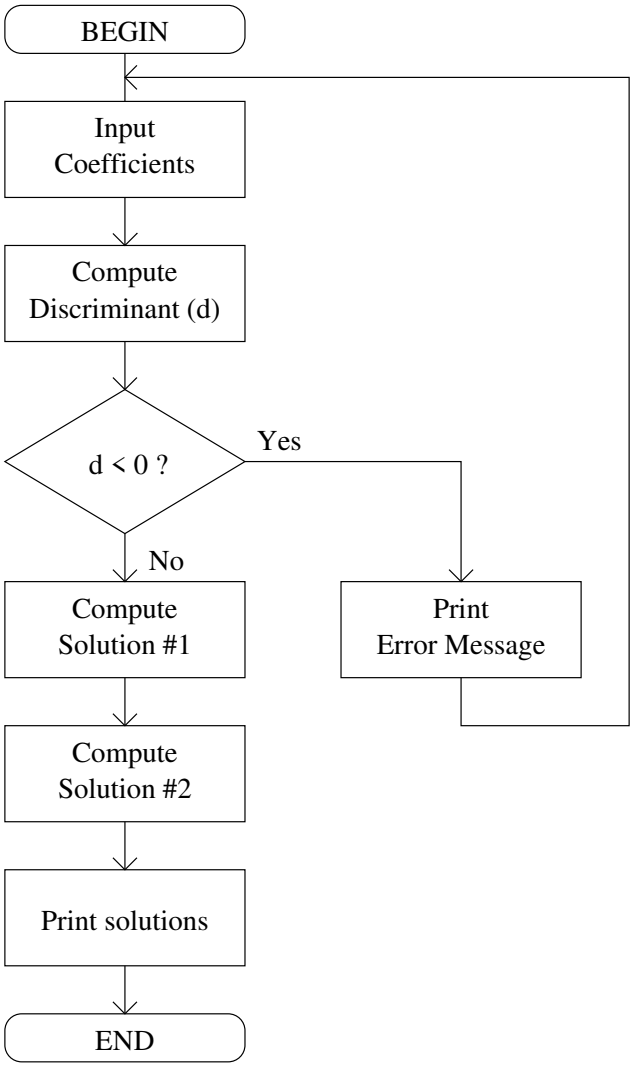



Figure 3.3: Flow Chart

### 3.4.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the relationship between algorithms and programs? Describe an example of each.
2. Describe stepwise refinement.
3. What is a top-down design?
4. What is a flowchart?

## 3.5 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.

### 3.5.1 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.

### 3.5.2 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.

### 3.5.3 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.

---

### 3.5.4 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.

### 3.5.5 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.

### 3.5.6 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.

### 3.5.7 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 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 4

# Unix Overview: Enough to Make You Dangerous

---

**Note** We cover this in roughly three 50-minute lectures, though for some courses it may be reasonable to teach Unix in more depth.

---

A "little" knowledge is a dangerous thing. Unfortunately, it is all we have time for in this course. Do not be content with the Unix we teach you here. If you master everything covered here, you will still be a Unix beginner. You will be much better off after you learn Unix properly. A free, comprehensive introduction is available in the Research Computing User's Guide: https://acadix.biz/publications.php

The purpose of this Unix overview is to learn enough to create directories for programming projects, **cd** around your directories, edit C source files and makefiles, and run a few other basic commands as part of the coding and testing process. However, there is more material covered here than is necessary for basic C coding. Some non-essential topics such as redirection and pipes are covered to give the reader better idea of how powerful Unix really is, so that they will understand the potential benefits of learning more about it.

---

**Note** Instructors: Take care not to get bogged down in this chapter. It's easy to get caught up in discussing Unix since there are so many great features to talk about, but remember that this course is about C programming, and the students only need to know enough Unix to write code. A separate course on Unix would be a great idea for many of the students.

---

## 4.1 What is an Operating System?

An operating system (OS) is not a "program". It is a collection of programs and subprograms that enable users to process files, communicate, write new programs, etc. An operating system contains several major parts:

- The *kernel* is the core of the operating system that controls the hardware. It is essentially a *library* (covered in Chapter 20) of functions called by programs that need to access hardware other than memory and the CPU, to work with files, network connections, the display, etc.

  Calls to kernel subprograms are called *system calls*.

  *Device drivers* are the parts of the kernel that actually manipulates the hardware. Writing device drivers requires understanding of the specific hardware device and how to communicate with it. Device drivers can be and often are written entirely in C. Assembly language is sometimes used as well.

- The *bootstrap program* is responsible for starting the operating system. The computer's BIOS (Basic Input/Output System) or other firmware, stored in ROM, checks the first block on the boot disk (the boot block) for a special structure. If found, it uses the information in the boot block to locate and load the first stage of the boot program provided by the operating system into RAM and begin running it. This begins the process of loading other OS components into memory, such as the kernel.

- A *user interface* allows people to communicate with the operating system. It may be a *command-line interface (CLI)* or a *menu-driven interface*, such as a *graphical user interface (GUI)*. MS Windows and Apple Macintosh computers have one standard GUI. Most Unix systems allow the user to choose from many different GUIs.
- *Utility programs*, often called *userland tools*, allow users to perform common tasks, such as manipulate files, communicate with other computers or people, etc. Typical Unix systems come with hundreds of utility programs preinstalled and make it easy to install thousands more.

In a *protected mode* OS, the kernel has complete control of the hardware. Processes (running programs) cannot access memory allocated to other processes. Users cannot interfere with the operation of each others' programs.

A *real mode* operating system allows direct access to all memory. A faulty or malicious program can even overwrite the kernel. DOS (disk operating system, the predecessor of MS Windows) is a real mode operating system. Real mode systems are typically only used in embedded applications today. All modern PC and server operating systems use protected mode.

### 4.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What are the major components of an operating system?
2. What is a protected mode operating system and where is it essential?

## 4.2 Unix Operating Systems (Was "The Unix Operating System")

Most people make most things far more complicated than they need to be. For many, it's essentially a deliberate, though not entirely conscious choice. People are emotionally driven by ego to show off how clever they are. This is especially true of engineers:

---

**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)

---

The wisest among us are self-aware enough to consciously override the impulse to look clever, for the sake of a more reliable and less costly product.

cleverness * wisdom = constant

Always try to follow the KISS principle: Keep it Simple, Stupid.

The original C and Unix developers were very wise. They designed an extremely simple and elegant new language in C, and then used it to build the simplest and most elegant operating system to date. Both C and Unix have been the model for most languages and operating systems that have followed.

It has often been said that people who use Unix have no need to write programs. This is true in many cases. Unix systems come with many utility programs that can be combined to accomplish most common tasks. Some of these will be covered in the coming sections.

**Caution**

Using a strong password and protecting it from exposure is especially important on a system where you can remotely log in and run programs. If someone cracks your email password, they can potentially log into the system and do nasty things in your name. On a remotely accessible Unix system, a hacker from another part of the world could log in as you, delete or alter your files, or use your account to launch other kinds of attacks.

A VPN is no protection against this if your VPN credentials are the same as your Unix login credentials, which is often the case. In fact, a VPN can *reduce* security by making it impossible to limit incoming connections to certain IP addresses.

Keep important passwords separated from less important ones, and take extra security precautions on the important ones.

Use a password vault such as KeePassXC (https://keepassxc.org/) to store your passwords and use an *extremely* secure password for KeePassXC itself.

Different terminal types use different *magic sequences* to control the screen and send different sequences when keys are pressed. For example, sending the character sequence "\033[H\033[2J" (\033 is character 33 octal, 27 decimal) to an xterm terminal causes it to clear the screen.

If your TERM environment variable is set incorrectly, strange things will happen while running programs that need to move the cursor around, set bold face fonts, change colors, etc. This is rare, but if your terminal is behaving strangely, it is something to look into.

Addendum: The **qterm** command for querying the terminal type is not available on all Unix systems and generally not needed anymore. Your terminal type will be correctly detected in most cases.

The Unix command-line interface (CLI) is provided by a program called a *shell*. The term comes from the analogy of an operating system as a nut, with the kernel being the inner layer encapsulating the hardware, and the user interface being the outer layer, or shell. Like many programs, a shell prints a *prompt* when it is expecting user input. In this text, we use the prompt "shell-prompt: " in all examples.

We will primarily use the CLI for this course. While virtually all Unix systems have GUIs, they are not always feasible for use on remote servers. A CLI performs better over slow connections such as WiFi and home Internet. A CLI is also a more productive interface for many types of work once you develop some skill with it. It provides instant access to virtually unlimited functionality, whereas a GUI is limited by what fits on the screen.

### 4.2.1 Addendum: Unix Commands

A Unix command consists of a command name followed by zero or more *arguments*, separated by spaces or tab characters:

- The command name is either the filename of a program or an internal command that is part of the shell.
- Flag (switch) arguments are special words or characters that tell the command how to behave. The almost always begin with a "-".
- Data arguments are either actual data values or the names of files/directories containing the data. They should never begin with a "-".

```
# Display the name of this Unix machine
shell-prompt: hostname
remote.server.edu

# Display the name of this Unix machine without domain fields
shell-prompt: hostname -s
remote
```

```
# Display only the domain name of this Unix machine
shell-prompt: hostname -d
server.edu

# Print the octal value of decimal 64
shell-prompt: printf "%o\n" 64
100
```

### 4.2.2 Addendum: Processes

A *process* in Unix is the execution of a program. I.e., the running of a program, not an unfortunate ritual beginning with a blindfold and a cigarette. When you log into a Unix system remotely, the first program run is a shell. I.e., the login session starts a shell process. If Joe and Sarah are both running the shell program **tcsh**, there are two processes, but only one program.

### 4.2.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is the purpose of the TERM environment variable?
2. What is a shell?
3. What is a GUI?
4. What is an advantage of a CLI over a GUI?
5. What are the components of a Unix command and how are they recognized?
6. What separates command components?
7. What is a process?

## 4.3 The Unix File-system

To a Unix operating system, a file is simply a sequence of bytes. End of story. Any structure within a file is of no concern to the operating system and entirely up to individual programs to interpret.

### 4.3.1 Partitions

Disks and *partitions* (portions of a disk) may each contain a *file system*, a tree structure containing files and directories (folders).

On MS Windows, each file system is assigned a drive letter such as C: or D:. Each has a root directory, called "\", e.g. "C:\", under which all other files and directories are contained.

On a Unix, there is only one root directory, called "/". Note that Unix uses a forward slash to separate file and directory names, unlike Windows with uses a backslash. Each disk partition under Unix is *mounted* (associated with a directory in the unified tree). One file system must be mounted as "/" and others are mounted on subdirectories. The **mount** command shows all of the file systems and where they are mounted in the directory tree. In the partial output of **mount** below, we see that the partition `/dev/mfid0p2`, which contains a UFS file system, is mounted as the root directory `/`. There are ZFS file systems mounted on `/unixdev1`, `/usr/home`, and `/usr/src`. Each of these would be a different drive letter on Windows.

```
shell-prompt: mount
/dev/mfid0p2 on / (ufs, local, soft-updates, journaled soft-updates)
devfs on /dev (devfs)
fdescfs on /dev/fd (fdescfs)
procfs on /proc (procfs, local)
zroot/unixdev1 on /unixdev1 (zfs, local, nfsv4acls)
zroot/usr/home on /usr/home (zfs, local, nfsv4acls)
zroot/usr/src on /usr/src (zfs, local, nfsv4acls)
```

Addendum: The book discusses IDE/EIDE, and SCSI. These two categories of disk interfaces still exist today, but have evolved. *ATA* is another name for IDE. Modern computers use *SATA (Serial ATA)* and *SAS (Serial Attached SCSI)* interfaces to replace the old parallel (multiple data wire) ATA and SCSI interfaces. Hardware engineers have found that we can actually achieve higher transfer rates over a single wire than we can over multiple wires, because of electromagnetic interference caused by nearby wires.

### 4.3.2 Directories

A *directory* is a file system object that documents the names and locations of other file system objects like files and other directories. In the context of other operating systems, it is often called a *folder*, as an abstraction to signify that it "contains" files and other folders.

A directory is actually a special type of file. The name "directory" is an analogy to the directory you might find in a building lobby listing the names and room numbers of each occupant.

The *root directory* is the one object in a file system that is not "contained" within another directory. In Unix, it is called "/". I.e. it has no parent directory.

A *full pathname* or *absolute pathname* indicates the complete path from the root directory to any given file system object. Multiple objects may have the same name, but their absolute pathnames are unique. Absolute pathnames always begin with a '/' or a '~' (explained shortly).

```
/usr/home/joe/Programs
/usr/home/sarah/Programs
```

A *home directory* is a directory belonging to an individual user, under which most or all of their files and other directories are found. On many Unix systems, all of the home directories are under `/home` or `/usr/home`. On macOS, they are under `/Users`.

```
/home/joe
/home/sarah
```

A user's own home directory can be referred to as "~" in many, but not all, situations. A different user's home directory can be referred to as "~user". The "~" symbol is not a standard Unix feature, so it will not work in all programming languages. It is recognized by most shells, however.

The *current working directory (CWD)* is an absolute pathname that a process prepends to pathnames that are not absolute. It is a property of every Unix process. It is often referred to as the directory the process is "in" at the moment.

If a pathname does not begin with a '/' or a '~', Unix prepends the CWD to compute the absolute pathname of the object:

absolute pathname = CWD + "/" + relative pathname

This applies in all programming languages, since construction of the absolute pathname is performed by a kernel routine.

We can change the CWD of our shell process using the **cd** command and display it using **pwd**. Typing **cd** with no directory name argument changes the CWD to your home directory. (It's like clicking your heels together three times. See Wizard of Oz if you didn't catch that.)

```
shell-prompt: pwd
/usr/home/bacon
shell-prompt: cd /etc
shell-prompt: pwd
```

```
/etc
shell-prompt: cd
shell-prompt: pwd
/usr/home/bacon
```

A *relative pathname* is the path not from the root directory, but from the CWD. If the CWD of Joe's tcsh process is his home directory, `/usr/home/joe`, then the relative pathname of `/usr/home/joe/Programs` is `Programs`.

Any pathname that is not absolute (does not begin with a '/' or '~') is relative to the CWD.

There are two standard special symbols for directory names:

- "." is the CWD. `./Programs` is the same as `Programs`. This is useful for referring to the CWD in some commands, such as

  ```
  shell-prompt: cp /etc/motd .    # Copy /etc/motd file to the CWD
  ```

  If you somehow end up with a filename that begins with a '-', you'll have a hard time using it as an argument to any Unix command, because the command will think the filename is a flag or a set of flags. You can simply prefix it with "./" so it no longer begins with a '-'.

  ```
  shell-prompt: rm -file
  rm: illegal option -- l
  usage: rm [-f | -i] [-dIPRrvWx] file ...
         unlink [--] file
  shell-prompt: rm ./-file
  ```

- ".." is the parent of the CWD. If the CWD of Joe's tcsh is `/usr/home/joe`, then the relative pathname of Sarah's Programs directory is `../sarah/Programs`.

  ```
  shell-prompt: pwd
  /usr/home/joe

  shell-prompt: cd ..
  shell-prompt: pwd
  /usr/home

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

### 4.3.3 Permissions

To control access to files and directories, each file system object has nine bits to indicate various permissions. There are three categories of users:

- User: The individual owner of the object
- Group: The group owner of the object
- Other: All other users on the system

Each of the three categories can have read, write, and/or execute permissions on any given file system object.

Execute permission on a file means that it is a program that people can execute (run). Execute permissions on a directory mean that a process can search the directory. Without execute permissions on a directory, processes cannot do much with it.

To see the permissions, we run **ls -l**. The **ls** command lists a directory and `-l` is a *flag* that tells **ls** to produce a "long" listing, with more information than the default. If we provide a file or directory name, **ls** will list that file or directory, otherwise it will list the CWD.

```
shell-prompt: ls -l /
drwxr-xr-x     2 root  wheel   1024 Nov 30 12:12 bin/
-rw-------     1 root  wheel   4096 Nov  4 16:23 entropy

shell-prompt: cd /

shell-prompt: ls -l
drwxr-xr-x     2 root  wheel   1024 Nov 30 12:12 bin/
-rw-------     1 root  wheel   4096 Nov  4 16:23 entropy
```

The partial listing above shows the permissions on the `/bin` directory and the file `/entropy`.

The first character is the file system object type. "-" means a regular file and "d" means a directory. There are other types that we will not cover here.

The next three characters show the user's read, write, and execute permissions in that order. A "-" means permission is denied, while an "r", "w", or "x" means read, write, or execute permission is granted.

The next three characters are group permissions, and the last three are "other" permissions.

To change permissions on a file, use the **chmod** (change mode) command. There are other mode bits associated with each file system object besides permissions, and chmod can be used to control all of them. We are concerned only with permissions here.

Addendum: We can use symbolic or absolute octal permissions specifiers as arguments to **chmod**. The symbolic form consists of a user category with one or more of 'u' (user), 'g' (group), and 'o' (other), followed by a '+' or '-' to grant or revoke permissions, followed by one or more of 'r', 'w', or 'x'. Bits not indicated by the specifier are unaffected.

Multiple specifiers can be used, separated by commas, but no whitespace!

```
# Prevent members outside the group from accessing Private-files
# and allow group read access at the same time
shell-prompt: chmod g+rx,o-rwx Private-files

# Allow members of the group and others to read Public-files
shell-prompt: chmod go+rx Public-files
```

Octal permissions specifiers set all bits absolutely. Each octal digit is three bits, where 1 = grant and 0 = revoke.

```
# Grant user read/write access, group read access, revoke "other" access
# 750 = 111 101 000
shell-prompt: chmod 750 Public-files
```

### 4.3.4 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is a file on a Unix system?
2. What is a filesystem?
3. What is a directory?
4. What is an absolute/full pathname of a filesystem object? Give an example.
5. How do we recognize an absolute pathname?
6. What is a home directory?
7. What is a CWD and what is it a property of?

8. If the CWD is /home/joe, what is the absolute pathname of "Programs/prog1.c"?
9. How can you change the CWD of your shell process to /usr/local/bin?
10. How can you change the CWD of your shell process to your home directory?
11. How can you rename a file called "-prog2.c" to "prog2.c", given that a '-' indicates a flag argument? (Use the **mv** command.)
12. How can you change the CWD of your shell process to the parent of the CWD?
13. How can you see the permissions on all the files in /etc?
14. How can you change the permissions on the directory Programs so that members of the group can read it, but nobody else can?

## 4.4 The Shell Environment

A *shell* is a program that implements a *command line interface (CLI)*. It reads in commands (using a simple string input), *parses* the command into command name, flags (switches), and data arguments, and then executes the command.

A shell may run under a terminal emulator on a local display, or may run on a remote computer, started by an SSH session.

Since command names and arguments are separated by whitespace (spaces and tabs), arguments that contain whitespace are problematic. We must either enclose them in quotes (single or double) or *escape* the whitespace characters by preceding them with a backslash, '\'.

```
# Suppose "Program Files" is the name of a directory
ls Program Files    # ls gets two arguments, "Program" and "Files"
ls: Files: No such file or directory
ls: Program: No such file or directory

ls "Program Files"  # One argument
prog.c

ls 'Program Files'  # One argument
prog.c

ls Program\ Files   # One argument
prog.c
```

*Internal commands* are part of the shell. The include commands such as **cd**, which changes the CWD of the shell process.

*External commands* are programs separate from the shell. An external command is simply the filename of any executable file, such as a compiled C program or a Python script.

### 4.4.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Why are arguments containing whitespace problematic and how do we get around the problem?
2. What are internal commands?
3. What are external commands?

## 4.5 Getting Help

In the early days of computing, documentation was provided in the form of paper (often ring-bound) manuals. Computer users would actually have to get up out of their chairs, walk to where the manuals were kept, and walk back to their desk. The designers of Unix sought an end to this injustice and invented the idea of online documentation, which could be viewed on the terminal screen. This way, they could come to work in the morning, sit in their chairs, turn their heads toward the screen and move nothing but their fingertips until the workday ended 10 to 16 hours later. Company staff would stop by once a day to refill the intravenous coffee drip and remove spider webs.

---

**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)

---

All standard Unix commands, C library functions, and some system files, are documented in simple markup files called *man pages*. Man pages are not always a good tutorial for a given subject, but are usually the best reference for looking up details quickly. For example, if you want to find out what flag arguments are available for the **ls** command, run:

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

The **man** command uses the **more** command to view the man page. The **more** command is a *paginator*, which displays a text file, allowing the user to move forward or backward one line or one page at a time, search for text, etc. A few of the most useful keys when using **more**:

- 'h': Display help screen, showing available key commands.
- Space bar: Move forward one page.
- 'b': Move back one page.
- Enter, or down-arrow: Move forward one line.
- Up-arrow: Move back one line.
- '/': Search forward for a string, entered after typing '/'.

Man pages are good for relatively simple programs and functions. For highly complex programs, the flat file program is difficult to navigate. If you want to experience the pain for yourself, try running **man tcsh** or **man bash**. These man pages are several thousand lines long. Such complex programs are better documented with a system that supports easy navigation, such as HTML.

The "SEE ALSO" section of each man page is an enormously valuable resource for learning about the existence of Unix commands, library functions, etc. Scroll down to the "SEE ALSO" section for a quick look every time you read a man page.

### 4.5.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What suffering did our forefathers have to endure in order to read documentation? How was this injustice finally eradicated?
2. How can you find out what arguments are required by the **strcmp()** function?

3. What are man pages generally good for and not so good for?
4. How can you find out what command or functions are available related to computing the sine of an angle?
5. How can you find out what keystrokes control the viewer while reading a man page?
6. How can you use man pages to discover related commands and functions?

## 4.6 Some Useful Commands

A basic set of useful Unix commands is included in the lab manual and will be covered in lab, where you should try them out during class.

Also see the book for a brief list of common commands.

Here are a few examples:

```
cp /etc/hosts .             # Copy /etc/hosts to the CWD
mv pro1.c prog1.c           # Rename pro1.c to prog1.c
mv ../prog1.c .             # Move prog1.c from the parent of CWD to CWD
rm prog1.o                  # Remove prog1.o
cat file.txt                # Echo contents of file.txt to the terminal
head -n 10 file.txt         # Display first 10 lines of file.txt
tail -n 10 file.txt         # Display last 10 lines of file.txt
grep aardvark file.txt      # Show lines in file.txt containing "aardvark"
vi file.txt                 # Edit file.txt with Unix-standard visual editor
mkdir Program1              # Create directory Program1 in CWD
mkdir ./Program1            # Same as above
clear                       # Clear the terminal screen
date                        # Show today's date
```

### 4.6.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What Unix command could you use for each of the following tasks? Just name the command, no explanation needed.
   - Copy files
   - Move or rename files
   - View a text file on screen at a time
   - View the first N lines of a file
   - View the last N lines of a file
   - Search a file for strings or patterns
   - Sort a text file line-by-line
   - Create a directory
   - Reformat a C program
   - Combine files into an archive
   - Display a calendar for this month

## 4.7 A Few Shortcuts with T-shell and Bash

In the olden days, shells were very primitive and offered no features to edit commands other than backspacing to the location of an error and retyping everything after it. Modern shells offer very sophisticated features to help minimizing typing.

### 4.7.1 Command History

The shell *history* is a memory of recently entered commands, usually at least the last 1,000. We can see the history by typing **history**.

```
shell-prompt: history
   991  9:39    ape unix.dbk
   992  9:40    cd Books/Computer-books/C-Unix-lab/
   993  9:40    ls
   994  9:40    ape unix-overview.dbk
   995  9:47    svn add useful-unix-commands.dbk
   996  9:48    make
   997  9:48    ps
   998  9:53    ape useful-unix-commands.dbk
   999  11:37   history
```

You can execute previous commands using '!' followed by a history number or any number of characters at the beginning of a previous command. This will rerun the last command that began with those characters. Using the history above, the **make** command can be rerun using **!996**, **!m**, **!ma**, etc.

You can use the arrow keys to move around the command history as if editing a file. The up arrow will move back to the previous command, and the down arrow to the next one. Left and right arrows will move around the currently displayed command.

Instructor should demonstrate or reader should try this.

### 4.7.2 File Specification: Globbing

Sometimes we want to provide many filename arguments to a command. Unix shells have a feature that allows us to indicate a "glob" of files with a simple pattern, rather than typing all the filenames. A '*' represents any sequence of characters, including none. For example, if a directory contains thousands of files, and we want to list just those with names ending in ".c", we can use the following:

```
shell-prompt: ls *.c
```

Or, to remove all files with names ending in ".o":

```
shell-prompt: rm *.o
```

The shell expands "*.c" to a list of filenames that end in ".c" *before* running the **ls** command. If there are three C programs in the CWD called "main.c", "search.c", and "sort.c", then the shell will actually execute the following command after expanding the glob:

```
shell-prompt: ls main.c search.c sort.c
```

A list or range of characters enclosed in square brackets will match any one of those characters. For example, if a the CWD contains files prog1.c through prog9.c, the following would match prog3.c, prog4.c, and prog5.c.

```
shell-prompt: ls prog[345].c
shell-prompt: ls prog[3-5].c
```

### 4.7.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How can you see a list of recently executed commands?
2. How can you execute the most recent command beginning with "cc"?
3. How can you scroll back through the last few commands?
4. How can you remove all the files in the CWD with names ending in ".core"?

## 4.8 Unix Input and Output

### 4.8.1 Standard Streams

All Unix processes have at least three file streams open from the moment they are born: The *standard input*, *standard output*, and *standard error*. The standard input is attached to the terminal keyboard by default. The standard output and standard error are both attached to the terminal screen by default. Hence, program output to both may be mixed up on your screen. Figure 4.1 depicts the relationship between devices, streams, and a process.

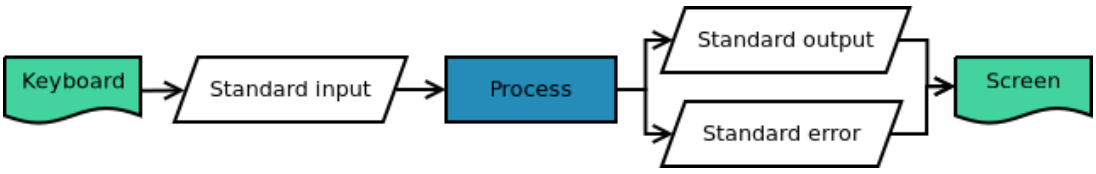



Figure 4.1: Standard Streams

When we run a command such as **ls**, it sends normal output to the standard output and error, warning, and informational messages to the standard error.

---

**Note**

Unix programs NEVER receive input from or send output to a specific hardware device directly. They always send and receive data through streams (or the lower level file descriptors underlying streams).
The statement "The ls command lists files on the terminal screen." is incorrect. Unix does not work that way.

---

### 4.8.2 Redirection

Any Unix program can have its streams or descriptors disconnected from one file or device and connected to a different one. This is called *redirection*. Unix shells make this very simple. To redirect the standard output to a file, we simply place the filename after '>' in the command:

```
shell-prompt: ls > ls-output.txt
```

This will create a file called `ls-output.txt` which will contain the output of **ls**. The output will not appear on the screen, since the standard output was disconnected from the screen and connected to the file `ls-output.txt`, as shown in Figure 4.2.

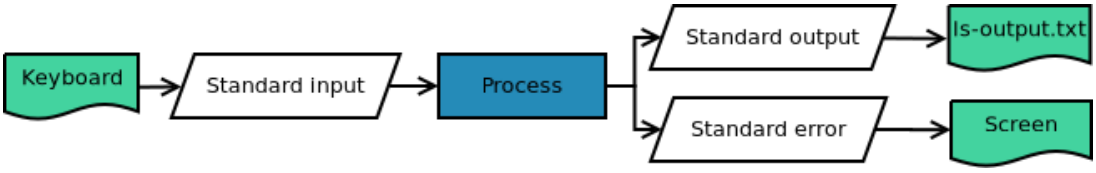



Figure 4.2: Redirection

Likewise, we can redirect the standard input by placing a filename before '<'. Think of < and > as arrows pointing in the direction of data flow. In most modern shells, we can redirect the standard output and standard error together using '>&'. In some shells, we can redirect the standard error separately using '2>'. ( It uses '2' because the integer file descriptors for standard input, standard output, and standard error are 0, 1, and 2. )

### 4.8.3 Pipes

We can also redirect the output of one process straight to the standard input of another. This saves time and space by running both processes at the same time and eliminating the need for a temporary file:

```
# Run ls, saving the output, then use more to view the output
shell-prompt: ls > ls-output.txt
shell-prompt: more ls-output.txt

# Run ls and pipe the output directly through more
shell-prompt: ls | more
```

### 4.8.4 Device Independence

Redirection and pipes are made possible by the concept of *device independence*. Device independence means that every input and output device on a Unix system is read or written in exactly the same way as an ordinary file. In fact, most devices have a filename under the `/dev` directory. For instance, the mouse or touchpad may be represented as `/dev/sysmouse` and the keyboard as `/dev/kbd0`.

All Unix commands can read or write devices just like ordinary files. For example, one could print a program a text printer using a command such as **cat prog1.c > /dev/printer**.

C programs use the same read and write functions to access a file, the keyboard, the mouse, or a network connection.

### 4.8.5 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What are the standard streams used by all Unix processes and to what device are they normally connected?
2. Is it correct to say that a Unix program takes input from the keyboard? Why or why not?
3. What is redirection?
4. Show a Unix command that writes the last 10 lines of the file input1.txt to the file input1-last-10.txt.
5. Show a Unix command that runs the program prog1, which is in the CWD, and reads input from the standard input, using input1.txt as input, and shows the output of the program one screen at a time.
6. What is device independence?

## 4.9 Job Control

Unix commands can be run in the *foreground* or the *background*. The only difference is that the foreground process receives input from the keyboard. All background processes must have their input redirected from another source. It would be an amazing coincidence if multiple processes could all use exactly the same input, so it makes no sense to send keyboard input to more than one process. Multiple processes *can* all send output to the same terminal screen, though this is not likely a good idea, since it would be very confusing to anyone looking at the screen. Background processes should generally have their output redirected as well.

The commands shown so far have all been foreground processes. We can run a process in the background simply by placing an '&' at the very end of the command:

```
shell-prompt: long-running-program < input.txt >& output.txt &
```

Now the process is running in the background, and the shell immediately takes another command rather than waiting, as it normally would for a foreground process.

We can terminate the foreground process by typing Ctrl+c.

We can suspend the foreground process by typing Ctrl+z. It can then be resumed in the foreground by typing **fg**, or in the background by typing **bg**.

### 4.9.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is the difference between a foreground process and a background process?
2. How can you terminate the foreground process?
3. How can you pause the foreground process, list the contents of the CWD, and then resume the process in the foreground?
4. How can you run a ./prog1 so that is immediately runs in the background and you can continue using the shell for other commands?

## 4.10 Shell Variables and Environment Variables

A Unix shell is actually a language interpreter than supports conditionals, loops, etc. and uses variables, like any other programming language. When you enter Unix commands at the shell prompt, the shell is actually running a program as you type it.

All shell variables are strings. There are no integers, floating point values, or other types.

Some shell variable names are reserved and have special meaning, such as `prompt` in tcsh and `PS1` in bash, which indicate what the shell prompt should look like. You can change your shell prompt by changing the `prompt` variable. We can see all of the current shell variables by typing **set**.

```
shell-prompt: set
```

Environment variables are like shell variables, but they are a property of *every Unix process*, not just a shell process.

The most important property of environment variables as that they are passed on to child processes. When you run an external command from the shell, all environment variables in the shell process are inherited by the new process created to run the command.

This is a simple form of *interprocess communication (IPC)*, which is covered in detail in Chapter 28.

For example, we can set the `LSCOLORS` environment variable in the shell process. It is inherited by the process running the **ls** command and used to colorize the output. The `TERM` environment variable is used by programs that need to control the terminal screen and understand special keys such as F1 and arrow keys, which send a magic sequence rather than a single character.

One of the most important environment variables is `PATH`, which contains a colon-separated list of directories that are searched for external commands. Most commands are installed into one of the "bin" directories:

```
shell-prompt: printenv PATH
/usr/local/bin:/usr/bin:/bin
```

If a program is not in one of the directories in your `PATH`, you will get a "command not found" error when you try to run it.

### 4.10.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is a shell variable?
2. Can we use any name we want for a shell variable?
3. What is an environment variable?
4. Are environment variables always created by a shell process?
5. How does the shell find the programs that constitute external commands?

## 4.11 Shell Scripts

Thanks to device independence, the input of a shell process can be redirected from a file rather than the keyboard, just like any other Unix program. This means that *anything* you type at the shell prompt can also be placed in a file called a *shell script*, which we can then run repeatedly.

**Note** Any complex command or sequence of commands that you *might* want to run again should be placed in a script rather than typed in repeatedly.

Since there are multiple shells in Unix and the commands in a script may not be compatible with the shell we use interactively, we need to specify which shell should run the commands in our scripts, using a *shebang line*. The shebang line is a special comment that begins in the very first character of the file. The most portable shell to use for scripting is the POSIX Bourne shell, `/bin/sh`. Every Unix systems have a Bourne shell, while other shells are mostly add-one packages. A complete Bourne shell script would appear as follows:

```
#!/bin/sh -e

ls -als
hostname
```

The `-e` flag tells the Bourne shell process to terminate if any of the commands it executes fail (return a non-zero status).

Bourne shell is standardized by POSIX, so POSIX Bourne shell scripts should run without modification on any system. Other shells, such as bash and tcsh have many additional features, but they are primarily improvements to the CLI for interactive use. There are some additional features for scripting that POSIX Bourne shell does not have, but they don't make scripting much easier.

**Caution**
Only Bourne shell and C-shell (csh) should use an absolute pathname in the shebang line. All other shells, such as bash or tcsh, may be installed in different directories on different Unix systems, e.g. `/bin`, `/usr/local/bin`, `/opt/local/bin`, etc. To accommodate this, we use a different shebang form:

```
#!/usr/bin/env bash

set -e      # Enable terminate-on-error

ls -als
hostname
```

Using `/bin/bash` is not portable.

### 4.11.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is a shell script?
2. What Unix feature makes shell scripts possible?
3. Which of the many Unix shells is best for writing portable scripts?
4. Is the best shell for scripts also the best for interactive use?
5. What should the shebang line look like for a Bourne shell script?
6. What should the shebang line look like for a bash script?

## 4.12 Advanced: Make

Make is a tool that compares time stamps on a source file and a target file generated from the source, and runs a specified command if the source is newer.

**Note**
**Make** is commonly used to compile programs, but it is *not* just for programming. It can be and often is used to build many kinds of generated files. Keep an open mind when thinking about **make** as you may find it useful for many things besides programming.
This PDF is built from multiple DocBook XML source files using **make**. The book is built from multiple LaTeX source files using **make**.

**Make** is commonly used for building an executable file from multiple source files. When a programmer edits a source file and saves the changes, the time stamp is updated, so that file becomes newer than the executable file. Running make causes that file to be recompiled to rebuild the executable.

### 4.12.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What does make do?
2. What can make be used for?
3. What is make commonly used for?

# Part II

# Programming in C

# Chapter 5

# Getting Started with C and Unix

**Note** We cover this in roughly one 50-minute lecture, though for some courses it may be reasonable to teach Unix in more depth.

The world desperately needs good C programmers. A great deal of time and resources are wasted by interpreted language scripts doing heavy computation that a proper C program could do in 1/100 the time. In scientific computing, the same solution is often reinvented hundreds or thousands of times as a shell or python script, because no one has taken the time to develop a stable, installable software project to perform the task. The reason, for the most part, is simply that they don't know how. Shell scripting, Python, or R are often the only language they know, so they end up using them inappropriately, writing duplicate disposable scripts that perform poorly, do no error checking, and are often incorrect. Countless of man-hours are wasted on this duplicated effort, despite the fact that installing a tool for the task could be as easy as "pkg install tool-name", if someone took the time to write one.

My work as a bioinformatician aims to do just that, and to show others how it can be done, so we can all get our work done more easily and spend more time with our families and friends. https://github.com/auerlab/.

## 5.1 What is C?

C is a high-level, portable programming language with low-level capabilities and performance. It offers all of the most important features of a high level language, such as flow control constructs, subprograms, structures, and type definitions, while at the same time making it possible to do things that are otherwise only possible in assembly language. It also offers run time performance close to or equal to that of assembly language. C is used on all kinds of devices from the smallest microcontrollers to supercomputers.

An example at the low end is OpenVex, a 100% C firmware for Vex Robotics PIC-based microcontrollers, which have only 32 KiB of flash program memory and about 2 KiB of data RAM. OpenVex was developed using the open source SDCC (Small Device C Compiler). (https://github.com/outpaddling/OpenVex)

An example at the high end is samtools, one of the most heavily used tools in the bioinformatics field. Samtools is often used to process many terabytes of data on HPC (High Performance Computing) clusters. (https://www.htslib.org/)

Most C code in use today conforms to one of the ANSI or ISO standards. New standards are release about once every decade, but very few changes have been made to the language since the 1990s.

C is perhaps the simplest high-level language. The designers, led by Dennis Ritchie, made a conscious decision to leave out any features that could be implemented as a C function. This keeps the compiler simple and fast, and minimizes the learning curve. The C language itself can be mastered in a few months by a capable college student. From there, extending knowledge is a matter of learning about available function libraries.

C++, in contrast, is one of the most complex languages ever created. It would take at least three or four semesters to master all the features of C++.

### 5.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Is C a high-level language or a low-level language?
2. Compare C and C++.
3. How long should it take to master C? C++?

## 5.2 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 5.1** 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 (;).

### 5.2.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What are the major components of a C program?
2. What is a free-format language?

## 5.3 A Word about Performance

C is the fastest high-level language in existence. The only way one is likely to write faster code is by coding in assembly language. C programs are typically tens to hundreds of times as fast as the same design implemented entirely in an interpreted language, as we saw in Section 1.2.

This does *not* mean that coding in C automatically makes your programs as fast as they can be. Choosing the most efficient algorithms is usually the most important decision for achieving performance. An O(N*log N) algorithm implemented in an interpreted language will run faster than an O(N^2) algorithm implemented in C for some value of N and all greater values. In general, the factors that affect performance, from most to least important are:

1. Algorithm (e.g. selection sort vs quicksort, linear vs binary search)
2. Programming language (compiled languages are orders of magnitude faster than interpreted, simpler languages tend to run faster than more complex and abstract languages).

---

3. Code optimizations (integers are faster than floating point, using less memory leads to faster average memory access, etc.)
4. Parallelism (using multiple cores may help, but is not always possible)
5. Hardware speed (upgrading your PC is generally the least cost-effective way to make programs faster). However, it might be the only way if you don't have access to the source code.

Optimizing *all* of the code is foolish, however. Most parts of most programs do not contribute much to run time. *Profiling* is the process of determining *where* a program uses most of the CPU time. This can be done by inserting timers into the code (check the clock before and after a loop), or using compiler tools in some cases.

### 5.3.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How does the performance of C programs compare to the same algorithm/design implemented in other languages?
2. What are the major factors in program performance, from most to least important?
3. What is profiling and how does it help us with program performance?

## 5.4 Some Early Warnings

C and other high-performance languages achieve their speed in part by *not* performing run time checks. Some languages, for example, check every array reference to ensure that the subscript is within range, and check for integer overflow after every arithmetic operation. This has a high impact on program speed. C compilers do not perform such checks by default. This makes it possible to write programs that are much faster when such checks are not necessary.

Interpreted languages generally do perform such checks, and the performance penalty is minimal in this case. Since interpreted languages spend about 99% of their time parsing the code, even doubling the run time of the other 1% wouldn't make a noticeable difference.

These conditions only occur due to program bugs, and the designers of C thought it better to let bugs be discovered by other means and let programmers learn to be disciplined and write safe code, rather than become dependent on the compiler to find bugs for them.

It has often been said that C makes it easy to shoot yourself in the foot, while C++ offers more ways to shoot yourself in the foot.

### 5.4.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Why does C not check for run time errors by default?

## 5.5 Coding and Compiling a C Program

### 5.5.1 Coding

C code can be written using any text editor. However, it is highly advisable to use an *Integrated Development Environment (IDE)*. An IDE is an editor specifically designed to facilitate programming, by recognizing language features and interfacing with compilers and interpreters.

Without an IDE, one would have to either exit the editor to compile and test the program, or compile and test from another terminal emulator window. With an IDE, the programmer can run the compiler from within the editor. For example, in *Another Programmer's Editor (APE)*, one can build the executable simply by typing F7 (or Esc followed by x if the F7 key does not work). One can build and run by typing F5 (or Esc followed by r). See Figure 5.1.

IDEs also offer features such as syntax colorization, which makes the code easier to read and flags typos before you even attempt to compile or run the code. E.g., if a keyword such as "while" is mistyped, it will not be colorized, which you will likely notice immediately.

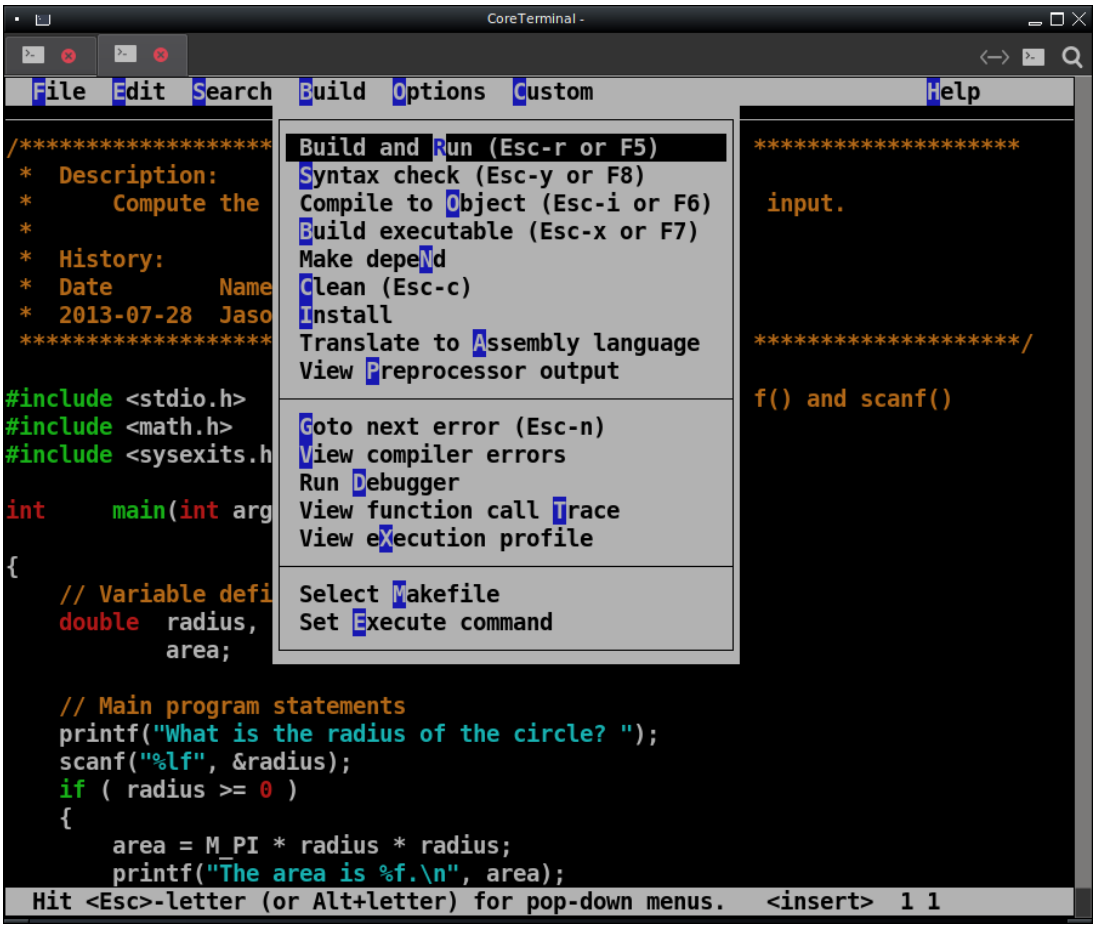



Figure 5.1: APE Build Menu

Most IDEs are graphical applications, which makes their use on remote computers over an SSH connection dubious. APE is a terminal-based application, which works just as well on a remote machine as on the local machines.

**Caution** Do not use a Windows editor to create text files for Unix. The Windows text file format is slightly different, having carriage returns at the end of each line. Some Unix programs will choke on these carriage returns.

### 5.5.2 Compiling

A *compiler* is a complex program that translates high level language source code to machine code. It must "understand" high-level constructs such as conditionals, loops, and subprograms. It should not be confused with an *assembler*, which is a very simple program that translates assembly language source code to machine code, one line at a time.

There are many C compilers, but most Unix systems use either clang/llvm (FreeBSD, macOS, etc) or gcc (Linux, OpenIndiana, etc). On all Unix system, the default compiler, whether it is clang, gcc, or something else, can be invoked as **cc**. There is no reason to explicitly invoke **clang** or **gcc**, unless you want to use a non-default compiler, which is rarely beneficial.

#### Compilation Stages

In all three languages, production of an executable file involves up to three steps, outlined below and in Figure 5.2.

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 Chapter 13.

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**.

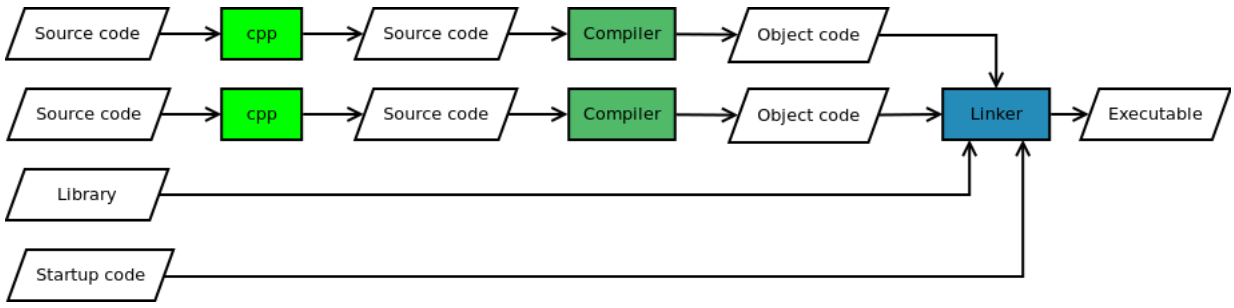



Figure 5.2: Compilation

Compilers include an *object code optimizer*, which combs through the machine code generated by the compiler's code generator, looking for ways to make it shorter and faster. The portable way to run the optimizer is using `-O`, `-O2`, `-O3`, etc. with the **cc** command. Higher numbers mean more aggressive and riskier optimizations. With clang, `-O` and `-O2` are actually the same.

Optimization levels beyond -O2 generally have very little impact on performance. Higher level optimizations may also make it difficult or impossible to use a debugger to locate problems in the source code. The executable contains a map connecting source code line numbers to locations in the generated machine code. Debuggers need this map to tell you where a problem occurred in the source code. Aggressive optimizations may rearrange the machine code to the point where it is no longer possible to maintain this map.

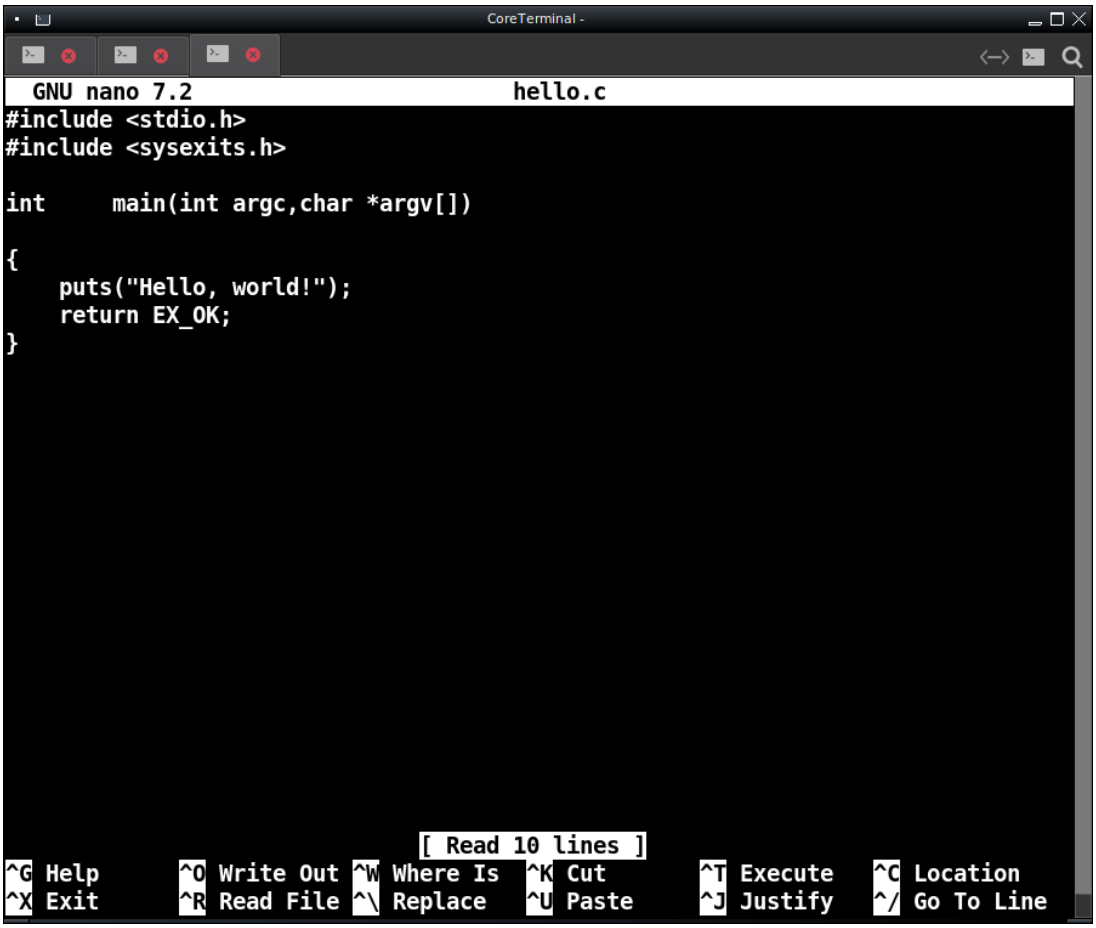
There are also more specific optimization flags, such as `-march=native`, which tells the compiler to utilize advance machine instructions on the local CPU. This will make the executable non-portable. I.e., if you compile on the latest AMD Ryzen processor with this flag, the compiler will generate instructions that don't exist on an Intel i5. Compiling with just `-O` will only generate instructions common to the entire family of CPUs to which the local CPU belongs.

### 5.5.3 A First Example

Addendum: The **pico** editor has been replaced by **nano**. Book says "** Type in the sample program above" when the example is actually on the next page.

Instructor will demonstrate the following:

```
shell-prompt: nano hello.c
```

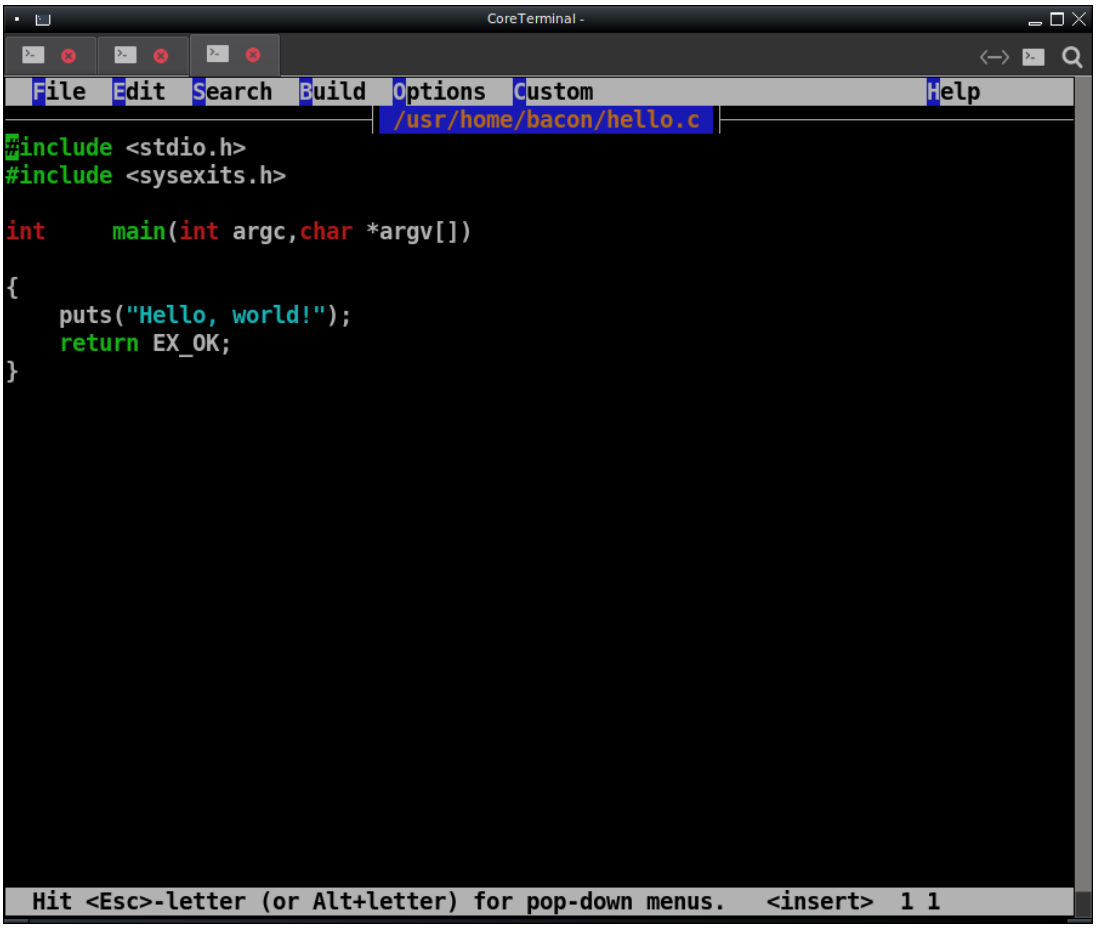


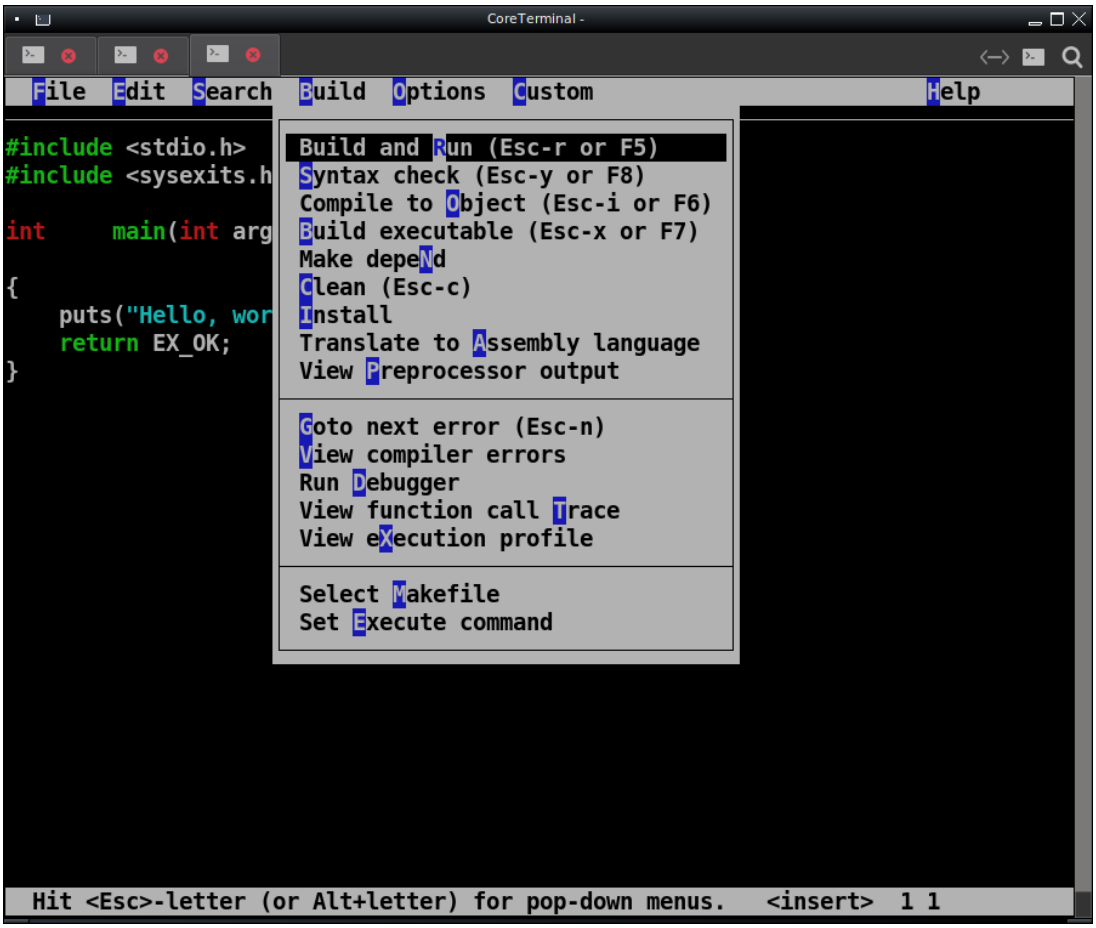

After saving the file and exiting the editor, we compile and run the program:

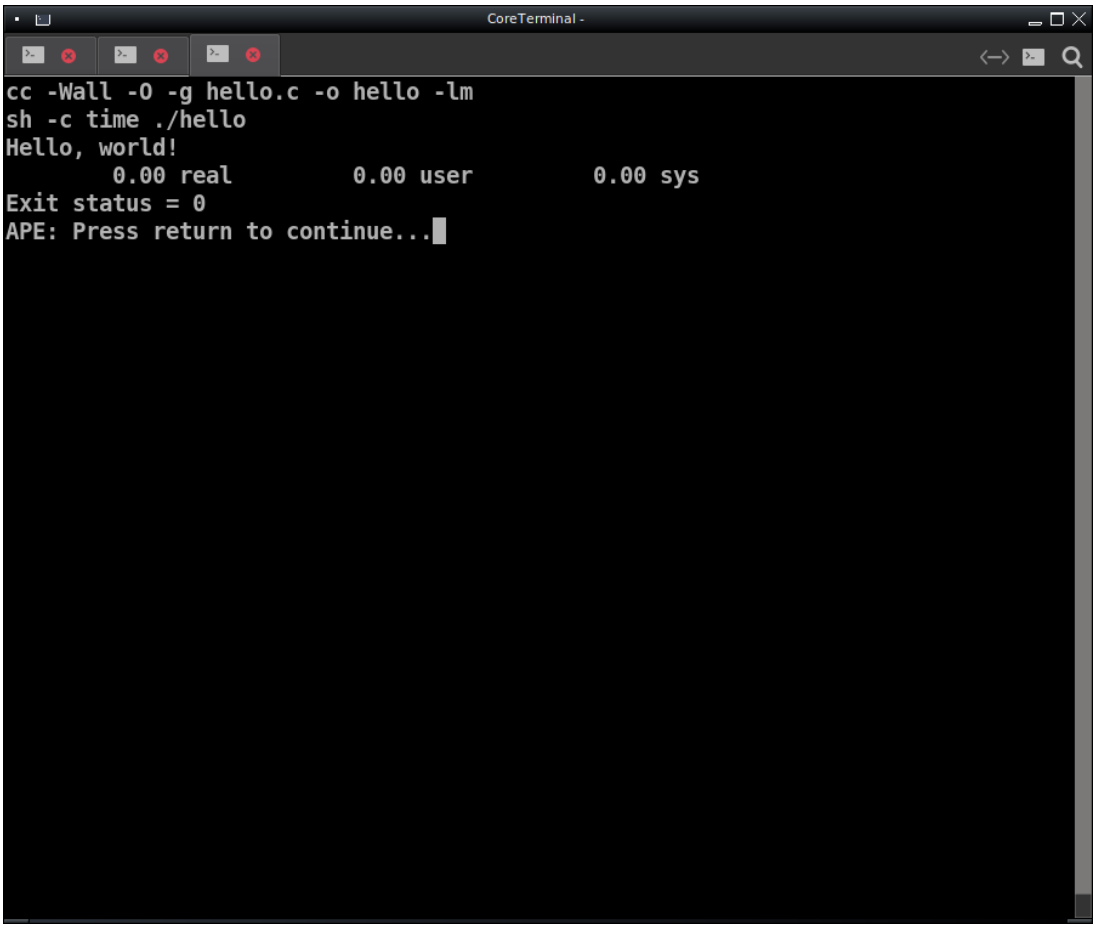
```
shell-prompt: cc -Wall hello.c -o hello
shell-prompt: ./hello
Hello, world!
```

Instructor may now to the same with APE, to demonstrate how much more effective an IDE is than a simple text editor like nano.

```
shell-prompt: ape hello.c
```


CoreTerminal -
File Edit Search Build Options Custom Help
/usr/home/bacon/hello.c
#include <stdio.h>
#include <sysexits.h>
int main(int argc,char *argv[])
{
puts("Hello, world!");
return EX_OK;
}
Hit <Esc>-letter (or Alt+letter) for pop-down menus. <insert> 1 1


CoreTerminal -
File Edit Search Build Options Custom Help
#include <stdio.h>
#include <sysexits.h
int main(int arg
{
puts("Hello, wor
return EX_OK;
}
Build and Run (Esc-r or F5)
Syntax check (Esc-y or F8)
Compile to Object (Esc-i or F6)
Build executable (Esc-x or F7)
Make depend
Clean (Esc-c)
Install
Translate to Assembly language
View Preprocessor output
Goto next error (Esc-n)
View compiler errors
Run Debugger
View function call Trace
View eXecution profile
Select Makefile
Set Execute command
Hit <Esc>-letter (or Alt+letter) for pop-down menus. <insert> 1 1




### 5.5.4 Handling Errors and Warnings

Error and warning messages are your friends. It is much better to get an error or warning message telling you exactly where a potential problem is, than to spend time trying to figure it out based on incorrect output or program crashes.

Always use the `-Wall` flag when compiling to help minimize your debugging effort.

Both **clang** and **gcc** have a `-Wall` flag that requests all possible warning messages to be generated. Wise programs always use this flag during development and clean up the code to silence warnings. This is an easy way to eliminate most bugs from your code.

In addition to compiler warnings, you can clean up the code further by running **cpplint file.c** or **splint file.c**. These commands check for formatting and style issues, security holes, etc.

Demonstrate.

### 5.5.5 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is an IDE?
2. What are some advantages of IDEs over simple text editors?
3. What are the stages in building an executable from a C source file?
4. What command should usually be used to compile C programs on a Unix system?

5. What is the safe way to run the object code optimizer so that debuggers will work and the executable will run on all related CPUs?
6. How do we get as much help as possible finding bugs in our code from clang or gcc?
7. What other tools can we use to check for potential problems in our code?
8. Following the sine example in the lab exercises, write a C program that asks the user for a number and prints the square root of the number. The program should print the best possible input prompt. Use the `scanf()` function for input and the standard library `sqrt()` function to compute the square root. Check the man page for `sqrt()` to see what header files and compiler flags it requires.

   Make sure the program compiles without warnings when using `-Wall`. Also run **cpplint** and **splint** on the code to check for style issues.

   What is the best name for the source file?

# Chapter 6

# Data Types

## 6.1 Introduction

Different data call for different data types. Some data can only be positive integers. Others may be fractions or irrational numbers.

Our own choices affect the type of the data. Representing monetary amounts in dollars requires fractions, such as $5.99. Representing them as cents makes everything in integer, e.g. 599 cents.

Choosing the optimal data type for your data is extremely important. The wrong data type can lead to reduced performance or incorrect results.

Changing the data type later is costly. Modifying a finished programs makes it unfinished again, and requires thorough retesting. THERE IS NO CHANGE SO SMALL THAT RETESTING IS NOT NECESSARY.

### 6.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Why is it important to choose optimal data types?
2. What happens if we discover a need to change a data type after a program is "finished" (knowing that no program is ever really finished)?

## 6.2 Variables

A *variable* is a name, or *identifier*, that refers to a memory location. The compiler converts variable names to memory addresses that are used in the machine language executable. A piece of data stored in memory is called an *object*. Hence, a variable in a program refers to an object.

C variable names have the same limitations as most other languages. I.e., they must begin with a letter or an underscore, which can be followed by more letters, underscores, or digits. The maximum length of a variable name in modern compilers is more than we would ever want to type. The rules for naming variables can be expressed using the *regular expression* "[A-Za-z_][A-Za-z_0-9]*", which means one character in the set containing A-Z, a-z, and _, followed by zero or more characters in the set containing A-Z, a-z, _, and 0-9.

Languages like C require us to *define* all variables. Some other languages, such as Matlab and Unix shells, do not. Variables in those languages are implicitly defined and allocated memory when they are first assigned a value.

When we define a variable in C, we are assigning a name and a data type, as well as allocating memory to contain the object. The data type tells the compiler what binary format should be used when storing the object in memory. For example, a C `int` is

stored in 16-bit or 32-bit two's complement format, an `unsigned short` in 16-bit unsigned binary, and a `double` in 64-bit IEEE floating point format. The compiler and the library functions take care of managing the binary formats, but it is important for us to know the limitations of each format.

---

**Note** *Defining* and *declaring* are not the same thing. A declaration does not allocate space, but merely states the type and possibly other attributes of something defined elsewhere, usually in a different source file. Declarations are also known as *allusions*. This will be clarified in later sections. For now, just be aware of the distinction.

---

**Example 6.1** Example

Write a program that inputs the radius of a planet in kilometers and prints the volume.
What data type(s) should we use for the radius and the volume, and why? Do we need fractional components? Do we even know the radius of a planet that accurately? Are planets perfectly round, so the radius is consistent to that degree? What are the ranges of radius and volume? The radius of Jupiter is 71,492 km according to Wikipedia. There are probably bigger planets in other solar systems. Does a 16-bit integer have enough range? How about a 32-bit integer?
Use the Unix **bc** command to compute the volume of Jupiter: 4 / 3 * pi(16) * 71492^3
We see that there are many questions that we need to answer before choosing a data type. Many programmers are unaware of these issues, which results in many problems for them and for the users of their software. We will clarify these issues in subsequent sections.

```
#include <stdio.h>
#include <sysexits.h>
#include <math.h>

int     main()

{
    // What C data type has enough range for Jupiter's volume?
    // The radius is 71,492 km, producing a volume of 1530597322872155.9 km^3.
    // We'll use double here to make sure that we have enough range,
    // and discuss this more later after examining C's data types.

    double  radius, volume;

    fputs("Please enter the radius of the planet in kilometers: ", stdout);
    if ( scanf("%lf", &radius) == 1 )
    {
        volume = 4.0 / 3.0 * M_PI * radius * radius * radius;
        printf("The volume is %f km^3\n", volume);
        return EX_OK;
    }
    else
    {
        fputs("Error reading the radius.  Please try again.\n", stderr);
        return EX_DATAERR;
    }

    return EX_OK;
}
```

The program above reads a string of characters entered by the user using `scanf()`. The `scanf()` function converts the character sequence to IEEE floating point format and stores the bits at the memory location allocated for `radius`. The program then computes the volume as an IEEE floating point value and stores the result at the memory location allocated for `volume`. Finally, the `printf()` converts the IEEE floating point value at memory location `volume` to a sequence of characters that people can read and sends them to the terminal screen.

**Programming with Style**
Note that the variable names used above are unambiguous. We did not use abbreviations such as "r" or "v" or "rad" or "vol". Doing so would demonstrate laziness or impatience. Fully descriptive variable names make the program *self-documenting*. This reduces the need for comments, so in a sense, it is a form of *enlightened* laziness.
The variable `volume` in the program above is not really necessary. We could have simply placed the expression in the `printf()` call. However, using a variable this way also makes the program more self-documenting, as well as preventing statements from becoming too long and complicated. The combined statement below takes a bit more effort to read than the two separate ones above. It's a small difference, but hundreds of situations like this one add up to a lot of fatigue during a 12-hour day a coding.

```
printf("The volume is %f\n", 4.0 / 3.0 * M_PI * radius * radius * radius);
```

### 6.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. How does machine language access data in memory?
2. What is a variable in a high-level language?
3. What happens when a user types a real number as input to the `scanf()` function?
4. Should variable names of the same type be placed on the same line or on separate lines?
5. What is the advantage of using complete words as variable names instead of abbreviations?
6. How much training should a user need before using a program?

## 6.3 C's Built-in Data Types

Unlike some languages (e.g. Matlab, Fortran), C supports only *scalar* (dimensionless, single-value) built-in types. *Aggregate* types (arrays, structures) can be defined by the programmer using the scalar types.

All C integer types can be either signed (two's complement) or unsigned. Use the `unsigned` modifier before the type to specify unsigned. There is also a `signed` modifier, but it is rarely used since `signed` in the default.

```
signed int      a;  // Two's comp
int             b;  // Same as a
unsigned int    c;  // Unsigned binary
```

Primitive Java data types have fixed sizes. For example, a Java `int` is 32 bits and a `long` is 64 bits. This makes code easily predictable regardless of the CPU on which they run. However, it has a cost in terms of performance.

In contrast, C `int` and `long` sizes depend on the CPU architecture. This means that we do not know their exact size or range when we write the code. However, this allows programs to achieve maximum performance on all CPUs. Adding 32-bit integers on 16-bit CPU or 64-bit integers on a 32-bit CPU requires two machine instructions, which slows down the program. To avoid this performance issue, C defines the `int` type so that operations can be completed by one machine instruction on virtually any CPU. The `int` is usually 16 bits on a 16-bit CPU and 32 bits on 32-bit and 64-bit CPUs. The minimum size is 16 bits, so `int` will require multiple precision arithmetic on 8-bit CPUs. However, 8-bit CPUs are rare and nobody expects them to be fast anyway. The `long` type is normally 32 bits on 8-bit, 16-bit, and 32-bit CPUs, but is 64 bits on 64-bit CPUs.

**Caution** Since `int` and `long` have variable size, we need to make contradictory and pessimistic assumptions about them. E.g., since an `int` may be either 16 or 32 bits, depending where our code is compiled, we must assume that it has the range of a 16 bit integer (-32,768 to +32,767) but the memory requirements of a 32-bit integer (4 bytes each). If we need greater range, `int` will not work on some CPUs. If we need to conserve memory for a large array, we should use `short`, which is always 16 bits.

Addendum: The book's integer data types table does not include types for 64-bit CPUs, which were not yet available when the book was written. Also new since publication is support for complex numbers (introduced in the C99 standard). Table 6.1 shows the data types available in modern C.

| 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 | +/- ($1.1754 \times 10^{-38}$ to $3.4028 \times 10^{38}$) | 24 bits (6-7 decimal digits) |
| double | Almost always 64-bit floating point | +/- ($2.2250 \times 10^{-308}$ to $1.7976 \times 10^{308}$) | 52 bits (15-16 decimal digits) |
| long double | 64, 80, 96, or 128-bit floating point | +/- $3.3621 \times 10^{-4932}$ to $1.1897 \times 10^{+4932}$) | 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 6.1: C Data Types

The `void` type is used to define functions that do not return a value, or *pointers* (variables that contain memory addresses) that point to an unspecified type of object. These topics are covered in later chapters.

Addendum: The C language traditionally did not have a Boolean type. C99 introduced the `_Bool` type, which is not meant to be used directly (as indicated by the leading '_'). The header file `stdbool.h` defines the `bool` type along with `true` and `false` constants. In reality, Booleans are represented as integers internally.

### 6.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is a scalar variable?
2. What is an aggregate variable?
3. What is the size of an `int` in C? A long? An unsigned int?
4. What is the range of an `int` in C? A long? An unsigned long?
5. When should we use the `int` data type?

## 6.4 Constants

Every constant in the C program has a data type, just like a variable. Carelessness when writing constants can affect program performance or cause incorrect output.

- Any sequence of only digits not beginning with '0' is a decimal `int`. Examples: 255, 13
- Any sequence of only digits beginning with '0' is an octal `int`. Examples: 0, 0377 = 255, 015 = 13
- Any sequence of only digits beginning with '0x' or '0X' is a hexadecimal `int`. Examples: 0xff = 255, 0xD = 13
- An integer constant followed by 'l' or 'L' is a `long int`. Examples: 255l, 0377l, 15L
- Addendum: An integer constant followed by 'u' or 'U' is an `unsigned int`. Examples: 255u, 0377u, 15U
- Addendum: An integer constant followed by 'ul' or 'UL' is a `unsigned long int`. Examples: 255ul, 0377ul, 15UL
- A sequence of digits containing a period is a `double`. Examples: 3.4, 0.56
- A sequence of digits containing a period followed by 'e' and an integer is a `double` in scientific notation. Examples: 3.4e5 = 3.4 * 10^5, 5.6e-1 = 5.6 * 10^-1
- A constant like those above followed by 'f' or 'F' is a `float`. Examples: 3.4e5f, 4.6f
- A single character or *escape sequence* such as \n, \r, \010 (= 8), etc. between single quotes is an `int`, NOT `char`!!! In C, values smaller than an `int` are promoted (converted) to `int` before all math operations, so using `int` speeds up the code by avoiding conversions. A \ followed by 3 digits inside single quotes is an octal number, sign-extended to `int`. Examples: '\377' = -1, '\177' = +127.
- A sequence of characters between double quotes is a *string*, which is an array of `chars`. Strings can also contain escape sequences such as \n or \377.

  The compiler adds a null byte ('\0') to each string constant. C library functions that work with strings expect the end of the string to be marked with a null byte.

So why is it so important to write constants with the correct type?

```
double  y, x;

// What's wrong with this code?
x = 8;
y = 3 / 4 * x;
```

3 / 4 is 0, so y is 0 rather than the expected 6.

```
float   y, x;

// What's wrong with this code?
y = 3.0 / 4.0 * x;
```

3.0 and 4.0 are doubles, which means we are mixing float and double in the expression. Conversions must be done between float and double, which slows down the program. More on this in Section 8.3.

Addendum, p120: `long` and `int` are usually the same on 32-bit computers, but not on 64-bit computers.

**Caution**

We must know the binary format to fully understand the effects of using constants.

```
char ch = 255;

printf("%d\n", ch);     // Prints -1, not 255.  Why?
```

### 6.4.1 Named Constants

Constants should almost never be *hard-coded*, i.e. written in literal form, into program statements. Rather, they should be given a name, usually in a header file, and the name used in statements.

This makes the code self-documenting and also means that we need only change the value in *one place*, should it ever need updating. Most constants used in programs end up being changed at some point in the future. If the value of Pi ever changed, we would have bigger problems than code maintainability, but most constants in programs are not universal laws of geometry or physics.

The traditional way to define constants in C is using the preprocessor. By convention, constants are written in all upper case letters, so that they are easily distinguished from variables:

```
#define MAX_LIST_SIZE   1000
#define DEBUG           1
```

This can also be done using the -D flag in the compile command, so that we can toggle a constant value without editing the code:

```
shell-prompt: cc -O -Wall -DDEBUG=1 file.c
```

We can also define a constant as a read-only variable by prefixing the type with `const`. Such a variable can *only* be assigned a value in the initializer of the variable definition:

```
const size_t    MAX_LIST_SIZE = 1000;

MAX_LIST_SIZE = 2000;   // Compile error
```

**Caution**

When defining floating point constants, don't be lazy. Provide the value to the full precision of the type, so you don't cause unnecessary round-off error.

```
#define PI      3.14                    // Sloppy coding
#define PI      3.14159265358979323846
```

**Caution**
Don't reinvent constants. Universal math constants, system-related limits (such as the maximum length of a filename), etc. are defined in the standard C headers such as `math.h` and `limits.h`.

```
// A small section of /usr/include/math.h
#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) */
```

**Note**
To see exactly what the preprocessor does with `#include` and `#define`, run **cc -E file.c | more**.

### 6.4.2 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What are the data type and decimal value of each of the following?
   (a) 45
   (b) 0xAu
   (c) 012L
   (d) 3.1e2
   (e) 3.1e2f
   (f) 'A'
   (g) '\033'
2. Show a C constant of type `int` with an octal value of 377.
3. Show a C constant of type unsigned long with a binary value of 10010011.
4. Show a C constant of type double with a value of 6.02 * 10^23.
5. What are the decimal ASCII/ISO values of each byte in the string constant "123\r"?
6. How can we assign an unsigned integer variable its maximum value without knowing the exact value?
7. What is the problem with the following code? Fix it.

```
double  base, height, triangle area;

triangle_area = 1 / 2 * base * height;
```

8. What is the problem with the following code? Fix it.

```
    float base, height, triangle area;

    triangle_area = 1.0 / 2.0 * base * height;
```

9. What is "hard coding"?
10. What is a better alternative to hard coding?
11. How can we define a constant MAX_NAME_LEN to 50 in a C program?
12. How can we define a constant MAX_NAME_LEN to 50 in a C compile command?
13. How do we ensure that our floating point constants don't cause unnecessary round-off error?

## 6.5 Initialization in Variable Definitions

We can assign a value to a variable where it is defined. However, it is usually better to assign a value immediately before the code that needs is initialized. This is an example of code *cohesion* at the lowest level, where statements that belong together are all in one place. Cohesion is also applied at higher levels, such as member functions (methods) of a class belonging together.

```
int     sum = 0, num;

// 30 lines of code

// Now we have to waste time scrolling back to verify that
// sum is properly initialized
while ( scanf("%d", &num) == 1 )
    sum += num;
```

```
int     sum, num;

// 30 lines of code

// This is more cohesive.  We see the initialization of sum right
// where it is needed and don't have to go looking for it.
sum = 0;
while ( scanf("%d", &num) == 1 )
    sum += num;
```

### 6.5.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What are the pros and cons of initializers in variable definitions?

## 6.6 Choosing the Right Data Type

Choosing a data type for each variable in a program is *not trivial* and is *extremely important*. It could mean the difference between correct output and incorrect output, and could significantly affect program performance.

- Avoid floating point like the plague. Floating point arithmetic takes three times as long as integer arithmetic. Floating point is less precise than an integer of the same since, since some bits are used for the exponent. Comparison of floating point values is unreliable due to round-off error.

```
double      x;

// Some code that computes x and y

// This will fail if round-off error results in a value of
// 0.99999999999999999999 instead of 1.0
if ( x == 1.0 )
{
}
```

  Floating point is the best option in some situations, but always try to make a program work with integers if possible.

- An `int` may be either 16 or 32 bits. You do not know where your code will be used in the future. Use `int` if 16 bits provides enough precision and 32 bits is not too much memory to use. The latter is only a concern for large arrays.

  `Int` is the fastest data type on all CPU architectures with at least 16-bit word sizes, since it can be processed by a single instruction.

- A `long`, may be either 32 or 64 bits, so make the same kind of pessimistic assumptions as we do for `int`.

  Don't use `long` if `int` will suffice. A `long` may require multiple precision arithmetic on some computers, where it is larger than the native word size.

- Use `short` *only* to save memory or to deliberately limit the range of a variable. We should almost never define a scalar `short` variable. Adding two `short` variables involves converting (promoting) both values to `int` first, which slows down the program.

- The `char` type is generally only used for strings, but if used as a small integer, apply the same logic as for `short`. A scalar variable for holding one character should usually be defined as `int`, not `char`.

- If an integer type does not have enough range, consider using `unsigned` before going to a larger type. For example, a variable that holds positive integers up to 50,000, it cannot be `int`, which ranges from -32,768 to +32,767. Using `long` instead would be foolish, however. An `unsigned int` provides the necessary range (0 to 65,535) while avoiding multiple precision arithmetic.

- If floating point cannot be avoided, use `double` unless you need to conserve memory (e.g. for large arrays). Using `float` will not make the program measurably faster and will severely limit precision to about 7 decimal digits, causing more round-off error in the results.

Table 6.2 shows run times of a simple loop using various data types.

| **Data Type** | **Run Time** |
|---|---|
| short | 0.77 |
| int | 0.61 |
| long (same as int on this system) | 0.61 |
| long long (multiple precision arithmetic) | 1.52 |
| float | 3.52 |
| double | 3.52 |

Table 6.2: Execution Time of a Loop with Various Types

Addendum: The header file `stdint.h` defines derived (not built into the compiler), fixed-sized integer types such as `int32_t`, `uint32_t`, `int64_t`, etc. for cases where you want to guarantee the exact size and range of an integer variable (unlike `int` which varies from one CPU to another) and are willing to accept the cost of multiple precision arithmetic in some cases (such as uint64_t on a 32-bit CPU). The file `inttypes.h` includes `stdint.h` and also defines facilities for reading and writing these types using `printf()` and `scanf()`.

### 6.6.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Why is it important to choose optimal data types?
2. When should we use floating point types?
3. When should we use `int`?
4. What is the advantage of using `int` over explicitly using a 16-bit integer or a 32-bit integer?
5. When should we use `short`?
6. When should we use long?
7. When should we use `unsigned`?
8. How do we choose between `float` and `double`?
9. What is the optimal data type for a scalar variable containing a person's age.
10. What is the optimal data type for a large array of a people's ages.
11. What is the optimal data type for a large grid of ocean depths measured in feet.
12. What is the optimal data type for a scalar variable containing molar concentration of a salt solution, ranging from roughly 10^-10 to 10^20.
13. What is the optimal data type for a large array containing molar concentrations of a salt solution, ranging from roughly 10^-10 to 10^20, and measured to an accuracy of 5 significant figures.
14. What is the optimal data type for a scalar variable containing net income of a company with a maximum of $2,000,000 per year.
15. What is the optimal data type for a large array of incomes similar to the previous question.

## 6.7 Creating New Type Names: Typedef

One of the most important features of a high level language is extensibility. The ability to create new data types is a key form of extensibility. This is very simple in C.

```
typedef existing-type new-type-name;
```

It is traditional to end user-defined type names with "_t":

```
// Save some typing when defining unsigned int variables
typedef unsigned int    uint_t;

// Make the code self-documenting when using char to hold small
// integers rather than characters
typedef char            tiny_int_t;
```

```
// Allow for easy switching between float (to save memory) and
// double (for better precision).  Just change double to float here
// and recompile the program for a lower memory executable.
typedef double  float_t;

int     main()
```

```
{
    // Use float_t rather than hard-code double or float everywhere
    float_t     x, y;

    ...
    return EX_OK;
}
```

Type definitions should generally be placed in header files, which can then be included in multiple source files that will use the type.

---

**Caution** Addendum: It may be tempting to use `#define` to create type aliases, but this approach can cause many problems that are avoided by using `typedef`, which is just as easy.

---

### 6.7.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Show a type definition equating `age_t` to `unsigned char`.

## 6.8 Addendum: Enumerated Types

An enumerated type is an integer type with a limited number of named values. The first identifier is assigned a value of 0 by default.

```
typedef enum { RED, GREEN, BLUE } color_t;

// Roughly equivalent to
typedef int color_t;

#define RED     0
#define GREEN   1
#define BLUE    2
```

We can alter the values as follows:

```
typedef enum { RED = 1, GREEN, BLUE } color_t;

// Roughly equivalent to
typedef int color_t;

#define RED     1
#define GREEN   2
#define BLUE    3
```

# Chapter 7

# Simple Input and Output

## 7.1 The Standard I/O Streams

All Unix input and output is ultimately performed by the low-level `read()` and `write()` system calls (kernel functions), which read and write blocks of characters to/from a file or I/O device. These functions are covered in Chapter 25.

To create the illusion of inputting and outputting single characters, numbers, and other formatted data, the Unix C libraries provide a *stream I/O* interface. When writing to a stream, stream I/O functions place individual characters into an *output buffer* (queue), a character array in memory. When the buffer is full, the entire buffer is written to the file or I/O device using a single `write()`. Likewise, when reading from a stream, a single `read()` reads a block of characters into an *input buffer*, and stream functions take characters from the buffer until there are none left, and which time the next block is read.

All Unix processes are born with three streams already open, namely `stdin`, `stdout`, and `stderr`. These streams are normally connected to the terminal, but can be redirected to/from any file or I/O device. More streams can be opened explicitly, as discussed in Chapter 21.

To use all of the standard I/O functions in this chapter, we need to include the header file `stdio.h` in our programs.

---

**Note** To find out what headers are required for any C library function, run **man function-name**.

---

### 7.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How does all Unix I/O ultimately occur?
2. What is a stream?
3. What standard streams are available in all Unix processes and to what devices are they connected?
4. How do we find out what header files are required for a C library function?

## 7.2 Single Character I/O

To input a single character from `stdin`, we use `getchar()`, which is part of the standard Unix C library. To output a single character to `stdout`, we use `putchar()`. The `getchar()` function reads a single character from the `stdin` buffer. If there are no characters remaining, it reads the next line of block from the file or device in order to refill the buffer. The `putchar` writes a single character to the `stdout` buffer. When the buffer is full, the buffer contents are written to the file or device, and the buffer is marked empty.

```
#include <stdio.h>
#include <sysexits.h>

int main()

{
    // Use int for scalar character variables, not char!
    // char and short are promoted to int in math operations and
    // when passing to functions, so using int in the first place
    // avoids overhead
    int     ch;

    ch = getchar();
    putchar(ch);

    return EX_OK;
}
```

Normally, input read from a terminal keyboard is *line-buffered*, meaning the low-level `read()` reads characters into the input buffer until return is pressed. The `getchar()` above will not respond immediately when a key is pressed, but will return the first character in the buffer after the entire line is entered. This behavior can be changed using methods discussed in Chapter 26. Input read from a file is normally *block-buffered*, so if the program above is run with input redirected from a file, an entire block from disk (typically something like 4096 characters) will be read into the input buffer before `getchar()` returns the first character.

### 7.2.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How does putchar() work with the stdout stream buffer?
2. What data type does getchar() return and why?

## 7.3 String I/O

The original developers of the standard C library made a few questionable decisions that we have been stuck with ever since. 1970 was a different time, and the designers of early library functions did not foresee the popularity of C and the need for better robustness and security that is obvious today.

One of the issues is the `gets()` function, which reads a line of text from `stdin` into a character array.

```
#include <stdio.h>
#include <sysexits.h>

#define MAX_STR_LEN 100

int     main()

{
    // Define an array of char to hold the string
    // +1 for the null terminator character
    char    string[MAX_STR_LEN + 1];

    gets(string);
}
```

The problem is that this function does not know the size of the target array. Hence, it is possible that someone could type in a line of 60 characters that the program tries to put in a 40-character array. This will result in corrupting memory addresses that follow the array, usually other variables defined in the same function. This sort of problem can cause incorrect output, program crashes, and security holes in some cases.

We could use the more general `fgets()`, which reads from any stream and requires the size of the array as an argument. This will ensure that the array is not overflowed, though it will leave the remaining part of the input in the input buffer to be taken in by the next input function called, which is probably expecting something else.

```
fgets(string, MAX_STR_LEN + 1, stdin);
```

Addendum: The `gets_s()` function, available in C11 (the 2011 C standard), is equivalent to `fgets()` used with `stdin`, except that the newline is not stored in the string. Note that it may not work with very old compilers or with newer ones where older standards such as C99 are chosen via compiler flags.

```
gets_s(string, MAX_STR_LEN + 1);
```

Another problem is that `fgets()` includes the newline character at the end of the string, which we usually don't want.

There is unfortunately no POSIX standard function that safely reads a line of text without appending the newline character to the string.

Addendum: The `xt_fgetline()` in libxtend (https://github.com/outpaddling/libxtend/) provides this functionality as conveniently as possible. The libxtend library adds many convenient low-level functions that some would say are missing from the standard C libraries.

String output to `stdout` can be performed using `puts()`, which automatically adds a newline character:

```
// We can leave off the array size if we provide an initializer
// More on this in the arrays chapter
char    commentary[] = "It's a beautiful day!";

puts("Hello, world!");
puts(commentary);
```

If you don't want a newline added automatically (as when you read a string with `fgets()`, which retains the newline read from the stream), you can use `fputs()`:

```
fputs("Hello, world!\n", stdout);
fputs(commentary, stdout);
putchar('\n');                  // fputs() didn't add this
```

### 7.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What problems might occur when using `gets()` and why?
2. Write a C program that asks the user for their name, and displays a simple greeting. Be sure to test this and all programs before submitting.

```
What is your name? Joe Piscopo
Hey, Joe Piscopo
```

## 7.4 Numeric I/O

### 7.4.1 Output with printf()

Character, string, and numeric output can all be printed using the `printf()` library function. This function takes a format string as the first argument, and a variable number of additional arguments. The format string must contain a *place holder*, A.K.A. *format specifier*, for each argument after the format string. The specifier must match the data type of the argument. Common specifiers are listed in Table 7.1. For more information, run **man 3 printf**. Running **man printf** will show the section 1 (Unix commands) printf. Section 3 contains man pages for standard library functions.

| 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 7.1: Format specifiers for printf()

For numbers, `printf()` converts the internal binary number (unsigned integer, two's complement, or floating point) to a string of characters, which are then sent to the stream output buffer as when using `putchar()`.

Values of type `char` and `short` are promoted to `int` when passed to a function. Hence they use the same format specifiers as `int`. Likewise, `float` values are promoted to `double`, so we use "%f" for both. "%lf" is equivalent to "%f" for `printf()` (though not for `scanf()`, described later). "%Lf" is used for `long double`.

```
// We deliberately use non-descriptive variable names here since
// these are meant to be arbitrary
char    a;
short   b;
int     c;
float   x;
double  y;

printf("%d %d %d\n", a, b, c);
printf("%f %f\n", x, y);
```

Preceding any of the integer specifiers with "l" indicates a `long` and "ll" indicates a `long long`.

```
long        a;
long long   b;

printf("%ld %lx %lld %llx\n", a, a, b, b);
```

**Caution**

The `printf()` function should not be used where a simpler function would suffice. I.e., to print a single character, use `putchar()` and to print a simple string, use `puts()` or `fputs()`, unless you need to control the format of the character or string output.

The `printf()` function scans the format string looking for format specifiers, which is wasteful if there are none. This will make no noticeable difference in performance for many programs, but it may also add up to significant CPU time in some cases.

```
    printf("\n");   // Overkill
    putchar('\n');  // More efficient

    printf("Hello, world!\n");  // Overkill
    puts("Hello, world!");      // More efficient

    printf("Please enter your name: ");         // Overkill
    fputs("Please enter your name: ", stdout);  // More efficient
```

**Caution**

On some systems (mostly 32-bit systems), `int` and `long` are the same size. Hence, we can easily interchange "%d" and "%ld" without consequences. However, the same code will not work properly on most 16-bit and 64-bit systems. This is one reason it's always a good idea to test your code on multiple operating systems, multiple CPU architectures, and multiple compilers (e.g. clang and gcc).

Format specifiers can be preceded by an optional field width, and in the case of floating point, the number of decimal places. Numbers are normally right-justified. We can indicate left-justification with a '-'.

```
    int     a = 34;
    double  x = 91.234;

    printf("%6d %10.2f\n", a, x);       // Default right-justified
    printf("%-6d %-10.2f\n", a, x);     // Force left-justified
```

```
Output with spaces indicated by '_':

____34_____91.23
34____91.23_____
```

Types defined in `stdint.h`, such as `int64_t` and `uint64_t` need special handling. On 64-bit systems, `int64_t` is equivalent to `long`. On 32-bit systems, it is equivalent to `long long`. The header `inttypes.h` defines constants that are platform-sensitive for this purpose, and will translate to "ld" or "lld" as appropriate for the platform. We need to use string constant concatenation for this:

```
    int64_t     big_num;

    printf("big_num = %" PRId64 ".\n", big_num);
```

### 7.4.2 Input with scanf()

The `scanf()` function is the converse of `printf()`. It reads character input from `stdin` and converts it to the formats indicated by the specifiers in the format string.

Understanding `scanf()` requires a quick preview to C argument passing and *pointers*. All arguments in C are *pass by value*, which means the function receives a copy of the value of the argument.

**Note** You may see the expression "call by value" in some texts. This doesn't make sense, since arguments are not *called*. Functions (methods, subprograms) are *called* and arguments are *passed* to them.

I.e., in the code below, `printf()` does not know the *memory address* of the variable `x`, and hence it is impossible for `printf()` to modify its value. The value 5.1 is copied from x in the calling function to a local variable in the `printf()` function.

```
double      x = 5.1;

printf("x = %f\n", x);
```

The `scanf()` function cannot work this way. The whole purpose of `scanf()` is to modify the values of the arguments in the calling function.

In some languages, we can define the arguments as *pass by reference*, which means the local variable in the called function is an alias, or reference to the argument in the calling function. I.e. they are two names for the same memory address. Effectively this means that the *address* of the argument is passed rather than the value. In C, there are no pass by reference arguments, but we can achieve the same thing by explicitly passing the address of a variable by value, using the '&' (address of) operator:

```
int     age;

fputs("How old are you? ", stdout);

// Pass the address of the age variable instead of the value
// so scanf() can change its content
scanf("%d", &age);
```

**Note** Some languages, such as Fortran, pass all arguments by reference. Fortran allows the programmer to tag certain argument variables in a subprogram as read-only, to protect the calling subprogram from side-effects. Other languages, such as C++, support both value and reference arguments.

The format specifiers for `scanf()`, listed in Table 7.2, are not quite the same as those for `printf()`.

| 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 7.2: Format specifiers for scanf()

One of the reasons they differ from `printf()` specifiers is that we always pass *addresses*, rather than *values* to `scanf()`. Addresses are all the same size and format (unsigned integers, usually matching the CPU's word size), regardless of what data type they point to. E.g., the address of an `int` is the same as the address of a `double`. Hence, there are no promotions when passing addresses to a function. The `scanf()` does need to know the size of format of the data *at the address*. We therefore need different specifiers for `char`, `short`, and `int` in `scanf()`. Likewise for `float` and `double`.

```
short   a;
int     b;
float   x;
double  y;

scanf("%hd %d %f %lf", &a, &b, &x, &y);
```

---

**Note** The `scanf()` function is very flexible about whitespace in the input. A space in the format string matches any number of spaces, tabs, or newlines in the input. Hence, in the statement above, the user could enter numbers on the same line separated by any number of spaces or tabs, as well as all or some of them on different lines.

---

Run **man scanf** for a full list of specifiers. There is no **scanf** Unix command, so no need for a section number here.

### 7.4.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Why do `char`, `short`, and `int` all use the same placeholders in `printf()`?
2. Do "%d" and "%ld" mean the same thing?
3. Why does `scanf()` not use the same format specifiers for `char`, `short`, and `int`, like `printf()`?
4. Where do we find a complete list of format specifiers for `printf()` and `scanf()`?
5. Write a C program that asks the user for the length and width of a rectangle and prints the area and perimeter.

```
Please enter the length and width of the rectangle: 2 3
The area is 6.000000 and the perimeter is 10.000000.
```

## 7.5 Using fprintf() for Debugging

Debugging a program (locating errors when it is not behaving correctly) can be very frustrating and time-consuming. There are many tools available to help debug programs. Unfortunately, many are not portable and most are difficult to use.

*Caveman debugging*, so called because it's a primitive method, is often the easiest and most useful approach. It involves making the program "talk", by printing intermediate results.

Debug output should be sent to `stderr` rather than `stdout` for two reasons:

- The `stderr` stream is by default unbuffered on most systems. This means that output is sent directly to the screen rather than lingering in a memory buffer until the buffer is full. If the program crashes shortly after printing a message to `stdout`, that message might die in the buffer, never making it to the screen, leading you to believe that the crash happened *before* the debug print, when in fact it happened after.

  We can correct for this by running `fflush(stdout)` immediately after printing debug messages to `stdout`, to flush whatever is in the output buffer to the destination device. This is normally not necessary if we use `stderr`.

```
// Using stderr
fprintf(stderr, "Done with loop.  sum = %d\n", sum);

// Using stdout
printf("Done with loop.  sum = %d\n", sum);
fflush(stdout);
```

- Sending debug output to `stderr` allows it to be separated from normal output using redirection. The normal output can then be examined without interference.

```
# Debug output still goes to the screen if it is printed to stderr
shell-prompt: ./myprog > normal-output.txt
```

### 7.5.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is caveman debugging? Is it obsolete?
2. To where should we print debug output?

## 7.6 Addendum: Robust I/O

Every input statement, or any other statement that might fail, should be wrapped in a conditional, which we will introduce in Chapter 9, to confirm that it succeeded, before proceeding to the next statement. Obviously, if an input or output statement fails, we don't want the program to simply continue as if nothing is wrong. For now, just note that the examples above, which do not verify success, are not robust. Proper C input is briefly demonstrated by the code below:

```
        // Sloppy code that ignores input errors
        scanf("%d", &line_count);

        // Robust use of input functions
        // scanf() returns the number of items successfully converted
        if ( scanf("%d", &line_count) != 1 )
        {
            fputs("Input error: Expected an integer.\n", stderr);
            exit(EX_DATAERR);
        }

        // Another alternative is to keep asking until the user gets it right
        while ( scanf("%d", &line_count) != 1 )
        {
            fputs("Input error: Expected an integer.  Please try again.\n", stderr);
            fpurge(stdin);  // Discard the rest of the input line
        }
```

The one exception to this rule is output to the terminal. Output functions like `putchar()`, `puts()`, and `printf()` do return a value that indicates success or failure, but we normally don't check it. We *do* always check when writing to a file, so that we can detect disk full or other write errors. This is covered in Chapter 21.

### 7.6.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Why should we always test for the success of input and output functions?
2. Are there any exceptions to the "always check" rule?

# Chapter 8

# C Statements and Expressions

## 8.1 Simple Expressions

An expression in C is anything with a value.

C does not hide (abstract) the way data are handled by hardware. There are no abstract types built into the language. Everything is either an integer, a floating point value, or just arbitrary bits. Boolean values are integers. 0 means false, non-zero means true. All abstract data types are defined by the programmer using type definitions.

Other languages distinguish between assignment statements and expressions. I.e., the '=' operator can only appear after an *lvalue* (variable or other mutable object on the LHS (left hand side) of a complete statement. Boolean expressions are distinct from numeric expressions in some languages.

In C, there is no separation between operator types (Boolean, arithmetic, etc.) All operators and data types can be mixed in the same expression. Expressions like the example below may look like nonsense, but the design philosophy of C is "trust the programmer". The language does not forbid anything unless it simply cannot work.

```
int     c = 5, d;

// Multiply c by 0 if c <= 1 or >= 10, assign 5 to d and add to c
// Java does not allow mixing Boolean expressions with integers this way
c = c * ((c > 1) && (c < 10)) + (d = 5);
```

Any expression followed by a ';' is a valid statement. Again, C does not enforce rules that do not need enforcement. This would needlessly complicate the compiler.

```
    3;                  // No purpose, but valid
    x == 5;             // No purpose, but valid
    y = x + (z = 10);   // Assigns 10 to z, then adds to x and assigns sum to y
```

### 8.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What abstract data types does C provide?
2. What is a statement in C?

## 8.2 C Operators

Most operators are *polymorphic* (operate on multiple types, with possibly different results). Division (/) is the most obvious example:

```
1 / 2       = 0
1.0 / 2.0   = 0.5
```

Integer and floating point addition are not the same thing either. Floating point is more complex and requires different machine instructions.

*Precedence* refers to which of two adjacent operators is evaluated first, without regard for order. Multiplication has higher precedence than addition, so it is always done before an adjacent addition:

```
// Multiplication comes first whether it is before or after
// addition because it has a higher precedence
y = x + a * 5;  // Same as y = x + (a * 5);
                // Same as y = a * 5 + x;
```

If two operators have the same precedence, *associativity* determines order of evaluation.

```
y = z + a - 5;  // + and - have same precedence
                // Left-to-right associativity means + happens first
```

### 8.2.1 Unary and Binary Operators

*Unary operators* have one operand.

```
y = -6;     // Unary minus (negation)
```

*Binary operators* have two operands.

```
y = x - 6;  // Binary minus (subtraction)
```

### 8.2.2 Math Operators

Math operators follow algebra and most other languages. (Actually, most modern languages follow C.) Table 8.1 outlines the C math operators.

| **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 8.1: Basic Math Operators in C

The '/' operator produces different results for integers than for floating point. The '+', '-', and '*' operators are also polymorphic, i.e. they do unsigned binary, two's complement, or floating point operations, depending on the operand types. However, the results are generally the same value regardless of type for operators other than division.

```
y = 1 / 2;       // 0
y = 1.0 / 2.0;   // 0.5
y = 1 / 2.0;     // 0.5
```

The remainder (%) operator works only for integers. There is no remainder in real division.

```
y = 3 % 5;       // 3
y = 3.0 % 5;     // Nonsensical
```

Instructor: Make up some mixed expressions using math operators including '/' and verbally quiz students on the value.

The ++ and -- operators add or subtract 1 from any integer value. When embedded in an expression, it matters whether they come before or after the variable.

```
int     c = 1, d;

// Assume the following statements are independent, not sequential
++c;         // c = 2
c++;         // c = 2

d = ++c;     // d = 2, c = 2      (pre-increment)
d = c++;     // d = 1, c = 2      (post-increment)
```

### 8.2.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is 7 / 2?
2. What is 7 % 2?
3. What is 7.0 / 2.0?
4. What are the values of a, b, c and d after the following code segment?

```
int     a, b, c, d;

a = 5;
b = 7;
c = a++;
d = ++b;
```

## 8.3 Mixed Expressions

The ability to freely mix data types is both a convenience and a curse. It saves some typing, but makes programs harder to predict, reduces performance, and can lead to serious errors (incorrect output) when used incorrectly.

When values of two different data types are operands to the same operator, the "lower" type is promoted to the "higher" type. "Higher" generally corresponds to more bits or more range, as shown in Table 8.2.

Implicit promotions occur in most modern languages, such as C, C++, Java, Fortran, etc.

The table does not show `short` or `char` because they are *always* promoted to `int` when used with math operators, even when not mixing. They are also promoted when passed to a function.

Example: Tracing a mixed expression evaluation:

| Type | Rank |
|---|---|
| double complex | 1 (highest) |
| float complex | 2 |
| double | 3 |
| float | 4 |
| unsigned long long | 5 |
| long long | 6 |
| unsigned long | 7 |
| long | 8 |
| unsigned int | 9 |
| int | 10 |

Table 8.2: Data Type Ranks

```
int     a = 5, b = 1;
long    c = 10;
float   x = 1.2f;
double  y = 2.0;

a = a / c + 4 * x - b / y;
```

What actually happens while evaluating this expression:

1. 5 in a (int two's comp) promoted to long 5L two's comp via sign-extension
2. a / c = 0L (long two's comp)
3. 4 (int two's comp) promoted to 4.0F (32-bit IEEE)
4. 4.0F * x = 4.8F
5. int 1 (two's comp) in b promoted to 1.0 (64-bit IEEE)
6. 1.0 / 2.0 = 0.5
7. 0 (two's comp) promoted to 0.0F (32-bit IEEE)
8. 0.0f + 4.8f = 4.8F
9. 4.8F (32-bit IEEE) promoted to 4.8 (64-bit IEEE)
10. 4.8 - 0.5 = 4.3
11. 4.3 (64-bit IEEE) truncated and demoted to int 4 (two's comp)
12. 4 assigned to a

Half of the steps above are data conversions. Only half perform useful work.

In addition, the integer division results in 0. This is more likely an oversight on the part of the programmer than intentional when expressions contain floating point.

Converting between signed and unsigned integers of the same size has no cost. The bits are not changed. Only the interpretation of the result is in question. Different machine instructions may be generated for signed and unsigned operations, but no additional instructions to convert the values will be inserted.

### 8.3.1 Explicit Conversions: Casts

We can force a conversion if we don't want the default operation to occur by stating the type in () before the value:

```
a = (double)a / c + 4 * x - b / y;

// Same as

a = (double)a / (double)c + 4 * x - b / y;

// This does not help
a = (double)(a / c) + 4 * x - b / y;
```

### 8.3.2 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is a promotion and when does it occur?
2. What is a demotion and when does it occur?
3. When are `char` and `short` values promoted?
4. List all the promotions, demotions, and mathematical operations that occur when evaluating the following expression. Indicate the data type of each intermediate result using standard C constants.

```
char        a = 2, b = 6;
long        c = 4, d = 10;
double      x = 9.0, y = 3.0;
float       z;

z = b / a * c / d + x * y;
```

5. Alter the expression above so that no integer divisions occur.

## 8.4 Bitwise Operators

| 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 |
| ^ | Bitwise XOR | 6 | Left to right |
| \| | Bitwise OR | 6 | Left to right |
| [operator]= (=, +=, -=, etc.) | Assignment | 7 | Right to left |

Table 8.3: Bitwise Operators in C

```
char            a = 0x0f,   // 00001111_2
                b,
                c = 0x80;   // 10000000_2
unsigned char   d = 0x80;
```

```
    b = a & 0x18;   // 00011000 & 00001111 = 00001000 = 0x08
    b = a | 0x18;   // 00011000 | 00001111 = 00011111 = 0x1f
    b = a ^ 0x18;   // 00011000 ^ 00001111 = 00010111 = 0x17
    b = ~a;         // 00001111' = 11110000 = 0xf0
    b = a ^ -1;     // 00001111 ^ 11111111 = 0xf0

    b = a << 2;     // 00001111 << 2 = 00111100 = 0x3c
    b = a >> 2;         // 00001111 >> 2 = 00000011 = 0x03

    b = c >> 4;         // 10000000 >> 4 = 11111000 = 0xf8 arithmetic shift
    b = d >> 4;         // 10000000 >> 4 = 00001000 = 0x08 logical shift
```

Shown vertically, so we can see how the bits align:

```
    0f  00001111    00001111    00001111
&   18  00011000  | 00011000  ^ 00011000
----------------    --------    --------
    08  00001000    00011111    00010111
```

An series of arithmetic shifts starting with 10000000.

```
10000000    = 01111111 + 1 = -10000000 = -128   // 10000000
11000000    = 00111111 + 1 = -01000000 = -64    // 10000000 >> 1
11100000    = 00011111 + 1 = -00100000 = -32    // 10000000 >> 2
```

With bitwise AND, OR, and XOR, we can think of one operand as the *value* and the other as the *mask*. Bits that are 0 in a mask are *reset*, or *cleared* in the result of an AND, while other bits in the value are unchanged. Bits that are 1 in the mask are *set* in the result of an OR, while other bits are unchanged. Bits that are 1 in the mask are inverted in the result of an XOR, while other bits are unchanged.

```
    unsigned char   a = 0x1c, b;    // 0001 1100

    // Clear the lowest 4 bits in a and assign the result to b
    b = a & 0xf0    // 0001 0000 = 0x10

    // Set bits 0 and 1 in a and assign the result to b
    b = a | 0x03;   // 0001 1111 = 0x1f

    // Invert the upper 4 bits in a, and assign the result to b
    b = a ^ 0xf0;   // 1110 1100 = 0xdc
```

If the value being shifted is a signed integer, the >> operator performs an *arithmetic shift*, preserving the sign bit. This divides a two's complement value by 2, e.g. 1000 = -8, 1100 = -4.

If the value is an unsigned integer, the >> operator performs a logical shift (leftmost bit cleared/reset to 0). This divides an unsigned value by 2, but does not work on two's complement.

Bit shifts require only 1 clock cycle on most CPUs. Multiplication with '*' and division with '/' may require many clock cycles, since they perform repeated addition or subtraction. If multiplying by a power of 2 or a short sum of powers of 2, using shift can speed up the program.

```
    y = y * 32;         // Several clock cycles for repeated addition
    y = y << 5;         // 1 clock cycle
```

### 8.4.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is the hexadecimal value of `c` after the following? Try to do it in your head or on paper first, then check by writing a 2-line program that prints `c`.

```
unsigned char   c = 0xf0 >> 3;
```

2. What is the hexadecimal value of `c` after the following?

```
char   c = 0xf0 >> 3;
```

3. What is the hexadecimal value of `c` after the following?

```
unsigned char   c = 0x1c & 0xf2;
```

4. What is the hexadecimal value of `c` after the following?

```
unsigned char   c = 0xf0 ^ -1;
```

5. What is the hexadecimal value of `c` after the following?

```
unsigned char   c = 0xfc | 7;
```

6. What operator and mask would you use to clear bits 2 and 3 of a char?
7. What operator and mask would you use to set bits 0 and 7 of a char?
8. What will happen if you accidentally use a logical operator such as `||` in place of a bitwise operator such as `|`?
9. Write a C program that inputs an integer and prints the value times 2, the value times 4, and the value times 8, using the most efficient method possible.

```
Please enter an integer: -2
-2 * 2 = -4
-2 * 4 = -8
-2 * 8 = -16
```

## 8.5 Addendum: More Performance Tricks

### 8.5.1 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;
```

### 8.5.2 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Rewrite the following expression to make it compute faster:

```
double  x, y;
int     a;

y = 4.5 * x * x * x - 5.3 * x * x + a * 32;
```

# Chapter 9

# Decisions with If and Switch

## 9.1 Program Flow

Two kinds of statements in C:

- Expressions followed by a semicolons.
- Flow control. Sometimes called "control flow", which doesn't really make sense. Our TVs don't have "control volume", they have "volume control". Cars and motorcycles don't have "control throttle", they have "throttle control".

  Flow control statements alter the normal top-to-bottom flow of statements, usually *conditionally*, based on some Boolean expression.

### 9.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What are the two kinds of statements in a C program and what do they do?

## 9.2 Boolean Expressions

### 9.2.1 Boolean Type

A *Boolean expression* is an algebraic expression with a value of true or false. The primary Boolean operators AND, OR, and NOT, behave much like mathematical operators -, +, and unary - (negation).

Addendum: Until the C99 standard, C had no Boolean data type. In C99 and later, `_Bool` is a keyword, though it is not used directly. We include `stdbool.h` and use the `bool` type, along with the constants `true` and `false`.

```
#include <stdio.h>
#include <sysexits.h>
#include <stdbool.h>

int     main()

{
    bool    is_cloudy = true;
```

```
    ...

    return EX_OK;
}
```

The `bool` type is actually an integer, typically equivalent to `char`. A value of 0 is interpreted as false, and any other value as true. In fact, any integer type can be used in place of `bool` in flow control statements.

### 9.2.2 Relations

Most Boolean expressions are formed from *relations*, comparisons of non-Boolean values, using the operators in Table 9.1.

| Operator | Relation |
|---|---|
| > | Greater than |
| >= | Greater than or equal |
| < | Less than |
| <= | Less than or equal |
| == | Equal |
| != | Not equal |

Table 9.1: Relational Operators

### 9.2.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is a Boolean expression?
2. What are the primary Boolean operators?
3. How are Boolean values treated in C?
4. How are most Boolean expressions formed?

## 9.3 The if-else Statement

### 9.3.1 Statement Syntax

The `if-else` statement is used to conditionally execute one or more statements.

```
    if ( Boolean-expression )
        statement1;
    else
        statement2;
```

The `else` is optional.

```
    if ( scanf("%d", &list_size) != 1 )
        fputs("Error reading list_size, expected positive integer.\n", stderr);
```

**Caution**

Never use == or != with floating point values. Round-off error may cause variables that are equal in theory to be unequal in reality. Other relational operators may be unreliable due to round-off error as well. Make sure you understand your data and the consequences of its inaccuracy to your code.

```c
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    double  x = 1.0 / 10.0;

    printf("%0.17f\n", x);
    if ( x == 0.1 )
        puts("x is 0.1");

    return EX_OK;
}
```

Output:

```
0.10000000000000001
```

**Caution**

A very common mistake in C is using = where == was intended. This will *assign* a value, rather than *compare*. If the value assigned is 0, the expression will be interpreted as false. If it is non-zero, then the expression is seen as true.

```c
    if ( aardvarks = 1 )    // Assigned 1 to aardvarks, always "true"
    {
        ...
    }
```

Use of assignments can be useful, however:

```c
    int     ch;

    // If not end-of-file, process new character in ch
    if ( (ch = getchar()) != EOF )
    {
        ...
    }
```

### 9.3.2 Compound Statements

To group statements under an `if` or `else`, we use a *compound statement*, which can actually be used anywhere.

```c
#include <stdio.h>
#include <sysexits.h>

int     main()

{
    {
        // A useless compound statement
        puts("Hello, world!");
    }
```

```
    return EX_OK;
}
```

```
    if ( Boolean-expression )
    {
        statement1;
        statement2;
    }
    else
    {
        statement3;
        statement4;
    }
```

Statements under an `if` or `else` should be consistently indented. The lab manual covers such coding standards in detail.

**Example 9.1** Example of `if-else`

```
/* FreeBSD Compile: cc -Wall prog -lm */
/* Linux Compile: cc -Wall prog -lm */
/* SunOS Compile: cc prog -lm */
/* SCO_SV Compile: cc -v prog -lm */

#include <stdio.h>  /* Input and output */
#include <sysexits.h>
#include <math.h>   /* Math functions - sqrt() */

int     main()

{
    double  a,b,c,root1,root2,discriminant,sq,two_a;

    printf("Enter coefficients A, B, and C: ");
    if ( scanf("%lf %lf %lf",&a,&b,&c) == 3 )
    {
        discriminant = b*b - 4.0*a*c;
        if ( discriminant >= 0.0 )
        {
            /* 2*a and sqrt(discriminant) used twice */
            /* so we pre-compute and use variables */
            two_a = 2.0 * a;
            sq = sqrt(discriminant);

            /* Find roots */
            root1 = (-b + sq) / two_a;
            root2 = (-b - sq) / two_a;
            printf("Roots are %f and %f.\n",root1,root2);
        }
        else
            printf("Sorry, no real roots.\n");

        return EX_OK;
    }

    fputs("Error: Please enter the 3 coefficients separated by whitespace.\n",
            stderr);
    return EX_DATAERR;
}
```

Think you got it? Prove it! Try doing it yourself without looking at the text.

### 9.3.3 Building Bigger Boolean Expressions

Boolean algebraic expressions can be constructed using the Boolean operators in Table 9.2.

| C Operator | Boolean Operation |
|---|---|
| && | AND |
| \|\| | OR |
| ! | NOT |

Table 9.2: Boolean Operators

```
if ( (artichokes == 0) && (artichokes_sold >= 1) )
    puts("Time to order more artichokes.");
else
    puts("We're good on artichokes for now.");
```

### 9.3.4 De Morgan's Rules

Boolean NOT can be distributed and factored out. We must toggle AND and OR in the process:

```
! (A && B) is the same as !A || !B
! (A || B) is the same is !A && !B
```

### 9.3.5 Nested `if` Statements

When nesting `if-else` statements, be very careful about indentation to avoid making the code misleading. Using curly braces is often a good idea for readability even if they are not necessary.

```
// Sloppy and misleading
if ( artichokes == 0 )
    if ( artichokes_sold >= 1 )
        printf("Time to order more artichokes.\n");
else
    printf("Artichokes aren't selling.\n");
else
    printf("We're OK on the artichokes for now.\n");
```

```
if ( artichokes == 0 )
    if ( artichokes_sold >= 1 )
        printf("Time to order more artichokes.\n");
    else
        printf("Artichokes aren't selling.\n");
else
    printf("We're OK on the artichokes for now.\n");
```

```
if ( artichokes == 0 )
{
    if ( artichokes_sold >= 1 )
        printf("Time to order more artichokes.\n");
    else
        printf("Artichokes aren't selling.\n");
}
else
    printf("We're OK on the artichokes for now.\n");
```

### 9.3.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What do we need to watch out for when using floating point in relational expressions?
2. Should we use == or = in an `if` expression?
3. What is a compound statement and where can we use one?
4. How should we format `if-else` blocks?
5. Write a C code segment that checks to see of the value of variable `count` is between 0 and `MAX_COUNT` (inclusive), and if not, prints an error message stating that it's out of range.
6. Write a C code segment that checks to see if the variable `temperature` is above `MAX_TEMPERATURE`. It should then check another variable `hot_time`, print a warning if it is greater than `WARNING_TIME` minutes, and call the function `shutdown()` if it is greater than `CRITICAL_TIME`.
7. Simplify the following `if` statement using De Morgan's rule:

```
if ( ! ((a < 10) || (a > 20)) )
    statement;
```

## 9.4 Switch

A `switch` statement can replace a series of `if-else` statements, but only for *integral* (discrete) data types, i.e. not floating point.

```
if ( c == 1 )
    statement1;
else if ( c == 2 )
    statement2;
else
    statement3;

switch(c)
{
    case    1:
        statement1;
        break;

    case    2:
        statement2;
        break;

    default:
        statement3;
        break;
}
```

The `break` statements are not required, but usually necessary. If we omit the `break` after `statement1`, then both `statement1` and `statement2` will be executed when `c == 1`. This is occasionally useful, but would not be equivalent to the `if-else` above.

C compilers will attempt to translate a **switch** statement to a jump table at the machine code level. Rather than sequentially compare the switch variable (`c` above) to each case value, the addresses of each statement are stored in an array, and the switch variable (or some hash of the switch variable) is used as a subscript to that array. This makes jumping to any case instantaneous rather than a linear search through the case values.

### 9.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Write a C code segment that compares an integer variable `color` to the constants RED, GREEN, and BLUE, and prints a string matching the color in each case.

## 9.5 The Conditional Operator

The conditional operator is essentially an `if-else` statement implemented as an expression.

```
    if ( a < 0 )
        b = -a;
    else
        b = a;
```

```
    b = a < 0 ? -a : a;
```

### 9.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Write a C statement that assigns `z` the minimum of `x` and `y`, using only a conditional operator.

## 9.6 Performance

### 9.6.1 Programmer Psychology

Studies have shown that 60% of `if` conditions are false. This is due to a human tendency to think in terms of the less likely alternative. "If there's a tornado" vs "If there's not a tornado". Compilers optimize machine code so that the `else` clause is faster in order to take advantage of this statistic. Because of the way machine/assembly language works, one clause or the other must use an extra jump instruction:

```
    if ( c == 1 )
        a = 5;
    else
        a = 7;
```

```
        // The if clause is 2 instructions after the bne
        // The else clause is only 1
        bne     c, 1, else
        li      a, 5
        jmp     done
else:   li      a, 7
done:
```

### 9.6.2 Minimizing Boolean Expression Evaluation

Boolean operators are evaluated left to right. If the expression left of `&&` is false, then the expression on the right is irrelevant and will not be evaluated. Likewise, if the expression left of an `||` is true, then the expression on the right is not evaluated.

```
if ( (n >= MIN) && (n <= MAX) )
{
    ...
}
```

If `n < MIN`, then the relation `n <= MAX` will not be evaluated. If it is more likely that `n` is greater than MAX than it is to be less than MIN, we should rearrange the condition above to maximize the likelihood that the second relation is unnecessary.

```
// Put the less likely relation first when using AND
if ( (n <= MAX) && (n >= MIN) )
{
    ...
}
```

Likewise, when using OR, we want the more likely relation first.

### 9.6.3 Using Data Types to Reduce Boolean Expressions

```
int     n;

if ( (n >= 0) && (n <= MAX) )
{
    ...
}
```

An unsigned value is always greater than 0, so if we cast `n` to `unsigned`, then `(n >= 0)` becomes useless.

```
int     n;

// n may be less than 0, but we cast to unsigned to eliminate a compare
if ( (unsigned int)n <= MAX )
{
    ...
}
```

This works because negative numbers in two's complement become very large positive numbers when cast to unsigned, beyond the range of the two's complement system. E.g., -1 is all 1111...11, the largest possible unsigned value, well beyond 0111...11, the largest signed value. The constant MAX cannot be greater than the largest possible signed value, so all negative numbers cast to unsigned will be greater than MAX.

This is one of many clever tricks we can use to improve performance. We should consider, however, whether the benefit to performance outweighs the cost to readability.

If you use a trick like this, it *must* be clearly documented in a comment.

### 9.6.4 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Write an optimal `if-else` statement that prints "Just another day in the cheese state." if the temperature is below 90 F (32 C) in Milwaukee, or "It's a hot one in Wisconsin." if it isn't. Explain your code in a comment.
2. Write an optimal `if-else` statement that checks the temperature in Milwaukee in January, and prints "Better bring the beer in." if the temperature is below 30 F (-1 C) or above 50 F (10 C), and "It's OK to leave the beer outside" otherwise. Explain your code in a comment.

# Chapter 10

# Repetition: Loops

## 10.1 Loops and Performance

Loops should be avoided as much as possible.

Addendum: 1990s Pentium: 20 million instructions per second. 2023 Intel/AMD processors: Billions of instructions per second.

### 10.1.1 Loops and Performance

#### The Execution Path

Sequence of instructions executed. Different from sequence in the program file due to flow control.

Two ways to speed up a program:

- Shorten the execution path (run fewer machine instructions)
  - Eliminate loops or shorten loops by using more efficient algorithms, e.g. binary search vs linear search.
    Watch out for hidden loops in functions that must use iterative methods, like `sin()`, `sqrt()`, etc. Avoid string processing. Strings are arrays of characters (in all languages, not just C). Replace strings with scalar integers if possible.
  - Reduce the amount of code inside the loop. Many existing programs have code inside loops that can simply be moved out.
  - Simplify expressions inside the loops. This reduces the amount of machine code generated.
- Replace machine instructions with faster instructions E.g. use shift instead of multiply:

```
    c *= 32;    // Multiplication is repeated addition, takes many clock cycles
    c <<= 5;    // Shift takes 1 clock cycle
```

### 10.1.2 Performance Measurement

Unix **time** command measures three main statistics:

- User time: Time a process spends using the CPU
- System time: Time the kernel spends using the CPU on behalf of a process
- Real time or wall time: Time elapsed on the clock while a process is running.

```
# Example
shell-prompt: time ./myprog < input.txt > output.txt
```

The **top** command monitors processes as they run and displays periodic updates on their resource use.

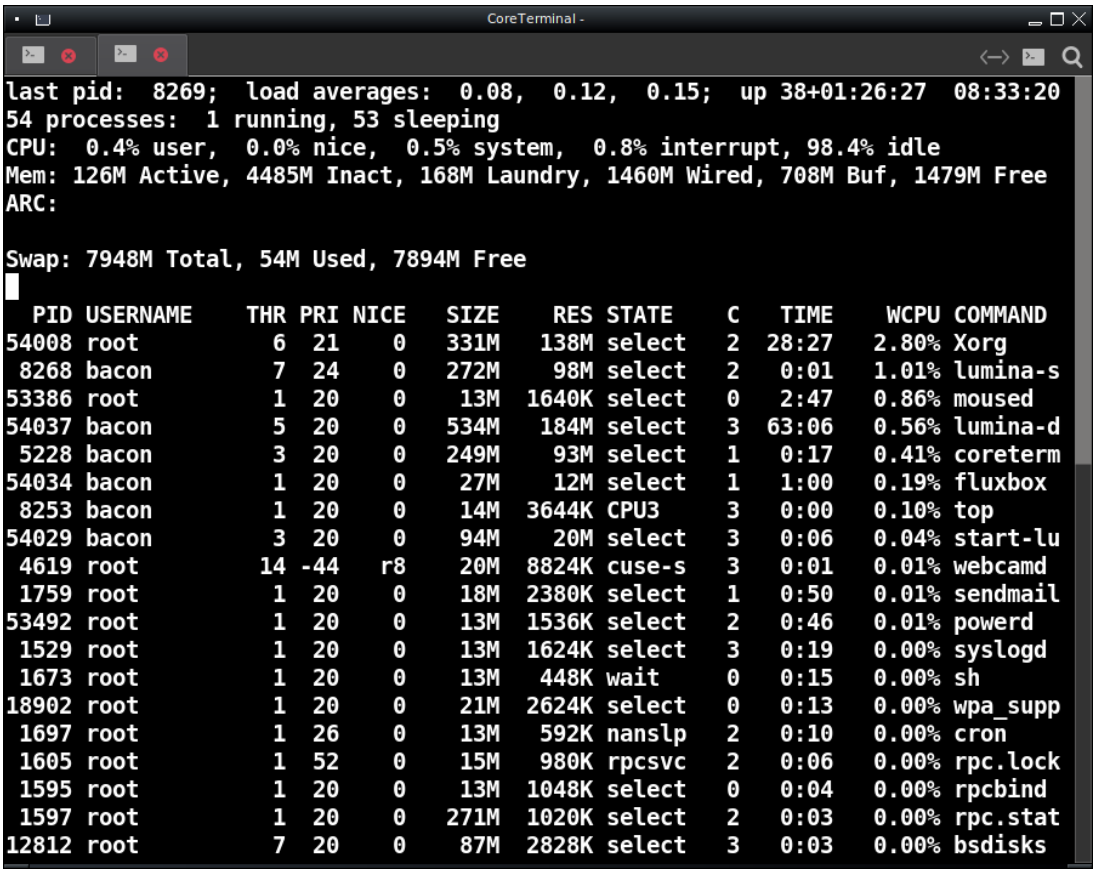



Figure 10.1: The top Command

### 10.1.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is the execution path?
2. What are the two general ways to speed up a program? ( 5 to 10 words each )
3. How can we measure the total run time of a Unix process?
4. How can we monitor resource use while a process is running?

## 10.2 The Universal Loop: while

While is the only loops we need. It can do anything that other loops do.

```c
#include <stdio.h>
#include <sysexits.h>

int     main()
{
    int     c;

    /* Initialize loop variable (prime the loop) */
    c = 1;

    /* Continue until c == 11 */
    while ( c <= 10 )
    {
        // Body
        printf("%d squared is %d\n", c, c * c);

        // Housekeeping
        ++c;
    }

    return EX_OK;
}
```

**Caution**
An *infinite loop* occurs when a loops never terminates, such as if we omit the `++c` above. Infinite loops have many possible causes.

Checking the loop condition and updating a loop variable are overhead costs, i.e. they take time, but do not compute any results. Loops can be *unrolled*, which means turned into a sequence of statement blocks instead of one block inside a loop. This eliminates loop overhead at the cost of more machine code. It is generally only desirable where performance is critical. The **clang** and **gcc** compilers will automatically unroll loops at the machine code level if you specify `-funroll-loops` or related options.

Code in loops should be indented consistently, as with conditionals. See the coding standards in the lab manual for details.

**Caution**
Since Boolean expressions are integers, and 0 means false, C programmers sometimes do things like the following:

```c
    int     c;

    // Count down from 10 to 1
    c = 10;
    while ( c )
    {
        printf("%d\n", c);
        --c;
    }
```

This will work, but there is no advantage to it, and it forces the reader to examine more code in order to understand what the while condition is doing. It's better to make your code easy to read than show off how clever you are as a C programmer. The code below makes it clear at a glance that the variable `c` is a number being compared to 0.

```c
    while ( c > 0 )
    {
        ...
    }
```

### 10.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What often causes an infinite loop?
2. What is loop overhead? How can we eliminate it, and at what cost.

## 10.3 The do-while Loop

The `do-while` loop is like a `while` loop, but checks the condition at the end instead of the beginning.

This means that a `do-while` always iterates at least once. It eliminates the need to *prime* the loop (force the condition to be true).

```
    c = 0;  // Prime the loop
    while ( c < 10 )
    {
        ++c;
    }
```

**Note** The term "prime" is borrowed from old carburetor-based cars, which were often difficult to start. People would prime the engine by spraying a flammable liquid into the carburetor before turning the engine over.

**Note**
A `do-while` is slightly faster than a while, since it eliminates an unconditional jump at the machine code level. A `while` checks the condition at the beginning, and must unconditionally jump back there from the end:

```
        // Hypothetical while loop assembly language
while:  beq x, 10, done     // Jump out of loop if c == 10

        // Loop body

        jmp while           // Unconditional jump overhead
done:

        // Hypothetical do-while loop assembly language
        // No unconditional jump overhead
do:     // Loop body

        blt c, 10, do       // Jump back to beginning if c < 10
```

Smart compilers may in some cases convert a `while` loop to a `do-while` if it is clear that it will iterate at least once anyway. Don't count it this, though. Write it as a `do-while` if possible.

### 10.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. How does the behavior of a `do-while` loop differ from that of a `while`?
2. Do `while` and `do-while` loops exhibit the same performance? Why or why not?

## 10.4 The for Loop

A `for` loop in many languages is a more limited kind of loop that iterates through integer or other enumerable values. Not so in C. In C, the `for` loop is actually a `while` loop with the initializer, condition, and housekeeping all collected into one place for easy reading. This makes the code more cohesive and makes it harder to forget to include the housekeeping (which is very easy to do with a while loop).

```
c = 0;              // Initialization
while ( c < 10 ) // Condition
{
    // Loop body

    ++c;            // Housekeeping
}

// Initialization, condition, housekeeping all together
// Another example of cohesive code
for (c = 0; c < 10; ++c)
{
    // Loop body
}
```

---

**Note**

Only the condition is required in a `for` loop. Either or both of the initialization and housekeeping can be omitted.

```
c = 0;              // Initialization
for ( ; c < 10; ) // Condition
{
    // Loop body

    ++c;            // Housekeeping
}
```

---

**Caution**
As stated earlier, we should never compare floating point values with == or !=, and we must use extreme caution when comparing them with other relational operators.

```
    double      x;

    for (x = 0.0; x != 1.0; x += 0.1)
    {
        printf("%f\n", x);
    }
```

```
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
...
And on forever...
```

Using `x < 1.0` does not completely solve the problem, either. It will still result in one extra iteration.
One solution is to use integers instead of floating point wherever possible. If we must use floating point, we should always allow a tolerance when doing comparisons.

```
    double      x, tolerance;

    // Stop when x is very close to 1.0 to accommodate round-off
    tolerance = 0.05;
    for (x = 0.0; ABS(x - 1.0) > tolerance; x += 0.1)
    {
        printf("%f\n", x);
    }
```

### 10.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. How does a `for` loop differ from a `while` loop?
2. How do direct floating point comparisons affect loops? How do we solve this problem?
3. Write a C program that prints the square root of every integer value from 1 to 100.

## 10.5 Nested Loops

```
    // Statements here are executed once.
    // These are the least important to optimize
```

```
for (row = 0; row < rows; ++row)
{
    // Statements here executed "rows" times
    // More important to optimize

    for (col = 0; col < cols; ++col)
    {
        // Statements here executed rows * cols times
        // Most important to optimize
    }
}
```

### 10.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. When optimizing a program with nested loops, where should we focus our efforts?

# Chapter 11

# Functions

## 11.1 Subprograms for Modularity

Turn one big job into a bunch of little jobs and knock them off one at a time, in a logical order.

Top-down design for cleaning the kitchen:

- Do the floor last, since it may get dirty while we clean things above it.
- Clean the stove
- Clean the table
- Wash the dishes
- Clean the sink

Top-down design for sorting a file:

- Read the file into an array (sorting directly on disk would be very inefficient, since disk access takes roughly 100,000 to 1,000,000 times as long as memory).
- Sort the list in memory using the most efficient available sorting algorithm. E.g. for selection sort (not the most efficient, but simple for the sake of this example):
  - Find the smallest element in the list
  - Swap the smallest element with the first
  - Repeat the above steps for the remaining elements
  - Continue repeating until only one element remains
- Output the sorted list.

### 11.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Write a top-down design for identifying lines in a file that contain a string provided by the user (i.e. how the **grep** command works).

## 11.2 Reusability and Encapsulation

Some subprograms are very esoteric and will never be used in another program. This is typical of higher level functions, e.g. those that are called directly by the main program.

Other subprograms may be more generally *reusable*. This is more likely for lower-level subprograms, especially *leaf* subprograms (subprograms that do not call any other subprograms).

Subprograms that might be useful in other programs should be eventually moved to a *library*, a collection of precompiled subprograms that are directly accessible to any program via the linker. Building libraries is covered in Chapter 20.

---

**Note** In my 35+ years of C programming, about 2/3 of all the functions I have written have gone into libraries. I often develop and test functions as part of a program, making them as general as possible with the intent of sharing them with other programs eventually, and once they are fully tested, move them to a library such as libxtend (https://github.com/outpaddling/libxtend) or biolibc (https://github.com/auerlab/biolibc).

---

*Encapsulation* is the practice of bundling data types with the operations that can be performed on the data, to form a *class*. Conceptually, the integer set and the operations +, -, *, and / form a class. We can (and should) try to encapsulate most of our derived data types as well. This is the focus of *object oriented design*.

In implementing an object oriented design, we restrict access to data to a group of *member functions*, i.e. functions that are considered part of the class. This can be done in any language with a little self-discipline, but object oriented languages such as Java and C++ provide syntactic support.

Object-oriented programming in C is covered in Chapter 18.

For now, each time you write a new function, think about whether it could go in a library and whether it should be a member of a class. Classes are often implemented as libraries.

### 11.2.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How can we avoid duplicating the effort of writing subprograms that are useful to more than one application?
2. What is encapsulation?

## 11.3 Writing Functions

Subprograms go by many names, such as *procedures* in Pascal and *subroutines* in Fortran, both of which are subprograms that do not return a value. The `writeln` procedure is a Pascal built-in procedure for writing a line of output to the terminal.

```
    writeln('Hello, world!');
```

*function* is a subprogram that returns a value. A call to a function is embedded in an expression. An example would be the `sin()` function:

```
    y = x * x + sin(theta) - 4.0;
```

All subprograms in C are called functions, because they all have a return type. Functions that don't return a value have the special return type `void`, but are still called functions. In C, we can choose to ignore the return type of a function and treat it like a Pascal Procedure or Fortran subroutine. We should almost never ignore the return value of a function, but one case where it's OK is when sending output to the terminal: E.g. the `printf()` function returns an `int`, the number of items successfully written.

```
// printf() prototype:
int     printf(char *format, ...);
```

```
    printf("Average = %f\n", average);
```

Functions in C are both *declared* and *defined*. A definition includes the body (executable statements) of the function, while the declaration defines only the *interface* (the arguments and return value types).

### 11.3.1 Function Definitions

We will focus on the ANSI / ISO standard for function definitions. The older K & R standard is still generally supported, but almost never used. K & R (Kernighan and Ritchie) is the old de facto C standard predating ANSI and ISO standards.

Every function should have a block comment above describing what the function does, and optionally, its modification history. Then comes the function header defining the return type and formal argument variables, and finally the body.

```
/*****************************************************************************
 *  Description:
 *
 *  History:
 *  Date        Name        Modification
 *  2023-03-13  Joe C. Unix Begin
 *****************************************************************************/

return-type name(formal argument variables)

{
    body

    return expression;
}
```

```
/*****************************************************************************
 *  Description:
 *      Compute and return the square of an integer.
 *      Detecting overflow is left to the user to maximize speed here.
 *
 *  History:
 *  Date        Name        Modification
 *  2023-03-13  Joe C. Unix Begin
 *****************************************************************************/

int     square(int n)

{
    return n * n;
}
```

The return type must be a *scalar* (dimensionless, single-value) type. C functions cannot return *aggregate* types (which contain multiple values) such as arrays or structures. They can, however, return *pointers* to (addresses of) aggregate types. This ensures efficiency, since returning an aggregate type would involve a loop copying potentially large amounts of data at the machine code level.

**Example 11.1** Function example

Below is a simple example of a C function. Note that the function is defined before is it called in main().

```
#include <stdio.h>
#include <sysexits.h>
```

```c
// Define the factorial function before calling it in main

double  factorial(int n)

{
    double  f = 1;

    while ( n > 1 )
        f *= n--;

    return f;
}


int     main()

{
    int     c;

    for (c = 0; c < 10; ++c)
        printf("%d! = %f\n", c, factorial(c));

    return EX_OK;
}
```

A *prototype* is a C function declaration that defines the return value and argument list of a function. This is useful when a function cannot be defined before it is called, or when it is defined in a different source file. Prototypes are usually placed in header files, which can then be included in any source file. A prototype is identical to the header of a function definition, except that it does not need to include the variable names.

C compilers are *one-pass compilers*, meaning that they only read the source file once. The compiler must see either a declaration or a definition of a function before it sees the first call to that function, so it knows the interface.

The old K & R function declarations did not include an argument list. These should not be used anymore, since all modern compilers support prototypes.

**Note** Run **more /usr/include/stdio.h** and scroll down to see the prototypes for standard stream functions such as getchar(), putchar(), printf(), scanf(), etc.

**Example 11.2** Function example

Below is the same example using a prototype to allow the factorial function to be defined later.

```c
#include <stdio.h>
#include <sysexits.h>

// Declare the factorial function before calling it in main
// A declaration that defines the argument list is called a prototype
// Variable names are optional

double  factorial(int);

int     main()

{
    int     c;

    for (c = 0; c < 10; ++c)
        printf("%d! = %f\n", c, factorial(c));
```

```
    return EX_OK;
}


double  factorial(int n)

{
    double  f = 1;

    while ( n > 1 )
        f *= n--;

    return f;
}
```

**Note**
Did it occur to you that the examples above are terribly inefficient? After computing N!, where N is any integer from 1 to 9, the program *recomputes* N! in the process of computing (N+1)!.
It would be more efficient not to use a function, but to compute N! in main() by simply multiplying the previous value by N.
An efficient factorial function requires more knowledge of C and is presented in Chapter 15.

### 11.3.2 Function Calls

The value of an expression in a function call is copied to the formal argument variable in the function. Table 11.1 shows how the variable `c` in main() and the variable `n` in the factorial function might, hypothetically, be organized in memory.

| Memory address | Variable | Value |
|---|---|---|
| 1000 | c (main) | 2 |
| 1004 | n (factorial) | 2 |

Table 11.1: Memory Map

### 11.3.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Why do we need prototypes in C?
2. Where are most prototypes found? Why?
3. Write a C program that prints a 10 x 10 multiplication table. Use a function called `prod()` that returns the product of the two arguments. Place the function definition after `main()` and a prototype before.

```
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
```

```
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

## 11.4 Local Variables

Variables defined inside a function only exist in that function. I.e., their *scope* is limited to that function. A variable with the same name defined elsewhere is a different variable.

```
/*
 *  n and f are in scope only from where they are defined to the
 *  end of the factorial function.
 */

unsigned long   factorial(unsigned long n)

{
    unsigned long   f;

    for (f = 1; n > 1; --n)
        f *= n;

    return f;
}


/*
 *  The variable n below is not the same variable as n in factorial()
 */

int     square(int n)

{
    return n * n;
}
```

### 11.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What does *scope* mean and of what is it a property?
2. Can a C program have multiple variables with the same name? Why or why not?

## 11.5 Arguments

### 11.5.1 Privacy

Variables in a function that receive arguments are called *formal argument variables*.

The only difference between a formal argument variable and a local variable is that the formal argument is initialized to a value received from the caller. Otherwise, formal argument variables are just like other local variables.

```
unsigned long   factorial(unsigned long n)

{
    // c here is useless.  We could just use n in its place, since
    // they are equivalent.
    unsigned long   c, f;

    for (c = n, f = 1; c > 1; --c)
        f *= c;

    return f;
}
```

### 11.5.2 Argument Passing

All C arguments are *passed by value*, meaning that the value of the argument in the caller is copied to the formal argument variable in the called function.

```
#include <stdio.h>
#include <sysexits.h>

unsigned    power(unsigned base, unsigned exponent)

{
    unsigned    p;

    for (p = 1; exponent >= 1; --exponent)
        p *= base;

    return p;
}

int     main()

{
    unsigned    exponent;

    for (exponent = 1; exponent <= 10; ++exponent)
        printf("2^%u = %u\n", exponent, power(2, exponent));

    return EX_OK;
}
```

In the program above, there are two separate variables called `exponent`, one in `main()` and one in `power()`. The variables `base` and `exponent` in `power()` receive copies of the values `2` and `exponent` in `main()`, as shown in Table 11.2.

| Address | Variable |
|---|---|
| 1000 | exponent (main) |
| 1004 | base |
| 1008 | exponent (power) |

Table 11.2: Argument Passing

Some languages, such as C++, support *pass by reference*, where the formal argument variable in the subprogram becomes an alias for (has the same memory address as) the argument passed by the caller. This can lead to *side effects*, where a variable is accidentally modified by a subprogram that was called. Other times, we *want* a subprogram to modify variables in the caller, as with `scanf()` or a `swap()` function.

Since C does not support pass by reference arguments, we must explicitly pass the address of a variable we want modified by a function:

```
// Passing the address of a variable by value simulates passing the
// variable by reference
scanf("%d %d", &a, &b);
swap(&a, &b);
```

Passing arguments this way is covered in Chapter 14.

**Caution** The terms "call by value" and "call by reference" are often used in place of "pass by value" and "pass by reference". They don't really make sense, since we do not call arguments, we call subprograms. Don't let mixed terminology like this confuse you. Functions are called, and arguments are passed to them.

### 11.5.3 Promotions in Argument Passing

If no prototype or function definition has defined an argument type, then arguments of type `char` and `short` are promoted to `int` when passed to a function, and arguments of type `float` are promoted to `double`. This is why `printf()` uses "%d" for `char`, `short`, and `int`, and "%f" for both `float` and `double`. The prototype for `printf()` only lists the format string, so the compiler does not know the types of other arguments.

We *can* define formal arguments of type `char`, `short`, and `float`, but it is not usually a good idea as these values are likely to be promoted implicitly within the function, reducing program performance.

### 11.5.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. How do formal argument variables differ from local variables in a function?
2. How are C arguments passed?
3. Draw a possible memory map showing the memory addresses, variable names, and contents of the arguments in `main()` and all variables in `pow()` below after `pow()` is called, but before the body begins executing. Be sure to account for the size of each variable and the fact that memory is byte addressable. Assume a starting address of 1000.

```
double  pow(double base, unsigned exponent)

{
    double  p;
    ...

    return p;
}


int     main()

{
    double      b = 2.0;
    unsigned    e = 10;

    printf("%f\n", pow(b, e));

    return EX_OK;
}
```

4. When are arguments promoted?

## 11.6 Library Functions

As stated early in this text, C is a minimalist language with only essential high-level language features and *no* predefined functions.

Most of the functionality of C comes from *libraries*, archives of precompiled functions added to our programs during the link stage of **cc**. The designers of the C language deliberately left out any feature that could be implemented as a function written in C.

C programming does not have to be a low-level experience if we utilize functions properly as a substitute for high-level features in other languages, such as vector operations in Matlab, Python, or R.

Each C compiler and operating system provides a huge collection of functions in the *standard libraries* along with prototypes and supporting data types and constants in the accompanying standard header files. The standard libraries are found under `/lib` and/or `/usr/lib`, and the standard headers under `/usr/include`.

Some Unix-compatible systems, such as most GNU/Linux distributions, place add-on libraries and headers in the same directories. Others, such as FreeBSD, keep all add-ons separate, e.g. under `/usr/local`.

Addendum: FreeBSD 13.1 the standard C library, `/usr/lib/libc.a`, contains 1,487 functions, compared to the 570 stated in the book from FreeBSD 2.1. The standard math library, `/usr/lib/libm.a`, contains 279 (150 in FreeBSD 2.1).

```
shell-prompt: nm /usr/lib/libm.a | awk '($2 == "T") && ($3 !~ "^_")' | wc -l
```

All standard library functions are documented in the man pages. Additional documentation may be available in other forms, such as GNU **info**. Sometimes there is a C function and a Unix command with the same name. In this case, only the Unix command will be shown by a standard man command. To see all documented features with that name, use the `-a` flag:

```
shell-prompt: man -a printf
```

### 11.6.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Where does most functionality come from an C programs?
2. How can we get information about the `chmod()` library function, given that there is a Unix command with the same name?

## 11.7 Documenting Functions

Each function should be documented with a block comment describing what the function does.

The function name by itself should provide a clear idea of what the function returns. If you cannot sum up what a function does in a simple name, then the function is doing too much and your code is not cohesive. The function should be split into two or more separate functions that can each be described with a simple name.

### 11.7.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How can we tell if our function is not cohesive?

## 11.8 Top-down Programming and Stubs

A *stub* is a skeletal function with a complete interface (arguments and return value), but without the function body implemented. A stub allows us to compile and test the function call before writing the function. This way, we can eliminate errors in the interface first, and then focus entirely on writing the function body, knowing that the interface is correct. Otherwise, if a function returns the wrong value, we don't know whether the interface or the body is to blame.

This is how we extend our best practice of very frequent, incremental testing to programs that contain new functions. If we do this consistently, debugging a program will never be difficult.

Suppose we want to print the square root of every integer from 1 to 100 and the C library did not provide a square root function.

```
#include <stdio.h>
#include <sysexits.h>

double  my_sqrt(double n);

int     main(int argc,char *argv[])

{
    double  x;

    for (x = 0.0; x <= 10.0; ++x)
        printf("sqrt(%0.1f) = %f\n", x, my_sqrt(x));

    return EX_OK;
}


/*
 *  Stub for sqrt(), allows compiling and testing before completion
 *  to test the caller and the interface.
 */

double  my_sqrt(double x)

{
    // Return any value that shows the arguments were received correctly
    // If there is more than one argument, we could return the sum of
    // all of them, for example

    return x;
}
```

We can now test the interface before proceeding to write the function body:

```
sqrt(0.0) = 0.000000
sqrt(1.0) = 1.000000
sqrt(2.0) = 2.000000
sqrt(3.0) = 3.000000
sqrt(4.0) = 4.000000
sqrt(5.0) = 5.000000
sqrt(6.0) = 6.000000
sqrt(7.0) = 7.000000
sqrt(8.0) = 8.000000
sqrt(9.0) = 9.000000
sqrt(10.0) = 10.000000
```

```
#include <stdio.h>
#include <sysexits.h>
#include <math.h>
```

```c
double  my_sqrt(double n);

int     main(int argc,char *argv[])

{
    double  x;

    for (x = 0.0; x <= 10.0; ++x)
        printf("sqrt(%0.1f) = %f\n", x, my_sqrt(x));

    return EX_OK;
}


/*****************************************************************************
 *  Description:
 *      Compute the square root of any non-negative real number x.
 *
 *  Returns:
 *      Square root of x >= 0, -1 if x < 0
 *
 *  History:
 *  Date        Name        Modification
 *  2023-03-13  Jason Bacon Begin
 *****************************************************************************/

double  my_sqrt(double x)

{
    // Use static so this is only initialized once at compile time
    const static double  tolerance = 0.00000001;
    double  guess, next_guess;

    if ( x < 0 )
        return -1;

    // Crude initial guess.  The closer we get to sqrt(x), the
    // fewer iterations the function will need to converge.  There
    // are more sophisticated initial guess algorithms available.
    next_guess = x / 2.0 + 0.1; // Add .1 to avoid divide by 0

    /*
     *      Estimate a square root using the Babylonian method.
     *      This is a numerical analysis method that computes successively
     *      better guesses given a reasonable initial guess.
     */
    do
    {
        guess = next_guess;

        // Babylonian formula for next guess
        next_guess = (guess + x / guess) / 2.0;

        // Loop until difference between guesses <= tolerance
        // Note: Using the fabs() function entails function call overhead.
        // A better solution would be a macro, which is covered in the
        // cpp chapter.  Iterative functions like this are expensive enough,
        // so we should do all we can to make them more efficient.
    }   while ( fabs(next_guess - guess) > tolerance );

    return next_guess;
```

```
}
```

```
sqrt(0.0) = 0.000000
sqrt(1.0) = 1.000000
sqrt(2.0) = 1.414214
sqrt(3.0) = 1.732051
sqrt(4.0) = 2.000000
sqrt(5.0) = 2.236068
sqrt(6.0) = 2.449490
sqrt(7.0) = 2.645751
sqrt(8.0) = 2.828427
sqrt(9.0) = 3.000000
sqrt(10.0) = 3.162278
```

### 11.8.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Write a stub for a `pow()` function that will eventually return a real base raised to a non-negative integer exponent.

## 11.9 Advanced: Recursion

C has the lowest function call overhead of any high-level language. This minimizes the overhead penalty of recursive functions.

The shorter the run time of the function body, the higher the overhead penalty. Below is a recursive factorial function. Since this function is so short, it is not a good candidate for recursion. The function call overhead, even in C, will exceed the time spent running the function body. The iterative version shown earlier is not efficient either. The only efficient way to implement a factorial function is using a lookup table, which is covered in Section 15.6.

```
/*
 *  This function does minimal computation, so the function call
 *  overhead accounts for most of the total run time.  Hence,
 *  this is not a good candidate for recursion.  It is a good
 *  academic example, however, since it is simple and easy to
 *  understand.
 */

unsigned long factorial(unsigned long n)

{
    if ( n < 2 )
        return 1;
    else
        return n * factorial(n - 1);
}
```

```
// Suppose we call factorial(5)

Address n   return
1000    5   5 * factorial(4)
1008    4   4 * factorial(3)
1016    3   3 * factorial(2)
1024    2   2 * factorial(1)
1032    1   1
```

### 11.9.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Write a recursive function to compute a power of a real base raised to a non-negative integer exponent.

## 11.10 Advanced: Scope and Storage Class

### 11.10.1 Scope

*Scope* refers to which part of a program can access a variable. In C, variables can be defined with a scope limited to any block statement, such as in an if block. If another variable with the same name exists with a broader scope, the one with the narrower scope is used.

**Caution** Instantiating local variables entails a small cost at run time. Hence, defining all variables that the start of a function will lead to slightly faster code than defining some of them in narrower blocks, such as loops, where they will be instantiated and destroyed multiple times.

```
int     c = 5;

if ( c == 5 )
{
    // This variable c is separate from the one used in the if condition
    // and takes precedence within the if block
    int     c = 2;

    printf("%d\n", c);  // prints 2, not 5
}
```

We can also define a variable in a for loop header, to save a line of code:

```
for (int c = 1; c <= 10; ++c)
    printf("%d\n", c);
```

### 11.10.2 Storage Class

#### Segments

The *storage class* of a variable or other memory block determine how, where, and when it is *instantiated* (allocated memory).

Every process running under Unix has its memory space segmented as shown in Table 11.3.

| Segment | Purpose |
|---|---|
| Text | Machine code |
| Data | Static variables |
| Stack | Auto variables, saved data |
| Heap | User-allocated memory |

Table 11.3: Machine Language Program Segments

Where memory address space is limited, the stack and heap can grow toward each other in order to utilize all available space. On 64-bit computers with virtual memory, this is not necessary, since there is more address space than we can possibly populate with any kind of real storage. $2^{64}$ = 1.8 * $10^{19}$, about 18 billion gigabytes, or over a billion 16 gibibyte memory modules. A typical PC only takes about 4 memory modules.

### Static

A static variable uses space in the data segment and is instantiated when the process is born. The variable remains in existence until the process terminates. A static variable retains its contents across function calls.

If a static variable definition has an initializer, it is initialized only once, when the process is born.

```
void count_calls()

{
    static int calls = 1;

    printf("I've been called %d times.\n", calls);
    ++calls;
}
```

**Caution** *Global* variables, i.e. variables defined outside of any function, are also static. Global variables should not be used in application programming, since they cause *side effects*, where what happens in one function affects the behavior of functions called in the future, without passing them information via arguments. This type of programming is not modular. Some system code, such as device drivers, may require the use of global variables, but there is always a more modular way to code applications.

### Auto

The *auto* storage class is the default for local variables in all functions (including main). We can use the keyword `auto` when defining variables, but there is no need to, since it is the default.

```
    auto int c;     // Same as "int c;"
```

Auto variables are instantiated in the stack segment at run-time, when the block containing the variable definition is entered. The space is automatically freed when the block is exited.

**Note** Auto variables entail a very small amount of overhead at run time, as the stack pointer must be updated each time auto variables are instantiated or destroyed. Static variables do not incur this overhead.

```
#include <stdio.h>
#include <sysexits.h>

void    fn2(void)

{
    int     c;

    printf("Address of c in fn2:  %p\n", &c);
}


void    fn1(void)
```

```
{
    int     c;

    printf("Address of c in fn1:  %p\n", &c);
    fn2();
}


int     main(int argc,char *argv[])

{
    int     c;

    printf("Address of c in main: %p\n", &c);
    fn1();

    return EX_OK;
}
```

| System Stack |
|---|
| Local variables for main() |
| Local variables for fn1() |
| Local variables for fn2() |

Table 11.4: Memory Map of Local Variables

```
Output of the program when run on an Intel Core i5 (64-bit) processor:

Address of c in main: 0x7fffffffe71c
Address of c in fn1:  0x7fffffffe718
Address of c in fn2:  0x7fffffffe714
```

If an `auto` variable has an initializer, it is initialized every time it is instantiated, i.e. every time the block defining it is reentered. This means that a local auto variable in a function always starts with the initial value, unlike `static` variables. It also adds to run time every time the block is entered.

#### Register

The `register` storage class requests that a variable be associated with a CPU register rather than a main memory address. Registers are much faster than memory.

This storage class is obsolete, since modern compilers can utilize registers much better at the machine code level than we can at the source code level. For example, the same register might be used to "cache" different variables (whichever one is being most heavily used) in different sections of the same function.

#### Const

The `const` modifier marks a variable as read-only. This forbids assigning it a new value, which will trigger a compiler error.

Variables marked `const` can only be assigned a value via an initializer.

Formal argument variables marked `const` receive a value as an argument and cannot be modified by the function.

```
void    function(const int n)

{
    const int   c = 1;
```

```
    c = 2;  // Compiler error
    n = 3;  // Compiler error
}
```

#### Volatile

The `volatile` modifier informs the compiler that a variable could be modified by external forces at any time. The compiler will not cache a copy of the variable in a register for better performance, since the copy in the register could become out-of-date at any moment.

Volatile variables are only necessary where code that modifies the variable may be executed *asynchronously*, e.g. as part of a *signal handler* or *event handler*. A signal handler is a function that is not called explicitly by the program, but in response to an *event* such as a key press or mouse click. This topic is covered in Section 28.2.

### 11.10.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What are the 4 segments of an executable and what do they contain?
2. When are static variables initialized? Auto variables?
3. When should we use the `register` storage class?

## 11.11 Advanced: The inline Request

The `inline` function modifier is a request to the compiler to eliminate function call overhead, but copying the machine code of the function body to the locations of the function calls, rather than inserting code that jumps to and returns from the function.

Inlining is particularly useful for very short, fast functions that are called many times. For larger, longer-running functions, the overhead of a function call tends to be trivial compared to the time spent running the body, so inlining has little effect. For functions only called a few times, the function call overhead does not add up.

Addendum: Modern compilers generally inline small functions automatically where possible, so this explicit request generally has little effect.

### 11.11.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What effect does inlining a function have?

# Chapter 12

# Programming with make

## 12.1 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
```

### 12.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 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?

## 12.2 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
```

### 12.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 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?

## 12.3 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 12.1: Standard Make Variables

### 12.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 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".

## 12.4 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
```

---

### 12.4.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 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?

## 12.5 Using Header Files

The C language was designed to separate code into *source files* and *header files*.

Now that you know how to build an executable from multiple source files with make, we can separate our code as it was intended.

Source files (ending in .c) should generally contain only function definitions, and `#includes` that include header (.h) files. Only header files should be included with `#include`, never source (.c) files.

Header files (ending in .h) should contain virtually all of our constant definitions, type definitions, function prototypes, and *anything else that might be useful to more than one source file*.

### 12.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What kind of code belongs in a source file (.c)?
2. What kind of code belongs in header files?

## 12.6 Makefile Generators

Addendum: **genmake**, **imake** no longer popular. They have been largely replaced by GNU **configure** and **cmake**.

Makefile generators aim to locate dependencies (e.g. libraries or tools on which a build depends) and automatically detect non-portable features of the operating system on which a program is being built. In some cases, they may even download and build dependencies for you.

The goal is to relieve end-users building the program from manual labor. The problem is, they almost never work consistently. Developers creating a **configure** or **cmake** script cannot possibly foresee all of the variables it will encounter on other peoples' computers, many of which have unique configurations. Attempts to make such scripts work reliably usually result in *feature creep* (A.K.A. *creeping feature syndrome*), where the script becomes increasingly complex in response to problem reports from end users. When such a script fails, the user is left with a nightmarishly complex problem.

Modern Unix systems offer an alternative to avoid this situation in the form of *package managers*, such as Debian Linux's apt, the FreeBSD ports system, MacPorts for macOS, the portable pkgsrc package manager that works on virtually any POSIX platform, and Redhat Linux's Yum. There are many others.

Package managers are highly evolved build systems that leverage the collaboration of thousands of experienced developers in order to maintain high-quality builds in a tightly controlled environment. They automatically manage dependencies as separate installations and incorporate patches to ensure a clean build of each package. This results in far more reliable software deployment than ad hoc builds using the upstream developers' **configure** or **cmake** script.

Simple Makefiles can be very portable if we know how to use **make** properly. If we leave dependency management to a package manager, we can provide a simple, reliable Makefile for our project that will require minimal maintenance.

### 12.6.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the goal of makefile generators?
2. What is the main problem with makefile generators?
3. What is an alternative to makefile generators that leads to less problematic software deployment?

# Chapter 13

# The C Preprocessor

The C preprocessor, **cpp**, is an example of a *stream editor*. It inputs text and outputs a modified version of the same text. Other examples of stream editors include the standard Unix commands **sed** and **m4**.

The **cpp** command is automatically run by **cc** to filter C source code before compilation.

Changes to the source code are made according to *cpp directives* embedded in the input, such as `#define` and `#include`.

---

**Note**
We can view the output of the C preprocessor by running **cc -E prog.c**. It is usually helpful to paginate it using more: **cc -E prog.c | more**.

---

## 13.1 Macros and Constants: #define

Any identifier created with `#define` is technically called a *macro*.

```
#define MAX_NAME_LEN    100
```

We can also define a macro in the compile command, using the `-D` flag:

```
shell-prompt: cc -DMAX_NAME_LEN=100 prog.c
```

C preprocessor macros can also take arguments, however, and hence act like functions.

```
#define ABS(x)   ((x) < 0 ? -(x) : (x))

int     main()

{
    int     a, b;

    a = ABS(b);

    ...
}
```

Preprocessor output:

```
int     main()

{
    int     a, b;
```

```
    a = ((b) < 0 ? -(b) : (b))

    ...
}
```

Many familiar library "functions" are actually macros defined in the headers:

```
shell-prompt: grep getchar /usr/include/stdio.h
#define getchar()       getc(stdin)
```

---

**Caution**

Macro arguments should always be enclosed in () in the text, as should the entire text of the macro.

```
#define ABS(x)  x < 0 ? -x : x

int     main()

{
    // Expands to y = x - 5 < 0 ? -x - 5 : x - 5 * 2;
    y = ABS(x - 5) * 2;

    ...
}
```

Problem 1: -x - 5 is not the same is -(x - 5).
Problem 2: This multiplies 5 * 2 instead of ABS(x - 5) * 2.

```
#define ABS(x)  ((x) < 0 ? -(x) : (x))

int     main()

{
    // Expands to y = ((x - 5) < 0 ? -(x - 5) : (x - 5)) * 2;
    // which is what we want.

    y = ABS(x - 5) * 2;

    ...
}
```

---

---

**Caution**

Another issue for which there is no hands-off solution is duplicating auto-increment and auto-decrement operations:

```
    y = ABS(x++);   // Expands to y = ((x++) < 0 ? -(x++) : (x++));
```

The variable `x` gets incremented twice in either case. This will not happen if ABS() is implemented as a function. This is why macros are conventionally named in all upper-case. At least then the programmer knows whether they are calling a function or a macro, and can watch out for side effects like this one.

---

Unlike C, the C preprocessor is line-oriented. If we want a macro to span multiple lines, we must use continuation characters at the end of all but the last line:

```
#define ABS(x) \
    ((x) < 0 ? -(x) : (x))
```

### 13.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Show a command to view the preprocessor output for the file prog1.c.
2. Show how to define the constant DEBUG with a value of 1, both within a program and from the command line.
3. Show how to define a macro called INVERSE that produces the mathematical inverse of a number.
4. What operators should we avoid using in macros? Why?
5. Why should we use parentheses extensively in macro definitions?

## 13.2 Functions vs. Macros

When a program calls a function, it must spend time passing arguments, jumping to, and returning from the function. This is known as *function call overhead*. Macros are faster than functions, because they have no such overhead. An inlined function behaves more like a macro.

For very small functions that execute quickly and are called many times, function call overhead can be significant. This would likely be the case for an absolute value function. For longer-running functions or functions that are only called a few times, the overhead is not significant.

Another advantage to macros over C functions is that they are polymorphic (type-independent), mostly.

```
#define ABS(x)   ((x) < 0 ? -(x) : (x))

int      main()

{
    int      a, b;
    double   x, y;

    a = ABS(b);
    y = ABS(x);

    ...
}
```

**Note**
In C, we would need to define a separate function for each data type to achieve the same effect. This is frequently done in the standard C libraries and headers:

```
shell-prompt: grep abs /usr/include/math.h
double  fabs(double);
float   fabsf(float);
long double     fabsl(long double);
```

In C++, we can *overload* functions, i.e. create multiple functions with the same name, but different interfaces (argument and return value types). Another option in C++ is a *template* function, in which case we define only one function, but with flexible argument and return types. Template functions incur a cost at run time, since they are generally just-in-time compiled as each version is needed.

### 13.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Why are macros typically faster than functions? When are they not?
2. How can we help programmers using our macros avoid side-effects like those caused by ++ and --?

## 13.3 Header Files: #include

The `#include` directive inserts a header file into the source code in place of itself.

System headers, usually found in `/usr/include`, are enclosed in angle brackets:

```
#include <stdio.h>
```

Headers that are part of our project, and typically found in the current working directory, are enclosed in double quotes:

```
#include "myheader.h"
```

Virtually all macro definitions, type definitions, and prototypes should reside in header files, so that they can be reused from multiple source files. Source (.c) files, should generally only contain `#includes` and function definitions.

If we wish to use a function definition in multiple different projects, placing it in a header file is not the right way to do it, since it leads to many problems such as duplicate definitions during compilation. Functions that need to be shared should be precompiled and placed in a *library*, covered in Chapter 20.

### 13.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the difference between angle brackets and double quotes in `#include`?
2. What should we put in ".c" source files and what should we put in headers?

## 13.4 Advanced: Conditional Compilation

Conditional compilation allows us to exclude sections of code from being compiled. This is often better than disabling it with an `if` statement, since it reduces the size of the executable and eliminates the cost of the `if` check at run-time.

### 13.4.1 #if

The `#if-#endif` directive pair is an `if` statement that operates during preprocessing, rather than during program execution. If the expression provided has a non-zero value, then the code within the `#if-#endif` is output by **cpp** and passed onto the compiler.

We can use this to conditionally include debug code in a build, for example"

```
shell-prompt: cc -DDEBUG=1 prog.c
```

```
// Caveman debug code to track list_size during execution
// If DEBUG is defined and non-zero, the fprintf() is passed onto
// the compiler.
#if DEBUG
    fprintf(stderr, "list_size = %zu\n", list_size);
#endif

    // This, on the other hand, is always compiled and the if test
    // incurs a small overhead cost at run time
    if ( DEBUG )
        fprintf(stderr, "list_size = %zu\n", list_size);
```

### 13.4.2 Preprocessor Operators

The `#if` directive accepts the same operators as a C Boolean expression, such as ==, !=, etc. One possible use is to support multiple levels of debug code:

```
// Debug level 1 or higher, basic debug output
#if DEBUG >= 1
    fprintf(stderr, "list_size = %zu\n", list_size);
#endif

// Debug level 2 or higher, more extensive debug output
#if DEBUG >= 2
    fprintf(stderr, "list[list_size] = %d\n", list[list_size]);
#endif
```

### 13.4.3 #ifdef, #ifndef, and defined()

The `#ifdef`, `#ifndef`, and `#if defined()` constructs simply check whether a macro is defined, regardless of its value.

```
shell-prompt: cc -DDEBUG prog.c
```

```
#ifdef DEBUG
    fprintf(stderr, "list_size = %zu\n", list_size);
#endif

// Equivalent to above
#if defined(DEBUG)
    fprintf(stderr, "list_size = %zu\n", list_size);
#endif
```

The `#if defined()` construct is useful when we want to check multiple macros. We can use Boolean operators as we do in a C Boolean expression.

```
#if defined(MACRO1) && defined(MACRO2)

#endif
```

### 13.4.4 Improving Portability

Many predefined macros exist to provide the compiler information about the build environment, such the operating system name and version, which is indicated by a macro such as `__linux__`, `__FreeBSD__`, or `__APPLE__`.

```
#if defined(__FreeBSD__) || defined(__linux__)

// Some code that only works on FreeBSD and Linux as far as we know

#endif
```

This sort of coding is generally to be avoided in favor of writing more portable code in the first place. Most existing operating systems, including BSD, Linux, and macOS are more than 99% POSIX-compliant. There is rarely a need to write non-portable code in application programming if we avoid OS-specific extensions and write POSIX-compliant code.

### 13.4.5 Nesting #include Efficiently

Suppose `file.c` includes `header1.h` and `header2.h`, and `header1.h` includes `header2.h`, because it needs a macro or prototype defined there. This is a common situation. If not dealt with, everything in `header2.h` will be processed twice, and hence redefined, leading to compiler errors and warnings.

To prevent the problems caused by redundant inclusions, system headers usually have their entire contents guarded by a macro, to avoid redefining everything they contain when they are included more than once:

```
// In header2.h
#ifndef _HEADER2_H_
#define _HEADER2_H_

// Contents of header2.h

#endif
```

This prevents everything in the header from being processed a second time, but **cpp** still has to read through the file again, which slows down compilation. To prevent the file from even being read again, we can guard the `#include`:

```
// In header1.h:
#ifndef _HEADER2_H_
#include <header2.h>
#endif
```

This prevents **cpp** from even opening and reading `header2.h` if it has been included previously. This should only be necessary in header files, where we can't predict what may have already been included upstream. In a C source file, we need only include `header1.h`, since it will pull in `header2.h` for us.

### 13.4.6 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Show how to print "list_size = " followed by the value of `list_size`, only if the macro DEBUG is defined.
2. When should we use platform-detection predefined macros such as `__FreeBSD__` and `__linux__`?

## 13.5 Advanced: Other Directives

The `#error` directive terminates preprocessing and issues an error message:

```
#ifndef __UNIX__
    #error "This program is only for Unix-compliant platforms."
#endif
```

The `#pragma` directive is an extensible interface for non-portable preprocessor features. Each compiler has its own set of pragmas, so this should be used with caution.

## 13.6 Advanced: The Paste Operator: ##

While **cpp** is a text editor, it does recognize and separate *tokens* of the C language, such as keywords, operators, variables, etc.

The *paste operator*, `##`, allows us to construct a C token, such as a variable name, from multiple parts.

```
#define GAME(game_num) game_##game_num

int     main()

{
    // Expands to printf("Game 1 attendance = %d\n", game_1);
    printf("Game 1 attendance = %d\n", GAME(1));

    ...
}
```

## 13.7 Advanced: Predefined Macros

C preprocessors include many other predefined macros to provide information that might be useful for error messages, etc.

```
    __DATE__    Compile date
    __FILE__    Source filename
    __LINE__    Source line
    __TIME__    Compile time
```

```
    fprintf(stderr, "Error detected in %s, line %d\n", __FILE__, __LINE__);
```

## 13.8 Addendum: Advanced: Variadic Macros (C99)

As of the C99 standard, macros can take a variable number of arguments, much like the `printf()` and `scanf()` functions.

```
#define debug_printf(format, ...) \
    #ifdef DEBUG \
        fprintf(stderr, format, __VA_ARGS__) \
    #endif
```

# Chapter 14

# Pointers

## 14.1 Pointers: This Stuff is BIG!

Pointers are one of the features of C that allow us to vastly improve performance of the code. For example, pointers allow us to avoid moving large amounts of data around (as you will see in Section 16.4), a wasteful practice that is harder to avoid in some other high-level languages.

As stated earlier, the C language is less abstract than other high-level languages. Nowhere is this more evident than in how C handles pointers. A pointer is simply a variable that contains the *address* of an object, rather than the object itself.

In C, we treat addresses of objects, and the objects to which they point, explicitly in all cases. This helps us better understand what is happening at the hardware (machine code) level. Abstractions such as references in C++ and Java are implemented as pointers at the machine code level.

### 14.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What are two benefits of learning about C pointers?

## 14.2 Pointer Basics

A pointer is a variable that contains a memory address. Memory is an array of bytes, with addresses starting at 0. Every variable name *represents* a memory address, but most *contain* an object such as integers, characters, or floating point values. A pointer variable *contains* a memory address of a value rather than the value itself.

A memory address is an unsigned integer, essentially a subscript to the array of bytes we call "memory". However, we should never use integer variables to contain addresses. The compiler must handle pointers differently in some situations. For example, the `++` operator adds 1 to an integer variable, but adds the size of the object pointed to by a pointer variable. I.e., if a pointer variable called `num_ptr` points to a `double` object, then `++num_ptr` adds 8, not 1, to the address contained in `num_ptr`, since a double occupies 8 consecutive memory addresses. The `++` moves the pointer to the next `double` in memory.

Also note that an address may or may not be the same size as an `int` or a `long`. Assigning a pointer to an `int` could therefore result in a loss of the higher bits of the address, which would obviously be catastrophic to the program.

### 14.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the difference between a C pointer variable and other C variables?
2. What is an address?
3. How do the ++ and −− operators affect pointers and non-pointers?
4. What is the difference between a pointer and an integer?

## 14.3 Defining Pointer Variables

To define a pointer variable, simply add a '*' before the variable name. The '&' (address of) operator can be used to refer to the address of any variable in a program.

```
int     x;          // Contains an int object
int     *ptr;       // Contains the address of an int object
```

**Caution**
The '*' in a variable definition is associated with the variable, not the type.

```
int     *p1, p2;    // One pointer, one int

int     *p3, *p4;   // Two pointers
```

### 14.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Show a single variable definition that defines two `double` variables called `a` and `b`, and two pointers to `doubles` called `ptr1` and `ptr2`.

## 14.4 Using Pointers: Indirection

Table 14.1 shows a possible memory image for the following variable definitions.

```
int     x;          // Contains an int object
int     *ptr;       // Contains the address of an int object
double  y;

x = 5;
ptr = &x;           // ptr now contains the address of variable x
```

The variable `ptr` is assigned the *address* (1000), not the *value* (5), of the variable `x`. After that, we can refer to the value in `x` using either `x`, or by *dereferencing* the pointer `ptr`, by placing a '*' in front of it. The expression `*ptr` in a statement means "the object ptr points to". Note that the same syntax, `*ptr` has a different meaning in a variable definition.

| Address | Name | Contents |
|---|---|---|
| 1000 | x | 5 |
| 1004 | ptr | 1000 |
| 1008 | y | ? |

Table 14.1: Memory map

```
printf("%d\n", x);
printf("%d\n", *ptr);    // Same effect as above
```

**Note**

While `x` and `*ptr` both refer to the object in `x`, accessing it as `*ptr` takes slightly longer, since the computer must first get the address from `ptr` and then get the value of `x` from that address. This is called *dereferencing*, or *indirect reference*, or simply *indirection*.

```
// RISC-V assembly language equivalent of y = x;
ld      x1, x   // Load value of x from memory into the CPU
sd      x1, y   // Store value to y

// RISC-V assembly language equivalent of y = *ptr;
ld      x1, ptr     // Load address of x from memory into the CPU
ld      x2, (x1)    // Load value pointed to by ptr into CPU
sd      x2, y       // Store value to y
```

**Caution**

Like all C variables, pointers contain garbage until they are assigned a value. Accessing an incorrect address through a pointer can result in serious data corruption, or possibly a program crash, usually caused by a *segmentation fault*, where the hardware detects an attempt to access an address without authorization. For example, trying to write to the text segment of a program (the machine instructions). We can assign the sentinel value `NULL` to a pointer to indicate that it does not contain a valid address.

```
int     *ptr = NULL;

if ( ptr == NULL )
{
}
```

**Note**

It is common to increment pointers or add an integer offset to a pointer, but this is the *only* way that pointers and integers should be mixed. Anything else is likely to result in data corruption. Manipulating pointers this way is covered in Chapter 15.

**Caution**

Some programmers might think it's clever to abbreviate their code as follows:

```
if ( ptr != NULL )  // Clear and readable

if ( !ptr )         // Unclear, just showing off
```

As NULL is defined as (void *)0 by recent C standards, the latter will work, but it is not quality code, since it makes the reader waste time verifying what the code is doing. There is no benefit to using `!ptr` in place of `ptr != NULL`. This is the same issue discussed earlier regarding integer comparisons to 0.

### 14.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Show a possible memory map for the following variables, assuming the address of `a` is 2000, an `int` is 32 bits, and an address is 64 bits.

```
    int     a = 5, *ptr = &a, b = 10;
```

2. Show how to print the value of **b** in the previous question using the pointer `ptr`.
3. Is there any cost to using pointers to access an object instead of accessing it directly from a non-pointer variable?
4. Will the following code work? What does it mean? Is there a better approach? Why?

```
    double  *ptr = NULL;

    if ( ptr )
    {
        printf("%f\n", *ptr);
    }
```

## 14.5 Pointers as Function Arguments

The `scanf()` function is an example of a function that uses pointers as arguments. Any function that needs to modify a variable in the caller, must have the address of that variable, not just a copy of its value. As another example, consider a simple swap function. It is impossible to swap two arguments when passing them by value:

```
int     main()

{
    int     x, y;

    x = 1, y = 2;
    swap(x, y);

    return EX_OK;
}


void    swap(int a, int b)

{
    int temp;

    temp = a;
    a = b;
    b = temp;
}
```

Since `a` and `b` are local variables that only have copies of **x** and `y`, `x` and `y` are unaffected by the swap.

Below is a useful swap function that can actually access the arguments `x` and `y`, since it receives their memory addresses:

```
int     main()

{
    int     x, y;
```

| Address | Name | Contents |
|---|---|---|
| 3000 | x | 1 |
| 3004 | y | 2 |
| 3008 | a | 1 |
| 3012 | b | 2 |
| 3016 | temp | ? |

Table 14.2: Variable contents at the start of swap()

| Address | Name | Contents |
|---|---|---|
| 3000 | x | 1 |
| 3004 | y | 2 |
| 3008 | a | 2 |
| 3012 | b | 1 |
| 3016 | temp | 1 |

Table 14.3: Variable contents at the end of swap()

```
    x = 1, y = 2;
    swap(&x, &y);

    return EX_OK;
}


void    swap(int *a, int *b)

{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

| Address | Name | Contents |
|---|---|---|
| 3000 | x | 1 |
| 3004 | y | 2 |
| 3008 | a | 3000 |
| 3016 | b | 3004 |
| 3024 | temp | ? |

Table 14.4: Variable contents at the start of swap()

Pointer argument variables are often defined as `const` in order to prevent program bugs from wreaking havoc at run time. There is a simple trick to understanding the use of `const` with pointers.

```
int     func(const char *str)

{
}
```

Read backward, saying "pointer" for '*': "str is a pointer to a char constant". We can change the address stored in `str`, but we cannot change the character at that address.

| Address | Name | Contents |
|---|---|---|
| 3000 | x | 2 |
| 3004 | y | 1 |
| 3008 | a | 3000 |
| 3016 | b | 3004 |
| 3024 | temp | 1 |

Table 14.5: Variable contents at the end of swap()

```
int     func(char const *str)

{
}
```

Read backward, saying "pointer" for '*': "str is a pointer to a constant char", which is the same thing as above.

```
int     func(char * const str)

{
}
```

"str is a constant pointer to a char", which is different. Here, the pointer (address) is constant, not the char to which it points. We cannot change the address stored in `str`, but we can change the character at that address.

### 14.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Write a C function that prompts the user for and returns the coefficients A, B, and C of a quadratic equation $Ax^2 + Bx + C = 0$.

## 14.6 Typedefs and Pointers

We can create a new pointer type using typedef:

```
typedef int * int_ptr_t;

int     main()

{
    // All the variable below are pointers to ints
    int         *p1, *p2;
    int_ptr_t   p3, p4;

    ...
}
```

Such a simple type definition has no major pros or cons. Most programmers don't bother defining such simple types, and prefer to see a '*' in the variable definition.

**Caution**
Using `#define` to create types is also possible, but discouraged. If we attempt such a thing for a pointer type, it will lead to problems:

```
#define int_ptr_t   int *

int     main()

{
    int_ptr_t   p3, p4;     // Expands to int   *p3, p4;

    ...
}
```

## 14.7 Addendum: C99: The `restrict` Pointer Modifier

The `restrict` modifier is a hint to the compiler that allows some additional optimizations.

```
int     function(int * restrict p)

{
    ...
}
```

This tells the compiler that no other pointer will be used to access the object pointed to by `p`. When multiple pointers do not point to the same object, the compiler can generate simpler machine code. https://en.wikipedia.org/wiki/Restrict.

**Caution** It is the programmer's responsibility to ensure that the code behaves according to the assumptions of `restrict`. Behavior is undefined otherwise. If you do not fully understand what `restrict` means, don't use it.

# Chapter 15

# Arrays and Strings

Using arrays, lists, vectors, or any other aggregate objects to inhale large amounts of data into memory, should be avoided as much as possible. Arrays are often used where they are not necessary, simply because it is intuitive.

Efficiency is important in all programming, but paramount in systems programming. Unlike application programs, system code is used by *everyone*, so the cost of performance and reliability issues is very high. Also, system code should stay out of the way, leaving most hardware resources available for the applications that need them.

Consider a problem specified as "Read in a list of values and then print their square roots". If we don't stop and think first, we might follow the subliminal suggestion to design a solution that involves an array, first reading in all the values, and then computing and printing the square roots. In reality, this problem does not require storing more than one value at a time in memory. More on this in Section 15.9.

Now consider a problem where we need to read a list of numbers and print them in reverse order. This problem, along with some matrix operations such as transposition, are difficult to solve efficiently without the use of arrays. So, arrays have their uses, though we should always avoid them where feasible.

In general, using arrays or similar constructs is a good idea where the only alternative is reading from disk repeatedly. If each value in a file need only be processed once, then storing multiple values in memory at the same time provides no advantage.

## 15.1 One-dimensional Arrays

A *scalar* variable is a variable that holds a single value. It is *dimensionless*, having no length, width, etc.

An array holds multiple values, across at least one dimension. A 1-dimensional array can hold a mathematical vector, a character string, or a list. A 2-dimensional array can hold a matrix, a table, etc.

A fixed size array can be created by following a variable name with the number of elements in square brackets. The size should always be a named constant, never a hard-coded value:

```
    long    ages[MAX_AGES];
```

**Caution**
Fixed size arrays usually waste a fair amount of memory, because the data they hold can vary greatly in size, and the array must be sized to hold the largest possible list, matrix, table, etc. This is less of a problem for typical PCs and servers, which use *virtual memory*, but a critical problem for embedded devices that address RAM directly and have very little of it.

**Caution**
The default storage class for arrays defined inside a function, like all other variables, is `auto`, which means they occupy stack space. Allocating large `auto` arrays can cause stack overflows. The stack size allowed for individual processes is limited on all POSIX systems. Some systems are more generous than others. For example, FreeBSD 13 allows 512 MiB by default, while Alma Linux allows only 8 MiB.

```
# ulimit reports stack size in kibibytes (KiB)

FreeBSD moray.acadix  bacon ~ 999: ulimit -s
524288

Linux alma8.acadix  bacon ~ 1001: (pkgsrc): ulimit -s
8192
```

Hence, large fixed size arrays must be defined as `static`, so that they reside in the data segment rather than the stack segment.

```
    static long    ages[MAX_AGES];
```

Better yet, arrays can be dynamically allocated on using `malloc()`, so they reside in the heap segment and don't waste space. This is covered in Chapter 16.

Individual elements in an array can be accessed using a *subscript*, also enclosed in square brackets.

```
long    ages[MAX_AGES];
size_t  c;

for (c = 0; c < MAX_AGES; ++c)
    printf("%ld\n", ages[c]);
```

Subscripts in C begin at 0 and end at the size of the array - 1. The subscript can be any integer expression, but usually it should be of type `size_t`, an unsigned integer defined in the standard header files, with the same size as a memory address. An array is a segment of memory, and a subscript is an offset added to the *base address* of the array (the address of the first element). The negative values in the range of an `int` serve no purpose for array subscripts, which are always positive. An `unsigned int` may not have enough range for a large array, and `long` or `unsigned long` will require costly multiple precision arithmetic on some CPUs.

A memory map of the variables above might appear as follows:

```
// Assuming a 64-bit computer, so long is 64 bits, and
// MAX_AGES is defined as 5
3000    ages[0]
3008    ages[1]
3016    ages[2]
3024    ages[3]
3032    ages[4]
3040    c
```

We can see from the memory map above that the memory address of each array element is the *base address* (3000 in this case) + subscript * sizeof(long). This *address calculation* must occur for every array access, so accessing array elements is slower than accessing scalar variables.

**Note** The `sizeof()` operator in C evaluates to the size of a type or object in bytes. It is a C operator, not a library function.

**Caution** If a program accesses `ages[5]` in the array above, it will actually be accessing the memory address of the variable `c` (and misinterpreting the binary data found there if it is not of the same data type as an array element). This is called a *stray subscript* or *out of bounds subscript*. Writing to an out of bounds address is called a *buffer overflow*, and is one of the most common program bugs leading to security breaches.

**Caution**
At the machine language level, *almost all* operations on arrays use loops. Exceptions include SIMD instructions that can process a very small vector (e.g. 4 values) in parallel. Some languages have built-in features that hide these loops, such as vector capabilities:

```
% Matlab code to swap two rows of a 2D matrix:
% These look like scalar operations at the source code level, but each
% contains an implicit loop at the machine code level.  We know that
% mat[r1] is a vector, since mat is a 2D matrix, yet there is only
% one subscript.  Many other languages support similar vector
% operations that contain hidden loops.

temp = mat[r1];
mat[r1] = mat[r2];
mat[r2] = temp;
```

Even in C, which has no built-in vector capabilities, loops may be hidden inside library functions:

```
// strlen() loops through the characters looking for the null terminator
name_len = strlen(name);

// Function call to add matrices.  The function itself must have
// a nested loop to traverse the matrices.
matrix_add(sum, mat1, mat2);
```

Always be aware that operations on arrays take many times longer than operations on scalar variables.

Arrays can be initialized in the definition using a list of values enclosed in curly braces. If an initializer is provided, the array size can be omitted, since the initializer indicates how many values there are.

```
int     ages[] = { 0, 0, 0, 0, 0 };
```

### 15.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the difference between a scalar variable and an array?
2. What are two drawbacks to fixed size arrays?
3. What is a potential problem with simple arrays defined inside functions?
4. How do we get past the limits of stack size for arrays?
5. Draw a possible memory map of the following variables, assuming a 64-bit CPU, and a starting address of 1000.

```
float   vector[4] = { 5, 2, 1, 7 };
size_t  vector_len = 0;
int     c = 1;
```

6. What happens when we write to an array using an out-of-bounds subscript?
7. Are higher level languages that support vector operations faster than C, since they don't need to use loops?

## 15.2 Arrays and Pointers

In C, an array name is actually a *pointer constant*. Except for the fact that we cannot change the address to which it points, an array name is 100% interchangeable with a pointer variable.

```
    // A standard local array on the stack
    int     ages[MAX_AGES];

    // A constant pointer to an array allocated on the stack by alloca()
    // is the same as the definition above.
    // Note: Use of alloca() is risky due to stack limitations and
    // lack of stack overflow detection.  Its use is discouraged.
    int * const ages = alloca(MAX_AGES * sizeof(int));
```

We can dereference an array name just like a pointer, and we can use subscripts on a pointer variable just like an array:

```
    int     ages[MAX_AGES], *p, first_age;

    p = ages;

    // All of the following are equivalent
    first_age = ages[0];
    first_age = *ages;
    first_age = p[0];
    first_age = *p;

    // This is illegal, since ages is a pointer CONSTANT.  It always
    // refers to the base address of ages, i.e. &ages[0], and cannot be
    // altered to point to any other address.
    ++ages;
```

---

**Note**

One peculiar feature of C is that the & (address of) operator has no effect on an array name. Since the array name is already a pointer, but not a variable, the designers of C chose to ignore the & operator:

```
int     main()

{
    int     list[MAX_LIST_SIZE];

    // The "%p" format specifier is used to print a memory address
    printf("%p %p\n", list, &list);
    return EX_OK;
}
```

```
// Actual output:
0x7fffffffe750 0x7fffffffe750
```

---

### 15.2.1 Pointers Instead of Subscripts

Anything that can be done with an array subscript can also be done with a pointer variable.

Using subscripts involves computing an address at run time, i.e. base-address + subscript * sizeof(array-type).

In place of subscripts, which must be multiplied by sizeof(type) and added to the base address of the array, we can use a pointer variable that points directly to each element in the array.

**Note** Note that any time we add an integer value to a pointer, the integer is first multiplied by the size of the type pointed to. The compiler knows the size of the data type, so this multiplication is done at compile-time, not at run time.

```
    int     ages[MAX_AGES], *p, *end_ages;
    size_t  c;

    // Accessing ages[c] involves computing ages + c * sizeof(int)
    // every time
    for (c = 0; c < MAX_AGES; ++c)
        printf("%d\n", ages[c]);

    // Accessing *p does not involve an address calculation, since p
    // contains the actual address of each array element
    // end_ages contains ages + MAX_AGES * sizeof(int), and
    // ++p adds sizeof(int) to p
    // The only address calculation here is end_ages, which is
    // computed only once
    for (p = ages, end_ages = ages + MAX_AGES; p < end_ages; ++p)
        printf("%d\n", *p);
```

```
// Values of c and p in sequence, assuming the base address of ages is 2000,
// MAX_AGES is 5, and an int is 32-bits
c       p
0       2000 = 2000 + 0 * sizeof(int)
1       2004 = 2000 + 1 * sizeof(int)
2       2008   ...
3       2012
4       2016 = 2000 + 4 * sizeof(int)

end_ages = 2000 + MAX_AGES * sizeof(int) = 2000 + 5 * 4 = 2020
```

Below is an example program and run times for each loop:

```
#include <stdio.h>
#include <sysexits.h>
#include <stdlib.h>
#include <sys/time.h>
#include <xtend/time.h>

#define LIST_SIZE   1000000000

int     main(int argc,char *argv[])

{
    static unsigned short     list[LIST_SIZE], *p, *end = list + LIST_SIZE;
    size_t  c, reps;
    struct timeval  start_time, end_time;

    puts("Timing array access via pointer...");
    gettimeofday(&start_time, NULL);
    for (reps = 0; reps < 10; ++reps)
    {
        for (p = list; p < end; ++p)
            *p = p - list + 1;

        // Prevent optimizer from eliminating useless loop above
        printf("%u\r", list[random() % LIST_SIZE]);
    }
    gettimeofday(&end_time, NULL);
    printf("time = %f\n", xt_difftimeofday(&end_time, &start_time) / 1000000.0);
```

```
    puts("Timing array access via subscript...");
    gettimeofday(&start_time, NULL);
    for (reps = 0; reps < 10; ++reps)
    {
        for (c = 0; c < LIST_SIZE; ++c)
            list[c] = c + 1;

        // Prevent optimizer from eliminating useless loop above
        printf("%u\r", list[random() % LIST_SIZE]);
    }
    gettimeofday(&end_time, NULL);
    printf("time = %f\n", xt_difftimeofday(&end_time, &start_time) / 1000000.0);
    return EX_OK;
}
```

```
Timing array access via pointer...
time = 5.050704
Timing array access via subscript...
time = 6.221000
```

For the sake of understanding, we can also consider the fact that `*(ages + c)` is equivalent to `ages[c]`, since the address `ages + c` is actually computed as ages + c * sizeof(int). Likewise, `ages + c` is equivalent to `&ages[c]`. Use the more readable of the alternatives. They do the exact same thing at the machine code level.

### 15.2.2 Using Subscripts with Pointer Variables

As mentioned earlier, we can use an array subscript on a pointer variable, the same as for an array. One difference is that it might actually make sense to use a negative subscript on a pointer, provided that it does not take us out of bounds, to an address before the base address.

```
    // Shift the elements of an array
    // This is expensive and should be avoided wherever possible
    // It's only an academic example to help understand a concept

    // Using subscripts
    for (c = 1; c < MAX_AGES; ++c)
        ages[c - 1] = ages[c];

    // Using a pointer and a subscript
    for (p = ages + 1; p < end_ages; ++p)
        p[-1] = *p;     // Or, *(p - 1) = *p;

    // Using two pointers
    for (src = ages + 1, dest = ages; src < end_ages; ++src, ++dest)
        *dest = *src;
```

An alternative to shifting the contents of the array would be simply setting a pointer to ages[1]. This avoids moving large amounts of data.

```
    p = &ages[1];   // p can now be used just like ages[] would
                    // be used after shifting
```

### 15.2.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the difference between an array name and a pointer variable in C?
2. How does using a pointer to access array elements improve performance?

## 15.3 Typedefs and Arrays

```
typedef unsigned int list_t[MAX_AGES];

int     main()

{
    list_t  ages;   // Array of MAX_AGES ints

    ...
}
```

### 15.3.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Show a typedef for an array of MAX_NAME_LEN + 1 characters.

## 15.4 Advanced: More Fun with Pointers

Skip for now.

## 15.5 Arrays and Functions

Hypothetically speaking, we could say that arrays are passed by reference, since the address is being sent to the function rather than the value.

Technically, since an array name represents an address, and not an object, it is actually a pointer constant being passed by value.

When passing an array to a C function, we should also pass the array size, or the size of the list stored in the array, as a separate argument. This allows the function to accept array arguments of different sizes, and to efficiently handle the case where the array is not full.

Note that we do not need to specify a size for an array in a formal argument variable, except for multidimensional arrays, where it is necessary to know how many elements are in each row, for example.

```
int     main()

{
    int     ages[MAX_AGES];
    size_t  age_count;          // <= MAX_AGES

    age_count = read_ages(ages, MAX_AGES);

    ...

    print_ages(ages, age_count);

    return EX_OK;
```

```
}


// No size indicated for ages, since it may receive the
// address of arrays of different sizes
// Could also define ages as "int *ages", optionally using const

void    print_ages(int ages[], size_t age_count)

{
    size_t  c;

    for (c = 0; c < age_count; ++c)
        printf("%d\n", ages[c]);
}
```

---

**Caution**

A function should never return the address of an `auto` variable:

```
int     *read_ages(size_t *list_size)

{
    int     ages[MAX_AGES];
    size_t  c;

    for (c = 0; scanf("%d", &ages[c]) == 1; ++c)
        ;

    *list_size = c;

    // This returns the address of an array that will cease to exist
    // as soon as the function returns.  The stack space where this array
    // is located will be repurposed by the next function called.

    return ages;
}
```

If a function returns the address of an array or any other variable, it must be defined as `static`, or allocated with `malloc()`, which is covered in Chapter 16.

---

### 15.5.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Write a C function that reads one line of text from stdin into a character array, stopping if the array is full before a newline is read. The newline should not be included in the string. The function should return the size of the string read.

## 15.6 Lookup Tables

If a value is expensive to compute at run time, and there aren't a lot of them, then a *lookup table* might be a good option. A lookup table is simply an array filled with precomputed values.

The factorial function, N!, grows so fast that 21! is beyond the range of a 64-bit unsigned integer. This makes it an ideal candidate for a lookup table. Rather than compute factorials at run time, which requires a loop, we precompute all the factorials we can represent with `uint64_t` and store them in an array, which is a trivial amount of memory in this case.

It is crucial that this array be defined as `static`. Static variables are initialized at compile time. In contrast, `auto` arrays are initialized at run time, every time the code block is entered. Initializing an array of 21 factorials at run time is equivalent to computing 20!, the most expensive of all of them.

```
uint64_t   slowfact(unsigned int n)

{
    // This is initialized at run time, every time the function is
    // called, which actually takes longer than computing a factorial
    // for n < 21.
    uint64_t table[] = { 1ul, 1ul, 2ul, 6ul, 24ul, 120ul, 720ul,
        5040ul, 40320ul, 362880ul, 3628800ul, 39916800ul, 479001600ul,
        6227020800ul, 87178291200ul, 1307674368000ul, 20922789888000ul,
        355687428096000ul, 6402373705728000ul, 121645100408832000ul,
        2432902008176640000 };

    // No need to check for n < 0 since n is unsigned
    return n <= 20 ? table[n] : -1.0;
}
```

```
uint64_t   fastfact(unsigned int n)

{
    // This is initialized at compile time, so the values are already
    // in the array before the function is called.  The only costs to
    // this function are function call overhead and a simple address
    // calculation table + n * sizeof(uint64_t).  Much faster
    // than a loop or recursion.
    static uint64_t table[] = { 1ul, 1ul, 2ul, 6ul, 24ul, 120ul, 720ul,
        5040ul, 40320ul, 362880ul, 3628800ul, 39916800ul, 479001600ul,
        6227020800ul, 87178291200ul, 1307674368000ul, 20922789888000ul,
        355687428096000ul, 6402373705728000ul, 121645100408832000ul,
        2432902008176640000 };

    // No need to check for n < 0 since n is unsigned
    return n <= 20 ? table[n] : -1.0;
}
```

---

**Note** One might think that using `double` will increase the number of factorials we can represent. However, any factorial with more than 16 significant digits will be rounded off, which actually happens at 23!. Multiple precision integers are the only option for large and reliable factorials.

---

### 15.6.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Why do lookup tables need to be defined as `static`?
2. Write a C function that uses a lookup table to return any integer power of 10 from exponents of 0 through 9. Justify your choice of data type.

## 15.7 Pointer Arguments and const

Passing pointers, such as array names, to a function gives the function access to variables outside the function. This opens the possibility of *side effects*, unintended modification of data by a function. Imagine how annoyed you would be if the variable

`angle` were modified by the `sin()` function call below:

```
    y = sin(angle);
```

The `const` modifier, in addition to being useful for defining constants, can be used to limit side effects on arguments.

The `const` modifier can appear in any of three positions in a pointer variable definition, two of which are equivalent. The key to understanding the effect of const is reading the definition backwards, saying "pointer" for the '*':

```
// ages is a pointer to an int constant
// i.e., we can change the address, but we can't change the int
void    print_ages(const int *ages, size_t max_ages)

// ages is a pointer to a constant int (same as above)
void    print_ages(int const *ages, size_t max_ages)

// ages is a constant pointer to an int
// i.e., we cannot change the address, but we can change the int
void    print_ages(int * const ages, size_t max_ages)

// As an array, ages is a constant pointer to an int, the same as above
void    print_ages(int ages[], size_t max_ages)

// ages is a constant pointer to an int constant
// i.e., we cannot change the address or the int
void    print_ages(const int * const ages, size_t max_ages)

// ages is a constant pointer to an int constant (same as above)
void    print_ages(const int ages[], size_t max_ages)
```

```
/*
 *  Here, we cannot change what string points to, nor can we change
 *  the characters in the array.  Hence, we need a local pointer or
 *  subscript variable to traverse the string.
 */

size_t  strlen(const char string[])

{
    size_t  length;
    char    *p;

    p = string;
    while ( *p++ != '\0' )
        ++length;

    return length;
}
```

```
/*
 *  This function protects against side-effects on the string contents
 *  while allowing us to iterate without a second variable.
 */

size_t  strlen(const char *p)

{
    size_t  length;

    while ( *p++ != '\0' )
        ++length;
```

```
    return length;
}
```

Since `p` is the only variable in the function that ever addresses the array, we can define it as `restrict` to enable additional optimizations. If you don't fully understand this, don't use `restrict`, as doing so will cause undefined behavior.

```
/*
 *  This function protects against side-effects on the string contents
 *  while allowing us to iterate without a second variable.
 */

size_t  strlen(const char * restrict p)

{
    size_t  length;

    while ( *p++ != '\0' )
        ++length;

    return length;
}
```

### 15.7.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Show a pointer variable definition equivalent to the following:

   ```
   double  vector[] = { 1.0, 2.0, 3.0 };
   ```

2. Write a C function that returns the average of the values in a vector. The array may or may not be full. Make sure the function is incapable of causing side effects.

## 15.8 Multi-dimensional Arrays

Technically, C does not support multi-dimensional arrays. However, it supports arrays of arrays, which is the same thing.

```
    double  matrix[MAX_ROWS][MAX_COLS];
```

C is a *row-major* language, which means that elements of the same row (first subscript is the same) of a 2-dimensional array are contiguous in memory:

```
// A 3 x 3 matrix in memory
1000    matrix[0][0]
1008    matrix[0][1]
1016    matrix[0][2]
1024    matrix[1][0]
1032    matrix[1][1]
1040    matrix[1][2]
1048    matrix[2][0]
1056    matrix[2][1]
1064    matrix[2][2]
```

Access times to dynamic RAM (DRAM), such as *DIMMs* (dual inline memory modules) is complicated. One thing is certain, however: Accessing addresses sequentially in increasing order is faster than jumping around to non-sequential addresses. As a result, the first loop below is *much* faster than the second when large amounts of memory are used. ( If the matrix fits in cache, we won't see much difference, if any, since cache access times are constant, unlike DRAM. )

```
    // Accesses memory addresses sequentially: 1000, 1008, 1016, ...
    for (row = 0; row < rows; ++row)
        for (col = 0; col < cols; ++col)
            matrix[row][col] = some_value;

    // Jumps around, e.g. 1000, 1024, 1048, 1008, 1032, ...
    for (col = 0; col < cols; ++col)
        for (row = 0; row < rows; ++row)
            matrix[row][col] = some_value;
```

In contrast, Fortran is a column-major language.

When passing a 2-dimensional array to a function, we can omit the number of rows. The compiler need only know how many columns are in each row in order to calculate where each row begins.

```
void    print_matrix(double matrix[][MAX_COLS])

{
}
```

**Caution**
The amount of wasted address space is multiplied with each dimension in a fixed size array. E.g., a 1,000,000 element 1D array that contains only 10 elements wastes almost 1,000,000 elements. A 1,000,000 x 1,000,000 2D array that contains a 10 x 10 matrix wastes almost 1,000,000 x 1,000,000 elements.
For this reason, it is even more advantageous to use dynamic memory allocation for multi-dimensional arrays.

### 15.8.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Why don't we need the first dimension of a 2-dimensional array in a formal argument?

## 15.9 Addendum: Performance and Portability

Any program can be made to work with any amount of RAM. This does not mean that it's always practical, but it's always possible. Arrays should only be used to avoid accessing mass storage repeatedly. If data read from a file are only accessed once, the program should not be using an array, list, vector, or any other dimensioned data structure. It only pays to load large amounts of data into memory is it will be accessed repeatedly, i.e. the alternative it repeated disk access.

Increasing the memory use of programs also leads to slower memory access due to the *memory hierarchy* employed by modern computers (covered in detail in a computer architecture course). On a typical PC, computer memory actually consists of a very small amount of very fast memory (less than 1 nanosecond access time), called *level 1 cache*, a larger amount of somewhat slower memory (level 2 cache), a still larger amount of even slower memory (level 3 cache), and a very large amount of very slow memory (dozens of nanoseconds access time), called *main memory*.

On systems with *virtual memory*, programs that exceed the capacity of main memory will end up using *swap space*, a portion of disk used to extend the apparent size of memory. Disk takes between 100,000 and 1,000,000 time as long to access as main memory.

| 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 15.1: The Memory Hierarchy

The smaller the memory footprint of a program, the less often it has to use the slower memory levels. If the code and data fit entirely in level 1 cache, then memory access speed is maximized.

The more memory a program uses, the slower the memory levels it must use. If all the data fit into the level 1 cache, average memory access time is minimized.

Below is an excerpt from the output of MST-bench, a benchmark program that measures maximum sustainable throughput of RAM and disk. Note how memory throughput for a small array is more than twice as fast as for a large array on the same machine.

```
Filling a 256.00 KiB array 65536 times 8 bytes at a time...
  256.00 KiB array         8.00 B blocks       772.00 ms     21222.80 MiB/s

Filling a 2.00 GiB array 8 times 8 bytes at a time...
    2.00 GiB array         8.00 B blocks      1784.00 ms      9183.86 MiB/s
```

Consider the following Specification: Read a list of values from a file and display them on the screen. Don't be tricked by the word "list" in the specification into thinking this requires an array. Think carefully before you design and implement a solution.

```
    // A clumsy design and implementation, using an array where it isn't needed
    // This drastically increases the memory requirements of the program
    // and limits the list size to available memory.  Increasing memory
    // use also reduces cache hit ratio, so average memory access is slower.
    // Total data memory used on a 64-bit CPU: Over 4 gigabytes
    // Limit on list size: 1 billion elements

    #define MAX_LIST_SIZE   1000000000

    int     list[MAX_LIST_SIZE];    // 4 gigabytes (1 billion x 32 bits)
    size_t  c, list_size;           // 2 64-bit unsigned integers (16 bytes)

    for (c = 0; (c < MAX_LIST_SIZE) && scanf("%d", &list[c]) == 1; ++c)
        ;
    list_size = c;

    for (c = 0; c < list_size; ++c)
        printf("%d\n", list[c]);
```

```
    // A smarter design and implementation that is both simpler and minimizes
    // memory use.
    // This will likely run entirely in cache memory, so memory throughput
    // is maximized.
    // Total data memory used on a 64-bit CPU: 12 bytes.
    // Limit on list size: Disk capacity
    // Note that this design also loops through the data only once,
    // whereas the array version loops twice.

    size_t  c;      // 64-bit (8-byte unsigned integer)
```

```
    int     value;  // 32-bit signed integer

    for (c = 0; (c < MAX_LIST_SIZE) && scanf("%d", &value) == 1; ++c)
        printf("%d\n", value);
```

As another example, adding two matrices stored in files can easily be done without arrays, reading one value from each file, adding them, and immediately outputting the sum. Many people would be tempted to use 2-dimensional arrays for this problem.

Using arrays greatly increases the memory requirements for a program and also limits what the program can do based on available memory. For example, a computer with 8 GiB of memory could not add matrices of more than about 333 million elements each, using arrays of `double`, or 666 million using arrays of `float`. Using double, 333 million elements * 3 matrices (two sources and one target) = 1 billion floating point values of 8 bytes each, or 8 gigabytes of memory. The same program without arrays can handle matrices of any size and uses a trivial amount of RAM.

### 15.9.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How does the use of large arrays hurt program performance?
2. How does the unnecessary use of arrays limit the utility of programs?

# Chapter 16

# Dynamic Memory Allocation

## 16.1 Dynamic Memory Allocation: `malloc`

Fixed size arrays are rarely full, and hence usually waste memory space. Dynamic memory allocation allows us to allocate exactly the amount of memory needed, at run time.

Some people erroneously think this is no longer a concern, because modern systems use virtual memory, and hence "address space is free". In virtual memory systems, programs allocate *virtual addresses*, which are arbitrarily mapped to *physical addresses*, the actual addresses within RAM/ROM. On virtual memory systems, physical memory isn't actually allocated until the virtual address is "touched" (read or written). If you run the **top** command, you will see two memory size columns, shown as SIZE and RES below. SIZE is the amount of virtual memory allocated, and RES is the amount of resident (physical) memory actually in use.

```
  PID USERNAME    THR PRI NICE   SIZE    RES STATE    C   TIME    WCPU COMMAND
 9220 bacon        17  21    0  3604M    48M uwait    1   0:10 100.30% java
86829 root          7  20    0   322M   108M select   0   9:56   0.27% Xorg
 7114 bacon         3  20    0   258M    86M select   0   0:31   0.18% coreterm
86857 bacon         5  20    0   533M   115M select   1  15:10   0.09% lumina-d
 9221 bacon         1  20    0    14M  3356K CPU3     3   0:00   0.05% top
```

Hence, just allocating a huge array doesn't necessarily use that much physical memory. With $2^{64}$ virtual addresses available to map to a much smaller physical memory, some programmers believe that we should feel free to allocate space with reckless abandon and leave it to the virtual memory system to allocate the correct amount of physical memory.

There are two problems with this belief:

- Not all systems use virtual memory. Some or all of your code may be useful on small embedded systems, for example.
- When a system with virtual memory runs out of physical memory, it may crash or become unresponsive due to *thrashing*, excessive swapping of memory pages to and from disk. Many systems (particularly shared systems) impose limits on virtual memory allocation to protect themselves from *memory leaks* (bugs that that cause programs to allocate unlimited amounts of memory), or *malloc bombs*, malicious programs that allocate massive amounts of memory to deliberately crash a system. The latter is a type of *denial of service* attack.

Dynamic memory allocation allows a program to decide at run time how much memory to allocate, as opposed to fixed size arrays where this decision is locked at compile time.

### 16.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is the advantage of dynamic memory allocation over fixed size arrays?
2. Is address space free?

## 16.2 Basic Usage

In C, all dynamic memory allocation is performed by `malloc()`, a standard library function, which manages blocks of memory in the heap segment, rather than the stack segment used by `auto` variables.

The `free()` function is used to free memory allocated by `malloc()`.

**Caution** The corresponding `free()` call should always be the next thing you write after adding a `malloc()`. Don't wait until after writing the intervening code. Adding the `free()` immediately will prevent memory leaks.

Some languages, like Java, have no equivalent to the `free()` function. Instead, they use a *garbage collector*, a complex system designed to automatically determine when memory should be freed. Garbage collectors relieve the programmer of responsibility, but generally result in higher memory use, since they have to be conservative to ensure that memory isn't freed too soon.

Garbage collectors also introduce small, seemingly random delays into program execution when they kick in. This can be a problem for *real time* applications which must respond to external events in a fast and predictable manner.

**Caution** The `malloc()` function allocates a specified number of *bytes*, not *objects*. The `malloc()` function returns the address of the allocated memory, or `NULL` if the allocation failed. We must remember to multiply by the size of an object, which is very easy to forget, often resulting in difficult debugging ventures due to an arrays that is too small.

We may forget to multiply by the size of the data type altogether. A simple solution to this problem is a wrapper function such as `xt_malloc()`, part of the open source libxtend library.

```
    #include <xtend/mem.h>

    double  *vector;
    size_t  vector_size;

    // xt_malloc() takes the number of objects and the size of an
    // object as separate arguments, so you will get a compiler error
    // if you forget either one of them.
    if ( (vector = xt_malloc(vector_size, sizeof(*vector))) == NULL )
    {
        fputs("Could not allocate memory for vector.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    free(vector);
```

Or, just define our own macro:

```
#define SAFE_MALLOC(items, size_of_item)    malloc((items) * (size_of_item))
```

We also must be careful to get the size of an object correct:

```
// Prevent ourselves from forgetting either size of # items
#define SAFE_MALLOC(items, size) malloc(items * size)

void    some_function()

{
    double  *vector;
    size_t  vector_size;

    // Some code to determine vector_size

    // Allocate the array
    if ( (vector = SAFE_MALLOC(vector_size, sizeof(double))) == NULL )
    {
        fputs("Could not allocate memory for vector vector.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    // Code that uses the vector

    // Free allocated memory, don't leak it
    free(vector);
}
```

The above will work, but it contains a time bomb: What if we decide to change the data type of `vector`? We will need to remember to update the `double` in the `malloc()` call as well, a nuisance that could easily be forgotten. This use of `double` is like using a hard-coded constant instead of a named constant. This is an example of *fragile* code. There are simple solutions to make this code more robust:

```
// Prevent ourselves from forgetting either size of # items
#define SAFE_MALLOC(items, size) malloc(items * size)

// Solution 1: typedef
// double is now hard-coded in only one place in the program
// We do not need to make multiple changes in sync to avoid
// regressions (new bugs)
typedef double real_t;

int     main()

{
    real_t  *vector;
    size_t  vector_size;

    if ( (vector = SAFE_MALLOC(vector_size, sizeof(real_t))) == NULL )
    {
        fputs("Could not allocate memory for vector.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    free(vector);
    ...
}
```

```
// Prevent ourselves from forgetting either size of # items
#define SAFE_MALLOC(items, size) malloc(items * size)

// Solution 2: Use an object of the correct type, not the type name
int     main()

{
```

```
    double  *vector;
    size_t  vector_size;

    // The sizeof() operator can take either a type or an object
    // Use an object so the size is correct even if we change the type
    // vector refers to an address and *vector refers to a double here
    if ( (vector = SAFE_MALLOC(vector_size, sizeof(*vector))) == NULL )
    {
        fputs("Could not allocate memory for vector.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    free(vector);
    ...
}
```

### 16.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. When should we add a `free()` call to a C program?
2. Describe one pro and one con of garbage collectors?
3. How do we ensure that the size portion of a `malloc()` argument is always correct?
4. How do we ensure that we don't forget the size portion of a `malloc()` argument altogether?

## 16.3 How `malloc()` Keeps Track: Heap Tables

The `malloc()` and `free()` functions must keep track of all the blocks of the address space that are allocated or free. They maintain a table of all allocated and free blocks, called the *heap table*. Programs that perform many small allocations, and especially variable sized allocations, will create a highly fragmented heap.

The more fragmented the heap, the larger the heap table, and the more expensive `malloc()` and `free()` operations become. There are many implementations of `malloc()` libraries that attempt to maximize performance. Some developers even choose specific `malloc()` libraries for specific applications.

Linked data structures such as linked lists and trees cause such heap fragmentation. If there is an alternative implementation that uses a simple array rather than a linked structure, it will likely lead to less memory management overhead.

If we don't know the size of a list before reading it, we can make a guess and then adjust it along the way using `realloc()`, which changes the size of a block allocated by `malloc()`. Using `realloc()` is expensive in that it may have to move all the data to a new location, so it should be used as few times as possible. It does avoid heap fragmentation better than linked structures, though.

```
// Prevent ourselves from forgetting either size of # items
#define SAFE_MALLOC(items, size) malloc(items * size)

#define INITIAL_ARRAY_SIZE   1000

int     main()

{
    double  *ages;
    size_t  ages_count, ages_array_size;
```

```
    ages_array_size = INITIAL_ARRAY_SIZE;  // Guess
    ages_count = 0;
    if ( (ages = SAFE_MALLOC(ages_array_size, sizeof(*ages))) == NULL )
    {
        fputs("Could not allocate memory for ages list.\n", stderr);
        exit(EX_UNAVAILABLE);
    }

    while ( scanf("%lf", &ages[ages_count]) == 1 )
    {
        if ( ++ages_count == ages_array_size )
        {
            // Minimize the number of reallocs by increasing the
            // array size exponentially rather than incrementally
            ages_array_size *= 2;
            if ( (ages = realloc(ages, ages_array_size * sizeof(*ages))) == NULL )
            {
                fputs("realloc() failed.\n", stderr);
                exit(EX_UNAVAILABLE);
            }
        }
    }

    // Trim back the allocation to the actual list size
    ages = realloc(ages, ages_count * sizeof(*ages));

    free(ages);

    ...
}
```

### 16.3.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How do we avoid a fragmented heap table and the associated performance hit?
2. How to we minimize the expense of `realloc()` calls?

## 16.4 Pointer Arrays

A *pointer array* is an array of pointers (addresses). It is one of the most ubiquitous data structures in systems programming.

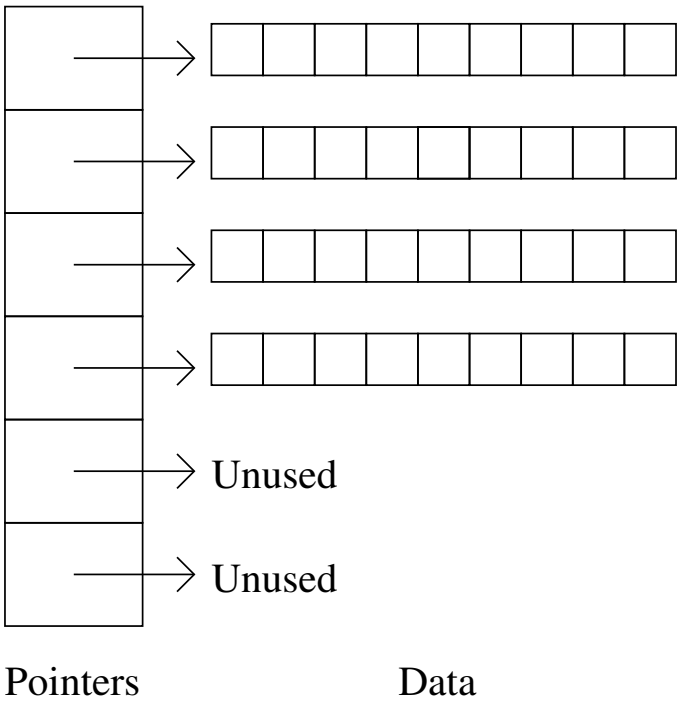



Figure 16.1: Pointer Array

Pointer arrays provide the same memory savings as a linear linked list, without sacrificing random access. Linked lists must be traversed from the beginning, while we can access any element of a pointer array instantly.

```
        Linked list:

        +-------------+     +-------------+
head -> | data | next | -> | data | next | -> NULL
        +-------------+     +-------------+
```

### 16.4.1 Pointer Arrays and Matrices

A pointer array is a convenient way to store a matrix using near-minimal memory. Some of the pointers may be unused, but this is a trivial amount of memory compared to the unused space is a fixed 2D array.

The array of pointers itself may have a fixed size:

```
    // A matrix with a fixed limit on rows, but no limit on columns
    double  *matrix[MAX_ROWS];
```

Or it may be dynamically allocated:

```
// Prevent ourselves from forgetting either size of # items
#define SAFE_MALLOC(items, size) malloc(items * size)

void    some_function()

{
    double  **matrix;

    // Allocate an array of rows pointers to doubles
    // Note that *matrix is a pointer, **matrix is a double
    if ( (matrix = SAFE_MALLOC(rows, sizeof(*matrix))) == NULL )
```

```
    {
        // Print error and exit
    }

    // Code to work with the matrix

    free(matrix);
}
```

In either case, we must allocate each row:

```
// Prevent ourselves from forgetting either size of # items
#define SAFE_MALLOC(items, size) malloc(items * size)

void    some_function()

{
    double  **matrix;

    // Allocate an array of pointers to doubles, one for each row
    // Note that *matrix is a pointer, **matrix is a double
    if ( (matrix = SAFE_MALLOC(rows, sizeof(*matrix))) == NULL )
    {
        // Print error and exit
    }

    // Allocate an array of doubles for each row (array size = # columns)
    for (row = 0; row < rows; ++row)
    {
        // Allocate an array of cols doubles (not pointers)
        if ( (matrix[row] = SAFE_MALLOC(cols, sizeof(**matrix))) == NULL )
        {
            // Print error and exit
        }

        for (col = 0; col < cols; ++col)
        {
            // Note that we can subscript a pointer array exactly
            // the same way as a 2D array
            // (e.g. double matrix[MAX_ROWS][MAX_COLS])
            // because pointers and array names are interchangeable
            if ( scanf("%lf", &matrix[row][col]) != 1 )
            {
                // Print error and exit
            }
        }
    }
}
```

To free a pointer array, we first free each array member, and then free the pointer array itself (if it was allocated).

```
void    some_function()

{
    // Code above

    // Free the matrix
    // Must free each row first, before we free the pointer to it
    for (row = 0; row < rows; ++row)
    {
        free(matrix[row]);
    }
```

```
    free(matrix);
}
```

A 10000 x 10000 fixed size array of doubles would require 800 megabytes of memory (10000 * 10000 * 8). If the matrix contained in the array is only 100 x 100, then only 80 KB are used and 799.92 MB are wasted.

The pointer array on a 64-bit computer uses 80 kilobytes (10000 * 8) bytes for the pointers in addition to the data for the pointers, so 800.01 MB total for a 10000 x 10000 matrix. For a 100 x 100 matrix, it will use 80 KB + 100 * 100 * 8 = 80.01 KB.

---

**Note**

A huge advantage of pointer arrays is that we can swap rows of the matrix (or any other pointer array) without moving any data. With a fixed size 2D array, swapping rows would require moving all the data + allocating a temporary array for one row:

```
    real_t  matrix[MAX_ROWS][MAX_COLS], temp[MAX_COLS];

    // Swap rows r1 and r2 using a loop
    // cols contains the actual number of cols in a row
    // so we don't waste time going all the way to MAX_COLS
    for (c = 0; c < cols; ++c)
    {
        temp[c] = matrix[r1][c];
        matrix[r1][c] = matrix[r2][c];
        matrix[r2][c] = temp[c];
    }

    // Swap rows r1 and r2 using memcpy(), may be faster due to
    // CPU-specific optimizations used by memcpy()
    memcpy(temp,mat[r1],cols*sizeof(real_t));
    memcpy(mat[r1],mat[r2],cols*sizeof(real_t));
    memcpy(mat[r2],temp,cols*sizeof(real_t));
```

To swap rows in a pointer array, we just swap the pointers to each row. No loops required, so this is an O(1) operation.

```
    double  **matrix, *temp;

    ...

    // Only swaps two addresses, no data, no loop
    // I.e. this is a scalar operation
    // Roughly (cols + loop_overhead) times faster than swaps above
    // E.g. if cols == 1000, this is more than 1000 times faster
    temp = mat[r1];
    mat[r1] = mat[r2];
    mat[r2] = temp;
```

---

### 16.4.2 Pointer Arrays and Strings

Arrays of strings are usually implemented as pointer arrays in C. This minimizes memory waste, especially given that strings tend to be highly variable in length.

The same memory and time savings apply as with matrices and other pointer arrays.

```
    // An array of names, + 1 to make room for the null terminator
    char    names[MAX_NAMES][MAX_NAME_LEN + 1],
            temp[MAX_NAME_LEN + 1];
    size_t  c1, c2;

    ...
```

```
    // Swap two strings
    // This is expensive, because strlcpy() uses a loop to copy
    // character-by-character
    strlcpy(temp, names[c1], MAX_NAME_LEN + 1);
    strlcpy(names[c1], names[c2], MAX_NAME_LEN + 1);
    strlcpy(names[c2], temp, MAX_NAME_LEN + 1);
```

```
    char    *names[MAX_NAMES],  // Or char **names followed by malloc()
            *temp;
    size_t  c1, c2;

    ...

    // Swapping strings in a pointer array is a scalar operation, no loops
    temp = names[c1];       // Just copying an address
    names[c1] = names[c2];
    names[c2] = temp;
```

### 16.4.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What are two advantages of pointer arrays over fixed size 2 dimensional arrays?

## 16.5 Command-line Arguments: argc and argv

The `main()` function of a C program actually receives three arguments from the caller.

```
int     main(int argc, char *argv[], char *envp[])

{
    ...
}
```

We can define **main()** with no arguments, the first two, or all three. The first two provide access to command-line arguments from the shell, or may be passed by a non-shell program in different ways. The third provides access to the environment inherited from the parent process and is covered in the next section.

Consider the command **cc -Wall -O prog.c**.

As a C program, **cc** receives the arguments as follows:

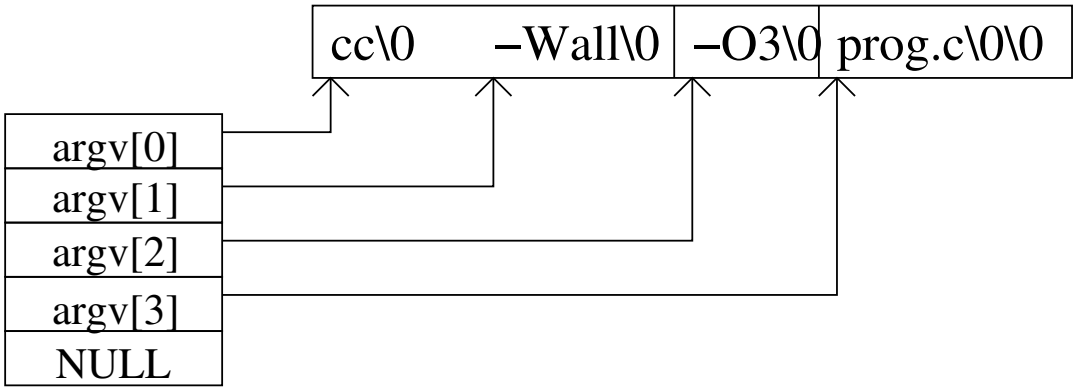



Figure 16.2: Arguments to cc

The value of `argc` is the number of command-line items, including the program name. The first element of the pointer array `argv`, **argv[0]**, is the name of the command as it was invoked. Some Unix commands have multiple names. For example, **fgrep** and **egrep** are links to **grep**. Running **fgrep** is the same as running **grep -F** and **egrep** is the same as **grep -E**.

We can easily process command-line arguments in a C program:

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[])

{
    size_t  c;

    printf("argc = %d\n", argc);
    for (c = 0; c < argc; ++c)
        printf("argv[%zu] = %s\n", c, argv[c]);

    return EX_OK;
}
```

```
shell-prompt: cc -O -Wall print-args.c -o print-args

shell-prompt: ./print-args Hi, Bob.
argc = 3
argv[0] = /home/bacon/print-args
argv[1] = Hi,
argv[2] = Bob.
```

### 16.5.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is `argc`?
2. What is `argv[0]` and how is it used?

## 16.6 The Environment: envp

The `envp` pointer array is structured the same way as `argv`, but each array element points to a string of the form "name=value", e.g. "TERM=xterm".

We can use `envp` to implement a simplified version of the standard Unix **printenv** command:

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[], char *envp[])

{
    size_t  c;

    for (c = 0; envp[c] != NULL; ++c)
        puts(envp[c]);

    return EX_OK;
}
```

```
#include <stdio.h>
#include <sysexits.h>

int     main(int argc,char *argv[], char *envp[])

{
    char    **p;    // A pointer to one of the pointers in the envp array

    for (p = envp; *p != NULL; ++p)
        puts(*p);

    return EX_OK;
}
```

```
shell-prompt: cc -O -Wall printenv.c -o printenv
shell-prompt: ./printenv
BLOCKSIZE=K
COLORFGBG=15;0
COLORTERM=xterm-256color
DESKTOP_SESSION=Lumina
DISPLAY=:0
HOME=/home/bacon
LANG=C.UTF-8
...
```

### 16.6.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is an alternative to using `envp` in a C program?

# Chapter 17

# Advanced: Function Pointers

Skip for now. Will come back to it if time permits.

## 17.1 Simple Function Pointers

## 17.2 Function Pointer Tables

# Chapter 18

# Structures and Unions

## 18.1 Structures

### 18.1.1 Structure Templates

Arrays bundle multiple objects of the same type. Structures bundle multiple objects of different types.

Structures are the ancestors of classes in object-oriented languages. A structure is like a class that has only data members, no function/method members. Structures can be easily used to implement object-oriented designs nonetheless.

#### Structure Templates

*Structure templates* define the members of a structure type. Below is a typical structure template embedded in a `typedef`.

```
// Enumerated types are integers with a limited set of named values
// The first has a value of 0 by default
typedef enum { UNDEFINED, RED, BLUE, WHITE, BLACK, CHARCOAL } color_t;
typedef enum { UNDEFINED, GAS, DIESEL, ELECTRIC } motor_t;
typedef enum { UNDEFINED, AUTOMATIC, MANUAL, NONE } transmission_t;

// Structure templates are usually associated with typedefs
typedef struct
{
    char            make[MAKE_MAX_CHAR + 1];
    char            model[MODEL_MAX_CHARS + 1];
    color_t         color;
    motor_t         motor;
    transmission_t  transmission;
    double          sticker_price;
    unsigned        days_on_lot;
}   car_t;

int     main()

{
    car_t   car1;

    ...

    // Note: Accessing structure members from main()
    // is not object-oriented programming.
    printf("Car #1 price = %f\n", car1.sticker_price);
```

```
    return EX_OK;
}
```

---

**Note** Addendum: To adhere to object oriented design principals, we can treat structures like classes, and designate certain functions as *member functions*. Only member functions should access structure members directly. C has no "private" modifier to enforce this. It is generally up to the programmer to use self-discipline.

---

The structure binding operator, `.`, has the highest precedence of any operator, so the following are equivalent:

```
    // These are equivalent
    ++car1.days_on_lot;     // Increments days_on_lot
    ++(car1.days_on_lot);
```

```
    // These are all equivalent, referring to the first character of model
    first_initial = car1.model[0];
    first_initial = (car1.model)[0];
    first_initial = *car1.model;        // Dereferences model, not car1
    first_initial = *(car1.model);

    // This one is different: It dereferences car1, not model
    // This implies that car1 is a pointer, not a structure
    // The () override the precedence of . over *
    model = (*car1).model;
```

### 18.1.2 K & R Structure Tags

The original C language that preceded ANSI and ISO standards is referred to as K & R (Kernighan and Ritchie) C. It originally did not support `typedef`.

We can also define a structure "type" without a `typedef`. Such *structure tags* are still used sometimes.

```
// car is a structure "tag", not a type
// It must ALWAYS be preceded by the word "struct"
struct car
{
    char            make[MAKE_MAX_CHAR + 1];
    char            model[MODEL_MAX_CHARS + 1];
    color_t         color;
    motor_t         motor;
    transmission_t  transmission;
    double          sticker_price;
    unsigned        days_on_lot;
};

int     main()

{
    struct car  car1, car2;

    ...
}
```

### 18.1.3 Copying Structures

Modern C standards allow entire structures to be copied in a simple assignment:

```
car_t   car1, car2;

car1 = car2;    // Copies the whole structure

// Equivalent to
memcpy(&car1, &car2, sizeof(car_t));
```

This should generally be avoided, except in rare circumstances. We should almost never duplicate aggregate objects, as this is expensive in terms of both CPU cycles and memory use. It requires a loop at the machine code level.

Also, if the structure contains any pointers, only the address in the pointer member is copied, and the new structure will point to the same object as the original. To fully duplicate a structure with pointers, it is best to write a function that duplicates any allocated objects to which the structure points.

### 18.1.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. How are structures related to classes?
2. Show a type definition for a structure containing a person's first name, last name, and age.
3. Can whole structure objects be assigned in C?

## 18.2 Pointers to Structures

Most structure access is actually done through pointers to the structures.

```
car_t   car1, *car_ptr;

car_ptr = &car1;
```

Because the structure binding operator, `.` has a higher precedence than the dereference operator, `*`, using structure pointers with `.` is untidy:

```
// This is an error, since it tries to dereference a non-pointer,
// days_on_lot, rather than car_ptr.  Furthermore, car_ptr is an
// address of a structure, not a structure, so '.' cannot be used with it.
languished = *car_ptr.days_on_lot;

// It is equivalent to
languished = *(car_ptr.days_on_lot);

// What we actually need
languished = (*car_ptr).days_on_lot;

// The above is untidy, so C provides a separate operator for
// structure POINTER binding
languished = car_ptr->days_on_lot;  // Same as (*car_ptr).days_on_lot;
```

### 18.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is an advantage to using pointers to structures rather than structure objects directly?

## 18.3 Structures, Functions, and OOP

ANSI/ISO C allows us to copy structures and also to pass structure objects to a function by value. This is inefficient and should be avoided. It is much faster to pass a pointer to a structure, since a memory address is a scalar value, while a structure object is an aggregate, which will require multiple machine instructions to copy (probably a loop), in addition to requiring more memory.

C does not provide syntactic features to encapsulate data and the functions that operate on them. This does not prevent us from using object-oriented design in C, however. We can separate member functions of a class by simply using self-discipline. Only member functions should access members of a structure used to implement a class.

---

**Note** To implement an object oriented design in C, follow one simple rule: Any given function should only access members of one structure type. This means the function is a member of that class and no others. The C compiler won't enforce this for you. It's up to you to be self-disciplined.

---

We can keep member functions in a separate source file and prefix their names with the class name. We can also use macros for simple member functions in order to separate interface from implementation without the cost of function call overhead.

```
/*
 * car.h
 */

typedef struct
{
    ...
}   car_t;

// Accessor macro
#define CAR_GET_COLOR(car)          ((car)->color)

// Mutator macro
#define CAR_SET_COLOR(car, color)   ((car)->color = (color))
```

```
/*
 * car.c
 */

#include "car.h"

// Equivalent to a constructor in C or Java
// Like car :: init() in a C++ class

void    car_init(car_t *car)

{
    car->color = UNDEFINED;
    ...
}

// Mutator function
// Like car :: set_color() in a C++ class

void    car_set_color(car_t *car, color_t color)

{
    car->color = color;
}
```

```
void    car_print(car_t *car)

{
    ...
}


int     car_read(car_t *car)

{
    int     status;

    ...

    return status;
}
```

```
/*
 *  main.c
 */

#include "car.h"

int     main()

{
    car_t   car1, car2;

    car_init(&car1);                // Like car1.init() in C++
    CAR_SET_COLOR(&car1, BLUE);
    ...
}
```

### 18.3.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What rule must we follow in order to implement a class in C?
2. Show a mutator function that sets the `age` field in the following structure.

```
typedef struct
{
    char    *first_name;
    char    *last_name;
    int     age;
}   person_t;
```

## 18.4 Nesting Structures

Structure members can be structures themselves. This enables different kinds of structures to share some common members, a simple form of *inheritance*.

```
// Features common to all kinds of vehicles
typedef struct
{
    char            make[MAKE_MAX_CHAR + 1];
    char            model[MODEL_MAX_CHARS + 1];
    color_t         color;
    motor_t         motor;
    transmission_t  transmission;
    double          sticker_price;
    unsigned        days_on_lot;
}   vehicle_t;

typedef struct
{
    vehicle_t   vehicle;

    // Only cars have spoilers and retractable headlights
    bool        spoiler;
    bool        retractable_headlights;
}   car_t;

typedef struct
{
    vehicle_t   vehicle;

    // Only trucks have bed liners and gates
    bool        bedliner;
    bool        lift_gate;
}   truck_t;


void    vehicle_set_color(vehicle_t *vehicle, color_t color)

{
    vehicle->color = color;
}


void    car_set_color(car_t *car, color_t color)

{
    // car->vehicle.color = color; would be legal in C, but would
    // not be object-oriented programming, since a car function would
    // be directly accessing a data member of the vehicle structure (class)
    // I.e. it would violate the core OOP principal of encapsulation.

    vehicle_set_color(&car->vehicle, color);
}
```

### 18.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. How can two different structures share common fields?
2. Show a structure definition for a mammal containing a Boolean field `furry` and a nested structure containing fields `average_weight` and `average_lifespan`, which can be shared with structures for reptiles, birds, fish, etc.

## 18.5 Lists of Structures

### 18.5.1 Arrays of Structures

Fixed size arrays of structure objects suffer from the same issue as other fixed size arrays: Wasted space when the array is not fully utilized (which is most of the time).

The problem is worse with structures, since they are larger objects than scalar data types. Hence, fixed arrays of structures should be avoided.

```
int     main()

{
    // Lots of memory is wasted when the list contained in the
    // cars array is smaller than MAX_CARS
    car_t   cars[MAX_CARS];

    ...
}
```

```
// Prevent outselves from forgetting either size of # items
#define SAFE_MALLOC(items, size) malloc(items * size)

int     main()

{
    // Dynamic allocation of the array avoids wasted memory
    car_t   *cars;
    size_t  actual_cars;

    ...

    if ( (cars = SAFE_MALLOC(actual_cars, sizeof(car_t))) == NULL )
    {
        // Error out
    }
    ...
}
```

### 18.5.2 Linked Lists

*Linked lists*, including linear linked lists and trees, allow us to allocate one object at a time.

Linear linked lists are time-inefficient compared to arrays, since linked lists can only be accessed sequentially. This defeats the purpose of storing data in random access memory.

When defining a type implementing a linked list with `typedef`, we have a syntactic problem:

```
typedef struct
{
    int         x;
    int         y;
    linked_t    *next;  // Oops, linked_t is not defined yet!
}   linked_t;
```

The above definition won't work, because `linked_t` is used before it is defined. This is called a *forward reference*, which is not allowed in C. This is the same reason we need prototypes for functions. To solve this, C allows us to assign a type name to a structure tag before the structure is defined.

```
typedef struct linked linked_t;

struct linked
{
    int         x;
    int         y;
    linked_t    *next;  // "struct linked *next;" will work as well
};
```

### 18.5.3 Pointer Arrays of Structures

Pointer arrays provide the same memory savings as linear linked lists, without sacrificing random access.

```
// Prevent outselves from forgetting either size of # items
#define SAFE_MALLOC(items, size) malloc(items * size)

int     main()

{
    car_t   **cars, temp_car;
    size_t  actual_cars;

    // Determine actual_cars somehow

    // First allocate an array of pointers
    if ( (cars = SAFE_MALLOC(actual_cars, sizeof(car_t *))) == NULL )
    {
        // Error out
    }

    // Assume car_read() is a member function that inputs a car structure
    for (actual_cars = 0; car_read(&temp_car) != EOF; ++actual_cars)
    {
        if ( (cars[actual_cars] = SAFE_MALLOC(1, sizeof(car_t)) == NULL )
        {
            // Error out
        }

        // Structure copy is expensive, but it's only once per
        // car during input
        cars[c] = temp;
    }
    ...
}
```

Also note that we can swap two pointers in the array much faster than we can swap two whole structure objects, as with matrices or strings.

### 18.5.4 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the problem with fixed size arrays of structures?
2. What is the down side of linear linked lists?

3. How can we get the memory savings of a linear linked list without sacrificing random access?
4. What is a major advantage of pointer arrays over arrays of structures?

## 18.6 Initializing Structures

Structure initializers look like array initializers, a list of values in curly braces.

When using structure initializers, it's a good idea to define them as a macro, so that changes only need to occur in one place and the code is less cluttered.

```
#define CAR_INIT_UNKNOWN \
    { "", "", UNDEFINED, UNDEFINED, UNDEFINED, 0.0, 0 }

#define CAR_INIT_HIGHLANDER \
    { "Toyota", "Highlander", WHITE, GAS, AUTOMATIC, 40000.0 }

#define CAR_INIT_LEAF \
    { "Nissan", "Leaf", WHITE, ELECTRIC, NONE, 30000.0 }

int     main()

{
    car_t   car1 = CAR_INIT_UNKNOWN;
    car_t   car2 = CAT_INIT_HIGHLANDER;
    car_t   car3 = CAT_INIT_LEAF;
    car_t   car4 = CAT_INIT_LEAF;

    ...
}
```

### 18.6.1 Addendum: Designated Structure Initializers (C99)

Traditional structure initializers are a bit hard to read and error-prone. We must carefully count the values and make sure they align with the appropriate structure member.

```
typedef struct
{
    int     x;
    int     y;
    int     z;
}   struct_t;

int     main()

{
    struct_t    object = { 1, 2, 3 };

    ...
}
```

Worse yet, if we rearrange the structure members and forget to rearrange all the initializers at the same time, we have introduced a *regression*, a new bug that did not exist before.

Designated structure initializers solve this problem by naming the structure members in the initializer:

```
            struct_t  object = { .x = 1, .y = 2, .z = 3 };
```

Of course, another solution is to use a constructor function:

```
void    struct_init(struct_t *object)

{
    object->x = 1;
    object->y = 2;
    object->z = 3;
}
```

### 18.6.2 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What are two advantages of using a macro to initialize a structure?
2. How does a designated structure initializer avoid bugs?

## 18.7 Advanced: Unions

A *union* looks much like a structure at the source code level, but serves a very different purpose.

Structure members coexist next to each other within a structure object.

Union members, in contrast, cannot coexist. *They occupy the same memory space.* Hence, only one member of a union can be in use at any given time. A union assigns multiple names and data types to the same memory location.

### 18.7.1 Unions for Subdividing Objects

A union can be used to access smaller components of a larger object:

```
// bigint and bytes are two names for the same memory address
typedef union
{
    uint32_t    bigint;
    char        bytes[4];
}   split_t;

int     main()

{
    split_t split;

    ...
}
```

The member `bytes[0]` refers to the first byte of the integer `bigint`. Assuming the variable `split` resides at address 1000:

```
1000    split.bigint  split.bytes[0]
1001    split.bigint  split.bytes[1]
1002    split.bigint  split.bytes[2]
1003    split.bigint  split.bytes[3]
```

The main problem here is that the order of the bytes of `bigint` is endian-dependent. On a little-endian CPU, the lowest byte of `bigint` will be at address 1000, while on a big-endian machine it will be at 1003.

If we specifically want the low byte of a larger integer, we should use a bitwise operator and a mask:

```
    char        low_byte;
    uint64_t    bigint;

    low_byte = bigint & 0x000000ff; // Get the lowest byte from bigint
    low_byte = bigint & 0x0000ff00; // Get the second lowest byte from bigint
```

### 18.7.2 Unions for Conserving Memory

Sometimes, two members of a structure are not useful at the same time.

```
typedef struct
{
    char            make[MAKE_MAX_CHAR + 1];
    char            model[MODEL_MAX_CHARS + 1];
    color_t         color;
    motor_t         motor;
    transmission_t  transmission;
    double          sticker_price;
    unsigned        days_on_lot;
    bool            lift_gate;      // Only for trucks
    bool            spoiler;        // Only for cars
}   vehicle_t;
```

```
typedef struct
{
    char            make[MAKE_MAX_CHAR + 1];
    char            model[MODEL_MAX_CHARS + 1];
    color_t         color;
    motor_t         motor;
    transmission_t  transmission;
    double          sticker_price;
    unsigned        days_on_lot;

    // lift_gate and spoiler are two names for the same memory location
    // This saves memory since only one of them can be used anyway
    union
    {
        bool            lift_gate;      // Only for trucks
        bool            spoiler;        // Only for cars
    }   vehicle_specific;
}   vehicle_t;
```

The nuisance here is that the union members require an extra tag:

```
    vehicle.color = BLUE;
    vehicle.vehicle_specific.lift_gate = true;
```

To eliminate this nuisance, C allows for *anonymous unions*, i.e. unions within a structure that have no name.

```
typedef struct
{
    char            make[MAKE_MAX_CHAR + 1];
    char            model[MODEL_MAX_CHARS + 1];
    color_t         color;
    motor_t         motor;
    transmission_t  transmission;
    double          sticker_price;
    unsigned        days_on_lot;
```

```
    // This union has no name, so lift_gate and spoiler appear to
    // be directly members of vehicle_t.
    union
    {
        bool            lift_gate;      // Only for trucks
        bool            spoiler;        // Only for cars
    };
}   vehicle_t;
```

```
    vehicle.color = BLUE;
    vehicle.lift_gate = true;
```

### 18.7.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is the difference between a structure and a union?
2. What is the problem with using a union to address individual bytes of an integer? What is the solution?

## 18.8 Advanced: Structure Alignment

---

**Note** Material from this section to the end of the chapter is optional. Instructors may or may not include it.

---

Some CPU architectures require that objects larger than a byte be aligned at certain memory addresses. E.g., a 32-bit integer might have to reside at an address that is a multiple of 4. Others may allow such objects to reside anywhere (unaligned), but access will be faster if they are aligned properly. E.g. reading a 32-bit integer at address 1002 might mean reading two 32-bit integers at 1000 and 1004 and only keeping the bytes from 1002, 1003, 1004, and 1005. Alignment requirements are very CPU-specific.

Because of this, structure members may need to be aligned either for basic functionality or for efficiency. This may involve inserting unused "padding" space between members:

```
typedef struct
{
    short   x;
    long    y;
}   some_struct_t;
```

```
1000    x
1001    x
1002    padding
1003    padding
1004    y           // y must be aligned for optimal performance
1005    y
1006    y
1007    y
```

If a program writes structures in binary form, then we must ensure that the order of the members and the padding are the same on all systems. Be aware that compilers may reorder members to reduce the need for padding and optimize alignment of members.

## 18.9 Advanced: Bit Fields

Structure members, unlike standalone variables, can occupy less than 1 byte. This may be useful for saving memory where a large array of structures is used, for example.

The total memory used by a group of bit fields is determined by the type of the member:

```
// sizeof(bits_t) = 2, since the bits fields are part of an
// unsigned short.  Only 12 of the 16 bits are actually used.
typedef struct
{
    unsigned short  a : 4;
    unsigned short  b : 3;
    unsigned short  c : 5;
}   bits_t;
```

## 18.10 Addendum: Advanced: Enforcing OOP in C, Opaque Structures

One option for OOP in C is to expose the entire structure definition to all code and simply expect discipline on the part of programmers using the class. If they directly access structure members from nonmember functions and the code breaks when we change the structure, it's their own fault.

```
// Header containing structure definition and type definition
// Included by application source files, so the structure
// members are visible (the structure is transparent)
typedef struct
{
    int     a;
    int     b;
}   class_t;

// Prototypes for member functions
// Users of the class must choose to use these API functions instead
// of accessing a and b directly from nonmember functions
void    class_init(class_t *class);
...
```

```
// Source file outside class
// struct my_class is not exposed in this source file since it
// does not include private-my-class.h.

#include <my-class.h>

int     main()

{
    class_t  object;

    object.a = 4;               // This is not OOP, but it is legal
                                // We should only access structure members
                                // directly in member functions

    class_set_a(&object, 4);    // This is the proper way for a non-member
                                // function to set a, using the mutator

    ...
}
```

As a service to users of the class, we can simulate the private data members of C++ and Java classes by making a structure *opaque* to source code outside the class while keeping it *transparent* to member functions (member functions can see inside it).

This simply means that we only expose the complete structure definition to member functions. Opaque structures can be created in C using a loophole regarding structure types. Recall from Section 18.5.2 that C allows us to use `typedef` on a structure definition with a tag, even if the structure is not yet defined.

We can use this to hide a structure definition from nonmember functions, while exposing it to member functions. We use two separate header files, one which includes API (Application Program Interface, the public function prototypes) and is included by files outside the class, and one with the private structure definition, which is only included by files defining member functions. Hence, the structure definition is opaque (invisible) to non-member functions, and they therefore cannot access the data members directly. They are forced to use the API provided by member functions, just like non-member functions in C++ or Java (that are not "friends").

This means that non-member functions cannot even define a variable of the structure type. Since the size of the structure is not known, the compiler cannot generate code to allocate the proper amount of memory. Non-member functions can, however, define pointers to the structure type, since the size of a pointer is the same for all data types.

```
/*
 *  Public API for class_t.  This header is installed with the library
 *  to expose the API to applications using the class library.
 */

// Typedef declaring structure type without exposing the structure
typedef struct class class_t;

// Constructors
class_t *class_init_0(void);
class_t *class_init_ab(int a, int b);

// Input object
int     class_read(class_t *class_ptr, FILE *stream);

// Output entire object
void    class_write(class_t *class_ptr, FILE *stream);
```

```
/*
 *  This header is only exposed to class members.  It is not installed with
 *  the library, so that the structure definition is not exposed to
 *  applications using the class and is completely opaque to them.
 *  They must use the API functions prototyped in api.h to maniulate
 *  objects of this class.
 */

struct class
{
    int     a;
    int     b;
};
```

```
/*
 *  C file containing definitions of class member functions.
 *  This file includes "class.h", a private header file containing
 *  the structure definition.  This file is not installed, so the
 *  structure is opaque to applications using this class library.
 */

#include <stdio.h>
#include <stdlib.h>
#include "class.h"
```

```c
#include "api.h"

/*
 *  Constructors
 */

class_t *class_init_0()

{
    class_t *temp = malloc(sizeof(class_t));
    temp->a = 0;
    temp->b = 0;

    return temp;
}


class_t *class_init_ab(int a, int b)

{
    class_t *temp = malloc(sizeof(class_t));
    temp->a = a;
    temp->b = b;

    return temp;
}


/*
 *  Input entire object
 */

int     class_read(class_t *class_ptr, FILE *stream)

{
    return fscanf(stream, "%d %d", &class_ptr->a, &class_ptr->b);
}


/*
 *  Output entire object
 */

void    class_write(class_t *class_ptr, FILE *stream)

{
    fprintf(stream, "a = %d b = %d\n", class_ptr->a, class_ptr->b);
}
```

```c
/*
 *  Test program for the class library.  Note that it does not include
 *  class.h, the private header file containing the structure definition.
 *  It only uses api.h, which exposes the class API.
 */

#include <stdio.h>
#include <sysexits.h>
#include <stdlib.h>
#include "api.h"

int     main(int argc,char *argv[])
```

```
{
    class_t *object = class_init_ab(1, 2);

    class_write(object, stdout);
    fputs("Enter integers a and b: ", stdout);
    class_read(object, stdin);
    class_write(object, stdout);

    return EX_OK;
}
```

```
##############################################################################
#   Makefile to build and install the class library.
#   Also builds a test program, oopc-test.
##############################################################################

##############################################################################
# Installed targets

BIN     = oopc-test
LIB     = libclass.a

##############################################################################
# List object files that comprise BIN and LIB.

BIN_OBJS    = oopc-test.o
LIB_OBJS    = class.o

##############################################################################
# Compile, link, and install options

PREFIX      ?= ../local

# Defaults that should work with GCC and Clang.
CC          ?= cc
CFLAGS      ?= -Wall -g -O
LD          = ${CC}
LDFLAGS     += -L. -lclass

AR          ?= ar
RANLIB      ?= ranlib

MKDIR       ?= mkdir
RM          ?= rm
INSTALL     ?= install

##############################################################################
# Standard targets required by package managers

.PHONY: all depend clean install

all:    ${BIN} ${LIB}

${BIN}: ${BIN_OBJS} ${LIB}
        ${LD} -o ${BIN} ${BIN_OBJS} ${LDFLAGS}

${LIB}: ${LIB_OBJS}
        ${AR} r ${LIB} ${LIB_OBJS}
        ${RANLIB} ${LIB}
```

```
class.o: class.c class.h api.h
        ${CC} -c ${CFLAGS} class.c

oopc-test.o: oopc-test.c api.h
        ${CC} -c ${CFLAGS} oopc-test.c

###############################################################################
# Remove generated files (objs and nroff output from man pages)

clean:
        rm -f *.o ${BIN} ${LIB}

###############################################################################
# Install all target files (binaries, libraries, docs, etc.)
# Do not install class.h, which is not meant to be exposed to applications.

install: all
        ${MKDIR} -p ${DESTDIR}${PREFIX}/include ${DESTDIR}${PREFIX}/lib
        ${INSTALL} -s -m 0755 ${BIN} ${DESTDIR}${PREFIX}/bin
        ${INSTALL} -m 0644 api.h ${DESTDIR}${PREFIX}/include
        ${INSTALL} -m 0644 ${LIB} ${DESTDIR}${PREFIX}/lib

test: ${BIN}
        ./oopc-test
```

Note that this precludes the use of macros as accessor functions, since a macro would expand to code that directly accesses the structure members, which are not visible to the code using the macro. Functions can be used and should serve the same purpose, though with a little added overhead. C function call overhead is very low.

## 18.11 Addendum: Advanced: Flexible Array Members (C99)

Skip for now, not often useful.

## 18.12 Addendum: Advanced: void pointers

Sometimes we may want to pass different kinds of structures to a function, depending on the data. This is useful for things like *plugins*, functions that are compiled separately from the program and linked conditionally at runtime using `dlopen()`. Plugins may be very different from each other, but nevertheless must have the same function interface.

An argument of type `void *` accepts a pointer to any data type. This allows the two functions below to receive different kinds of data through a common interface. The `data` argument can point to literally anything, a character string, an integer, any structure type, etc.

```
int     plugin1(int fd, unsigned mask, void *data);
int     plugin2(int fd, unsigned mask, void *data);
```

Many standard library functions involved in systems programming use void pointers.

# Chapter 19

# Debugging

This chapter will be covered very quickly. Students need only have awareness of debuggers like **gdb** and **lldb** and their basic use.

## 19.1 Thinkin' it Through

There are a few different ways to track down bugs in a program, including reading the code carefully, adding debug statements, and using a *debugger*, a tool specifically designed to aid in debugging other programs.

**Caution** Don't become dependent on debuggers. Avoid creating bugs in the first place. Catch bugs early by testing after every few lines of new code are added. Learn to *read your code* and understand what it's doing. This skill is far more valuable than knowing how to use a debugger. Cliche', but true: An ounce of prevention is worth a pound of cure.

However, even if you program intelligently 100% of the time, you will sometimes have to debug code that others wrote, or that you wrote long ago, before you were so wise. So, debuggers will be indispensable at times.

**Note** If a bug occurs and the problem is not obvious, first focus on determining *where* the problem is. This is usually quite easy, and once you determine where the faulty code is, it's usually not hard to see *what* the problem is and how to correct it.

### 19.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. How do we avoid the need for a debugger?
2. What is the first step in correcting a bug?

## 19.2 Making Programs Talk: Debug Code

Caveman debugging was introduced in Section 7.5. It involves adding print statements to your code that display intermediate results, so you can follow the progress of the program on the screen. As stated earlier, these messages should go to `stderr`, rather than `stdout`.

The `stdout` stream is buffered by default, so if your program crashes, some of the debug output may die in an output buffer, never reaching the screen. This will mislead you to believe that the program never made it to that debug statement.

Debug output needs to be disabled when it is no longer needed. We may want to simply remove the statements, or we may want to leave them intact in case we need them again. The latter can be done in 3 ways:

- Comment it out.

```
    // fprintf(stderr, "Done with loop.  sum = %d\n", sum);
```

- Mask (guard) it with an if statement. This has a small cost at run time, but allows debugging to be enabled or disabled without recompiling the program. Many programs accept a flag argument such as `--debug` to allow the user to easily enable debugging statements. Often, multiple different debugging levels are supported to make the debug output more or less verbose.

```
/*
 *  An exception to the no global variables rule, since it is used
 *  more like a constant.  It is set only at program startup, and
 *  read-only after that, so functions cannot cause side effects
 *  on each other.  Capitalized to make it clear that it is global.
 *  (I use all lower case for locals, capitalize globals, and use all
 *  upper case for macros.)
 */

bool    Debug = 0;

void    some_function()

{
    // Higher debugging level
    if ( Debug > 1 )
        fputs("Entered some_function().\n", stderr);
    ...

    // Lowest debugging level
    if ( Debug > 0 )
        fprintf(stderr, "Done with loop.  sum = %d\n", sum);
    ...
}
```

- Mask (guard) it with a preprocessor directive. The preprocessor completely removes the debug code before compilation, so there is no run time overhead. However, we must recompile the program to toggle debugging.

```
#if DEBUG > 0
    // This statement is not output by the preprocessor if DEBUG is 0
    fprintf(stderr, "Done with loop.  sum = %d\n", sum);
#endif
```

```
shell-prompt: cc -O -Wall -DEBUG=1 prog.c -o prog
```

### 19.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is caveman debugging? Is it obsolete?
2. Where should debug output be sent and why?
3. How do we disable debug output when we no longer need it?

## 19.3 Unix Debuggers: Which One?

There are many debuggers on the market, some of which have sophisticated graphical interfaces. We will introduce three tools that are free and portable to most POSIX platforms.

### 19.3.1 Debugger Features

Debuggers allow you to examine variable contents, making them an alternative to caveman debug code.

With a debugger, we can set *break points*, where the program is automatically stopped. This way, we can see how far the program is going before it crashes.

When a program terminates, we can use a debugger to get a *traceback* (or backtrace, or just trace), showing the exact point where it terminated, along with the function calls and argument values leading up to it.

Getting an accurate trace requires compiling with `-g` and *not* using aggressive optimizations. The `-g` flag tells the compiler to generate an address map, connecting the location of each machine instruction to a source statement. This map allows the debugger to tell you where in the source code the problem occurred. Remember that we are debugging machine code, not source code. Some aggressive optimizations make it impossible to generate an accurate map. Without `-g`, the debugger can trace the function calls, but cannot pinpoint a location in the source code.

---

**Note** Changing the optimization level may cause or eliminate a crash. The optimizer may alter the organization of data in memory, thus changing the impact of a stray pointer. This is another reason to test code incrementally to catch bugs before they become difficult.

---

### 19.3.2 Types of Debuggers

*Symbolic debuggers* use the *symbol table* in the executable to track *external* symbols (static variables and functions). They cannot always track local variables, whose address may be different each time a block of code is executed.

*Source level debuggers* use the complete address map generated by `-g` to determine the exact source statement associated with each machine instruction.

### 19.3.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How do we prepare a program for optimal debugging?

## 19.4 The GNU Debugger: `gdb`

### 19.4.1 Running Programs Under `gdb`

```
shell-prompt: cc -O -Wall -g prog.c -o prog
shell-prompt: gdb ./prog
(gdb) run [arguments]
Segmentation fault, core dumped.
(gdb) bt
...Shows backtrace
```

### 19.4.2 Using a Core File

A *core file* is a snapshot of a program in memory at the moment of termination. It allows a debugger to see the values of variables and other items on the stack to determine exactly where the program was at the moment of termination.

```
shell-prompt: cc -O -Wall -g prog.c -o prog
shell-prompt: ./prog
Segmentation fault, core dumped.
shell-prompt: gdb ./prog prog.core
(gdb) bt
...Shows backtrace
```

We can force a program to terminate and generate a core file using the `SIGABRT` signal.

```
shell-prompt: ps    # Or "top", determine the PID of your program
shell-prompt: kill -ABRT 8923
```

### 19.4.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Show how to force a process to terminate and generate a core file, then show a backtrace using **gdb**.

## 19.5 Addendum: The LLVM Debugger: `lldb`

### 19.5.1 Running Programs Under `lldb`

```
shell-prompt: cc -O -Wall -g prog.c -o prog
shell-prompt: lldb ./prog
(lldb) run [arguments]
Segmentation fault, core dumped.
(lldb) bt
...Shows backtrace
```

### 19.5.2 Using a Core File

```
shell-prompt: cc -O -Wall -g prog.c -o prog
shell-prompt: ./prog
Segmentation fault, core dumped.
shell-prompt: lldb -c prog.core [./prog]    # ./prog is optional
(lldb) bt
...Shows backtrace
```

### 19.5.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Show how to force a process to terminate and generate a core file, then show a backtrace using **lldb**.

## 19.6 Addendum: Valgrind

Valgrind is a tool used mainly for memory debugging (e.g. detecting memory leaks) and *profiling*, determining where a program spends its time.

Valgrind operates by decompiling machine code to its own intermediate format, analogous to Java byte code, and then running it under its own interpreter and JIT compiler.

# Part III

# Unix Library Functions and Their Use

As we have seen, C is a very simple language. Most of the syntax of the language was covered in the previous part of this text. If you've been doing your homework, you now know most of what there is to know about the C language itself.

However, most of the functionality of C programs comes not from language features, but from *library functions*, functions written in C, precompiled, and stored in archives called *libraries* from which the linker can extract them.

The standard C libraries are a *vast* collection of many thousands of functions that we can use to build sophisticated programs. They are like an unlimited supply of free Lego bricks for C programmers. This part of the text introduces a very tiny fraction of common library functions just to provide a rough idea of what is possible.

Furthermore, the functionality available in free, open source 3rd party libraries dwarfs the standard libraries. There are generic libraries like libxtend (https://github.com/outpaddling/libxtend), scientific libraries like biolibc (https://github.com/auerlab/biolibc), hashing libraries like xxHash (https://github.com/Cyan4973/xxHash), and other libraries for just about any purpose you can imagine.

Due to the vastness of the standard libraries, we are now going to shift our focus from the depth of understanding that we stressed while covering the C language, to a more breadth-based approach which aims only to provide familiarity with the nature of the standard libraries and the concepts associated with some of the key features.

From a college educational perspective, the standard libraries contain roughly 1,000 credits worth of material. You can spend your entire career working as a Unix systems programmer and only learn a small fraction of all there is to know about the standard libraries. Hence, a 3-credit course can only provide the most basic introduction.

This is a prime example of the fact that education is not so much about accumulating knowledge as it is about learning how to find it quickly and independently. We cannot succeed as programmers by memorizing facts about languages and libraries. We must be skilled at finding what we need when we need it, e.g. by checking man pages to find out which headers must be included and what link options are required for a given library function. We need to learn to use various resources to find out what functions exist that might serve our purposes. The SEE ALSO sections in the man pages and web searches are often useful. When reading books about programming, don't try to memorize details that can easily be looked up in the man pages. Recognize the important concepts and retain them, and focus on understanding them.

Our goal from this point on is simply to develop a sense of what the libraries have to offer, and the skills to look up the information we need quickly. For standard library functions, 99% of that information is a few seconds away in the man pages.

# Chapter 20

# Building Object Code Libraries

## 20.1 Creating Libraries

Libraries, as we now know, are collections of precompiled functions, such as `printf()`, `malloc()`, etc. that the linker can add to our executable when we build a program. A library is similar to a **tar** archive, in that it is a single file containing multiple files.

As discussed earlier, we tell the linker to use a library with the `-l` flag followed by the unique portion of the library filename. E.g., to extract code from `libm.a`, we use `-lm`. To extract code from `libxtend.a`, we use `-lxtend`. We use `-Ldirectory` to tell the linker to look in `directory` for additional library files.

For every new function you write, there are two choices:

- Add it to the program
- Add it to a library

If a function *might* be useful to another program, then adding it to a library makes it readily available and saves us the work of duplicating or moving it later.

Building and maintaining libraries is as easy as building and maintaining programs, as you will see in the coming sections.

---

**Note**
By default, most linkers extract an entire object file from a library, not individual functions. Hence, if you place multiple functions in a source file used to build a library, *all* of the functions in that source file will be linked into programs, even if only one of them is used. Sometimes two or more functions must always coexist, so placing them in the same source file makes sense. Other times, we may want a source file to contain only one library function.
Some compilers and linkers support extracting individual functions, but we should not assume this is possible. The portable approach is to separate functions that do not need each other into their own source files.

---

Specifying a single library in a link command may link in hundreds or even thousands of object files. The benefit of using a library instead of listing all these object files in the link command should be obvious.

### 20.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Which new functions should be placed in a library rather than made part of a specific application?

## 20.2 Static or Dynamic?

When using static libraries, the linker extracts object (.o) files from the library and adds them to your executable, just as if the object file were specified directly in the link command.

With *dynamic*, A.K.A. *shared* libraries, only a reference to the library is added to the executable. This results in potentially much smaller executables. The *loader*, which reads an executable file from disk and loads the machine code into memory, finds the shared libraries and extracts machine code from them at run time.

If shared library code was previously loaded for one application, then other applications can use the image already in memory. Hence, dynamic libraries also save memory *if* multiple processes are using the same shared library code at the same time. Good use cases include GUI libraries used by many desktop applications, such as GTK and Qt. GUIs contain a lot of code in order to handle all the details of a graphics interface (fonts, colors, etc) and are shared by many applications regardless of their domain (business, science, education, etc.).

Dynamic libraries add a lot of complexity, since applications must have access to the same library version used when the application was compiled, or at least one with an identical ABI. If we upgrade a shared library, we must keep the older versions around or recompile *all* programs that use the library. If the API (function interfaces) of the library has changed from the previous version, then client programs can either be modified or continue using the old version.

### 20.2.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What happens when we link to a static library?
2. What happens when we link to a dynamic (shared) library?
3. What is the advantage of static libraries?
4. What are the advantages of a dynamic library?

## 20.3 Creating Static Libraries

Static libraries are easy to create and are well standardized. Static library files have a ".a" extension. Building a static library entails two differences from building an executable:

- None of the source files can have a `main()` function.
- The link stage is replaced with an **ar** command, which creates a static library archive from the object files instead of an executable. We usually also run **ranlib** on the archive to build an index used by the linker to search the library.

Suppose we want to build our own math library using two source files containing math functions, `fastfact.c` and `intpower.c`. The Makefile below shows how we would do this.

---

**Note** Addendum: The book explicitly uses **gcc** in the example, which is not portable. At the time the book was written, it was common to install **gcc** on commercial Unix systems and use it instead of the native **cc** compiler. This is no longer common.

---

```
OBJS    = fastfact.o intpower.o
LIB     = libmymath.a
API     = mymath.h

CC      ?= cc
```

```
CFLAGS  ?= -Wall -O

AR      ?= ar
RANLIB  ?= ranlib
MKDIR   ?= mkdir
INSTALL ?= install
RM      ?= rm

#####################################################################
# This is the only major difference from building an executable

${LIB}: ${OBJS}
        ${AR} r ${LIB} ${OBJS}
        ${RANLIB} ${LIB}

#############################################################
# Building the object files is the same as for an executable

fastfact.o: fastfact.c mymath.h
        ${CC} -c ${CFLAGS} fastfact.c

intpower.o: intpower.c mymath.h
        ${CC} -c ${CFLAGS} intpower.c

.PHONY: install clean

install:
        ${MKDIR} ${DESTDIR}${PREFIX}/lib ${DESTDIR}${PREFIX}/include
        ${INSTALL} -c ${LIB} ${DESTDIR}${PREFIX}/lib
        ${INSTALL} -c ${API} ${DESTDIR}${PREFIX}/include

clean:
        ${RM} -f *.o ${LIB}
```

Output should appear as follows, assuming no object files exist:

```
cc -c -Wall -O fastfact.c
cc -c -Wall -O intpower.c
ar r libmymath.a fastfact.o intpower.o
ranlib libmymath.a
```

### 20.3.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Show the commands needed to create a static library `libstring.a` from `strcmp.o`, `strlen.o`, `strlcat.o`, and `strchr.o`.

## 20.4 Creating Dynamic (Shared) Libraries

Dynamic libraries are about as easy to create as static libraries, but they are not well standardized. For example, BSD and Linux systems use an extension of ".so" (short for shared object), while macOS uses ".dylib" (dynamic library). The tools for manipulating and examining shared libraries also differ across platforms. Dynamic libraries also must be tagged with a specific version, so that multiple versions of the same libraries can coexist and be distinguished by the linker. For the sake of simplicity, we will focus on static libraries in this class.

### 20.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the major challenge in creating dynamic (shared) libraries?

# Chapter 21

# Files and File Streams

Coverage of this topic will be brief and limited to key concepts and awareness of available functions. Most of the detailed information you need can be found in the man pages, e.g. **man fopen()**. Do not waste time memorizing the syntax of C library functions.

Students are not expected to memorize details about library functions. They should be aware of the common functions and their basic use, and how to find more information in the man pages.

## 21.1 FILE Streams

The concept of FILE streams was introduced in Chapter 7. A stream is a buffering mechanism built on top of the low-level `read()` and `write()` block I/O functions. Streams allow us the illusion of reading and writing one or a few bytes at a time, without the enormous overhead we would incur if we actually accessed disk or other devices every time we need a single character.

It can take up to a few milliseconds to locate a piece of data on a disk due to *seek time* (the time it takes to move the read/write heads to the proper cylinder, one of the concentric circles of data on a disk) and *latency time* (the time it takes to wait for the disk to rotate so the data pass under the heads). If we actually read one byte at a time from random locations on the disk, we would only get a few hundred bytes per second.

FILE streams read a large block of data into a buffer (character array) each time they read from disk. If our program reads only 1 character at a time from the file, it will get most of them from the buffer, and only suffer seek and latency times once per block instead of for every character. If a program reads large blocks of data, then FILE streams are not helpful and may actually hurt performance.

### 21.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How do FILE streams improve I/O performance?

## 21.2 The `FILE` Structure

Streams are implemented by a structure similar to the following:

```
typedef struct
{
    char    *buff;      /* I/O buffer */
    char    *p;         /* Next available character */
    size_t  bytes;      /* # of characters present */
    size_t  buffsize;   /* Size of the buffer */
    short   flags;      /* Stream parameters */
    short   file;       /* File descriptor buffered by this structure */
}   FILE;
```

The `buff` field is a character array allocated with `malloc()`.

When reading a file, `buff` is filled using the low-level `read()` function to read a block of data, and `p` is set to the beginning of `buff`. Then functions and macros such as `getc()` take one character at a time from `buff` and update `p`. When `p` reaches the end of the buffer, another block is read using `read()` to refill `buff`.

When writing a file, functions and macros such as `putc()` place characters into the buffer one at a time and update `p`. When the buffer is full, the entire contents are written with a single `write()` and the buffer is marked empty (`p` is set back to the beginning of `buff`).

### 21.2.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What happens the first time a program calls `getc()` on a new stream?
2. What happens the second time a program calls `getc()` on a new stream?
3. How often does `getc()` actually access an input device?

## 21.3 Basic Stream I/O Functions

### 21.3.1 Opening a File: `fopen()`

The `fopen()` function opens a file, and allocates and initializes a new `FILE` structure. It returns a pointer to the `FILE` structure if successful, or `NULL` if the file cannot be opened.

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

The common modes are:

- "r" for read-only
- "r+" for read+write on an existing file
- "w" for write-only (the file is overwritten if it exists or created if it does not)
- "w+" for overwrite and also allow reading
- "a" for append (add to an existing file)
- "a+" for append + allow reading

### 21.3.2 Closing a File: `fclose()`

The `fclose()` function closes a file and frees all memory associated with the `FILE` structure.

**Caution** An `fclose()` should be inserted into the program immediately after adding the corresponding `fopen()`, to ensure that we don't forget to close the file. Failing to close a file can result in lost data.

```
#include <stdio.h>
#include <sysexits.h>
#include <errno.h>      // Defines external status variable errno
#include <string.h>     // strerror()

void    some_function()

{
    char    *filename;
    FILE    *infile;

    ...

    // strerror(errno) returns a human readable message such as
    // "Permission denied" or "File does not exist".  This should always
    // be used when an error occurs opening a file.

    if ( (infile = fopen(filename, "r")) == NULL )
    {
        fprintf(stderr, "Cannot open %s: %s\n", filename, strerror(errno));
        exit(EX_NOINPUT);
    }

    ...

    fclose(infile);
}


void    some_other_function()

{
    char    *filename;
    FILE    *outfile;

    ...

    if ( (outfile = fopen(filename, "w")) == NULL )
    {
        fprintf(stderr, "Cannot open %s: %s\n", filename, strerror(errno));
        exit(EX_CANTCREAT);
    }

    ...

    fclose(outfile);
}
```

### 21.3.3 Reading Characters: `getc()` and `fgetc()`

The `getc()` macro and corresponding `fgetc()` function read a single character from a `FILE` stream. If the stream buffer is empty, they call `read()` to attempt to fill it.

They return the character read (as an `int`), or EOF if the end of the file was encountered or an error occurs. The `ferror()` or `feof()` functions can be used to distinguish between end-of-file and an error condition.

The macro is generally preferred, since reading a character is so simple that function call overhead is likely to exceed the cost of the useful work.

```
// Note that an int is returned, not a char
int     getc(FILE *stream);
int     fgetc(FILE *stream);
```

Note that `getchar()` is actually a macro that simply calls `getc(stdin)`:

```
#define getchar()       getc(stdin)
```

### 21.3.4 Writing Characters: `putc()` and `fputc()`

The `putc()` macro and `fputc()` function place a single character into an output stream buffer. When the buffer is full, it is flushed (written using `write()` and then marked empty).

Note that `putchar()` is actually a macro that simply calls `putc(ch, stdin)`:

```
#define putchar(ch)      putc(ch, stdout)
```

```
/*
 *  A simple "cat" command.
 *
 *  Note: Since this program has no need to examine individual
 *  characters, using stream I/O is an inefficient approach.  This
 *  is better done using low-level I/O with read() and write().
 */

#include <stdio.h>
#include <sysexits.h>
#include <errno.h>      // external errno variable
#include <string.h>     // strerror()
#include <stdlib.h>     // exit()

void    usage(char *argv[]);

int     main(int argc, char *argv[])

{
    char    *filename;
    FILE    *infile;
    int     ch;

    if ( argc != 2 )
        usage(argv);

    filename = argv[1];
    if ( (infile = fopen(filename, "r")) == NULL )
    {
        fprintf(stderr, "%s: Cannot open %s: %s\n",
                argv[0], argv[1], strerror(errno));
        return EX_NOINPUT;
    }
```

```
    while ( (ch = getc(infile)) != EOF )
        putchar(ch);

    if ( !feof(infile) )
    {
        fprintf(stderr, "Error occurred reading %s: %s\n",
                filename, strerror(errno));
        return EX_NOINPUT;
    }

    fclose(infile);

    return EX_OK;
}


void    usage(char *argv[])

{
    fprintf(stderr, "Usage: %s filename\n", argv[0]);
    exit(EX_USAGE);
}
```

### 21.3.5 Reading Lines: `fgets()`

We have already seen the `fgets()` function, which we used to input from `stdin`.

This function can be used with any stream opened by `fopen()` as well.

---

**Note** Be sure to *always* check the success status. The `fgets()` function returns `NULL` if either end-of-file or an error is encountered. Use `feof()` or `ferror()` to distinguish between the two.

---

```
    if ( (infile = fopen(filename, "r")) == NULL )
    {
        fprintf(stderr, "Cannot open %s: %s\n", filename, strerror(errno));
        exit(EX_NOINPUT);
    }

    while ( fgets(buff, BUFF_SIZE, infile) != NULL )
    {
        ...
    }

    if ( !feof(infile) )
    {
        fprintf(stderr, "Error reading %s: %s\n",
                filename, strerror(errno));
    }

    fclose(infile);
```

### 21.3.6 Writing Strings: `fputs()`

The `fputs()` function writes a string (not a line) to the given stream.

---

**Note** This function does not necessarily write a whole line. If the string does not contain a newline character, then next output will be on the same line.

---

**Note** When writing to anything other than a terminal, be sure to check the success status, so that disk full conditions or write errors are detected. This function returns a non-negative integer on success, or EOF on error. EOF is generally defined as -1.

---

```
if ( fputs(string, outfile) == EOF )
{
    // Error out
}
```

### 21.3.7 Reading Numbers and Formatted Text: `fscanf()`

The `fscanf()` function behaves exactly like `scanf()`, but can read from any stream. The following are equivalent:

```
scanf("%d", &x);
fscanf(stdin, "%d", &x);
```

### 21.3.8 Writing Numbers and Formatted Text: `fprintf()`

The `fprintf()` function behaves exactly like `printf()`, but can read from any stream. The following are equivalent:

```
printf("%d", x);
fprintf(stdout, "%d", x);
```

### 21.3.9 Detecting End-of-file: `feof()` and `EOF`

The `feof()` function returns a non-zero value if the end-of-file was reached by a previous stream I/O function call, or 0 if it was not.

It is usually more convenient to check the status of the I/O function as a loop condition and use `feof()` only to distinguish between EOF and error conditions.

### 21.3.10 Stream I/O Examples

See the book for a more complete example.

### 21.3.11 Binary I/O: `fread()` and `fwrite()`

The `fread()` and `fwrite()` functions read and write unmodified binary data to a file stream. Unlike `fscanf()` and `fprintf()`, numbers are not converted to or from text format.

Writing data in binary format is often faster, since it does not involve converting binary formats such as two's complement and IEEE floating point to or from character strings of digits, decimal points, etc. However, binary data may not be portable across hardware with different endianness or floating point formats.

```
unsigned short  x = 0xFF10;

// Write 2 bytes of raw binary data to outfile
// Bytes written are 0xFF (255) and 0x10 (16)
fwrite(&x, sizeof(x), 1, outfile);

// Converts 0xFF10 to text "ff10"
// Bytes written are 'F' (70), 'F' (70), '1' (49), and '0' (48)
fprintf(outfile, "%04X", x);
```

### 21.3.12 Addendum: Writing Robust Code

Once again, *all* input/output statements must be checked for success, except those writing to a terminal.

```
#include <errno.h>   // External errno variable
#include <string.h>  // strerror()

if ( (infile = fopen(filename, "r")) == NULL )
{
    fprintf(stderr, "Cannot open %s: %s\n", filename, strerror(errno));
    exit(EX_NOINPUT);   // man sysexits or see sysexits.h for codes
}

while ( (read_count = fscanf(infile, "%d %d", &x, &y)) == 2 )
{
    // Process input
    // If each fscanf() reads one line, increment line count
}

fclose(infile);

// If read_count is anything but 2 or EOF, something went wrong
if ( read_count != EOF )
{
    // If each fscanf() reads one line, report line number as well
    fprintf(stderr, "Error reading %s: %s\n", filename, strerror(errno));
    exit(EX_DATAERR);
}
```

### 21.3.13 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. When should we check the exit status of an input or output function?
2. When should an `fclose()` call be added to a program?
3. Show how to open a file called "records.txt" for both reading and writing, without truncating the file.
4. How do we know the difference between a read error and an end-of-file condition?
5. What are the pros and cons of `fread()` and `fwrite()` vs `fscanf()` and `fprintf()`?

# Chapter 22

# String and Character Functions

Coverage of this topic will be brief and limited to key concepts and awareness of available functions. Most of the information you need can be found in the man pages, e.g. **man strcmp**().

Students are not expected to memorize details about library functions. They should be aware of the common functions and their basic use, and how to find more information in the man pages.

## 22.1 Basic String Manipulation

A string in C is simply an array of characters.

The C language has no support for strings except for string constants (literals). String support is an inherently complex problem that the designers of C decided was best left to library functions. E.g. an operator such as == can only support one mode of comparison (case sensitive vs case insensitive, etc.), so users would have to turn to library functions for many tasks anyway.

---

**Note** C++ programmers often choose to use C string functions rather than C++ string classes.

---

C strings are *null-terminated*, i.e. terminated by a 0 byte denoted as `'\0'` to clarify that it's a character.

---

**Note** The data type of `'\0'` and any other constant in single quotes is actually `int`, but the quotes indicate to the reader that it's a character rather than a number.

---

---

**Note**
When defining a a fixed size array to hold a string, it's a common practice to add a `+ 1` to the size to accommodate the null byte.

```
    // This array can really hold a name up to NAME_MAX_CHARS
    char    first_name[NAME_MAX_CHARS + 1];
    int     ch;

    // This loop would need NAME_MAX_CHARS - 1 if not for the + 1 above
    for (c = 0; (ch = getchar()) != '\n') && (c < NAME_MAX_CHARS); ++c)
        first_name[c] = ch;
```

---

### 22.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is a string in C?

## 22.2 String Functions

Prototypes and other relevant items relating to standard library string functions are defined in `string.h`.

### 22.2.1 Copying Strings: `strlcpy()`

**Caution**
The original string copy function, `strcpy()`, does not know the size of the destination array. It can cause corruption of adjacent variables and possibly security holes. Most people agree that `strcpy()` should never be used.
This function was created early in the development of C and Unix, following the "trust the programmer" philosophy. However, at the time, "the programmer" referred to Dennis Ritchie, Ken Thompson, or a few other world-class computer scientists working at Bell Labs. Once the use of C expanded into a broader community, it became clear that certain standard library functions were too dangerous for general use.
The `strncpy()` was a replacement that stops at a specified number of characters, but does not null-terminate the string if it is truncated. This will almost certainly lead to problems later.
The `strlcpy()` function is a non-standard function that works like `strncpy()`, but ensures that the target string is always null-terminated.

The GNU community has so far declined to add `strlcpy()` to its standard libraries, citing some ideological arguments, which have validity, but pragmatically speaking, don't apply to every situation.

In reality, copying strings, or arrays of any kind for that matter, is something that should almost never happen. It's a waste of CPU time and memory. There are exceptions, of course.

If a function like `strlcpy()` has to truncate a string, then one could argue that there is a bug in the code. But sometimes a truncated string is the best option available. It depends on the application, of course. String manipulation is inherently complex and troublesome.

The `strsqueeze()` function in libxtend is another alternative that removes text from the *middle* of a string in order to fit it within a limited number of characters. This is useful for abbreviating long pathnames, for example, and is used by the APE editor for this purpose.

**Caution**
We can assign one char pointer to another, but this *does not copy the string*.

```
char    name1[NAME_MAX_CHARS + 1],
        name2[NAME_MAX_CHARS + 1],
        *p1, *p2;

        name1 = name2;  // This will cause a compiler error since
                        // name1 is a pointer constant, not a variable
        p1 = name;      // Not a copy, p1 and name now point to the
                        // same string
        p1 = p2;        // Not a copy, p1 and p2 now point to the
                        // same string
```

Sometimes this is what we want. In fact, assigning a pointer is much more efficient that copying a string, since it's a scalar operation, whereas a string copy requires a loop. If we don't really need a second copy of the string, using pointers this way is preferable.

### 22.2.2 Duplicating Strings: `strdup()`

The `strdup()` function uses `malloc()` to allocate an exact size array for a string and then copies the string to it.

**Caution** You must remember to free the memory allocated by `strdup()`, or you will have a memory leak.

This is good for building `argv` style arrays.

```
    char    temp_name[NAME_MAX_CHARS + 1],
            *names[MAX_NAMES];
    size_t  c, name_count;

    // Assume read_name() reads a string using fgets() or similar
    for (c = 0; (read_name(temp_name) != EOF) && (c < MAX_NAMES); ++c)
    {
        if ( (names[c] = strdup(temp_name)) == NULL )
        {
            fprintf(stderr, "Could not allocate memory for names[%zu].\n", c);
            exit(EX_UNAVAILABLE);
        }
    }

    // Loop may terminate due to EOF or read error
    if ( ferror(stdin) )
    {
        fputs("Error reading names.\n", stderr);
        exit(EX_DATAERR);
    }
    name_count = c;

    ...

    // When we're done with the names...
    for (c = 0; c < name_count; ++c)
        free(names[c]);
```

### 22.2.3 Finding String Length: `strlen()`

The `strlen()` function loops through a string to find the null byte at the end and then returns the length.

```
int     strlen(const char *string)

{
    char    *p;

    for (p = string; *p != '\0'; ++p)
        ;

    return p - string;
}
```

```
    char    *string;
    size_t  length;

    length = strlen(string);
```

Most null-terminated string operations are highly efficient since they minimize iterations rather than process the unused portions of a character array. The `strlen()` function is an exception, because getting the length of a string would not require a loop at all if we implemented strings in a way that keeps track of the length, e.g.

```
// An alternative to null-terminated strings
typedef struct
{
    size_t  length;
    char    *string;
}   string_t;
```

Avoid using `strlen()` as much as possible, by storing frequently used string lengths in a `size_t` variable.

### 22.2.4 Comparing Strings: `strcmp()` and strcasecmp()

In C, we use a library function to compare strings. Many languages support using operators such as == on strings. This is inherently complex, however, as there are different types of string comparison, e.g. case sensitive vs insensitive, lexical vs numeric, fixed-length vs null-terminated, etc.

The `strcmp()` function returns 0 if two strings are equal, a value less than 0 if the first string is lexically less than the second, and a value greater than 0 otherwise.

```
    char    *string1, *string2;

    ...

    if ( strcmp(string1, string2) > 0 )
    {
        ...
    }
```

**Caution**

You can use == on string objects in C, but it will compare the *addresses*, not the string objects.

```
    // Tells us whether string1 and string2 point to the same string
    if ( string1 == string2 )
    {
        ...
    }
```

Use `strcasecmp()` to compare strings without regard for upper and lower case letters. ( "Max" is the same a "MAX" or "max". )
Use `memcmp()` to compare a fixed number of characters rather than everything to the null terminator.

### 22.2.5 Concatenating Strings: `strlcat()`

The original function for concatenating strings was `strcat()`. It should not be used for the same reason as `strcpy()`.

```
    // Append string2 to string1
    // Does not know the size of the string1 array and could run off the end
    strcat(string1, string2);
```

The `strlcat()` function is also not standardized, but is available on many systems other than GNU/Linux and is available in libxtend on all systems.

```
    char    string1[MAX_LEN + 1], string2[MAX_LEN + 1];

    // Append string2 to string1
    // Third argument indicates the size of the destination array
    strlcat(string1, string2, MAX_LEN + 1);
```

Like `strlcpy()`, this could result in a truncated string. Use where appropriate.

### 22.2.6 Searching Strings: `strstr()` and `strchr()`

```
// Prototype
char    *strstr(const char *big, const char *little);
char    *strchr(const char *string, int character);
```

The `strstr()` function checks to see if the string `little` is a substring of `big`. It returns a pointer to the first occurrence of `little` within `big`, or `NULL` if it is not a substring.

The `strrchr()` function searches in reverse and returns the *last* occurrence of `little` within `big`.

The `strchr()` and `strrchr()` functions similarly search for a single character within a string.

---

**Note** There are numerous related library functions. Check the "SEE ALSO" section of the man page for more insight.

---

### 22.2.7 Building Formatted Strings: `snprintf()`

The `snprintf()` function copies formatted data into a string in the same way `fprintf()` sends it to a stream.

```
    char    string[MAX_CHARS + 1];

    snprintf(string, MAX_CHARS, "2 ^ %d is %0.2f.\n", c, power(2.0, c));
```

### 22.2.8 Tokenizing Strings: `strsep()`

A common task in programming is breaking up a string into pieces that are separated by whitespace or other delimiters. The `strsep()` function makes this easy. It returns the address of the next token separated by any character in the string "delim" and updates stringp to point to the first character after that token.

```
// Prototype
char *strsep(char **stringp, const char *delim);
```

```
    char    *string, *p, *tokens[MAX_TOKENS];
    int     c;

    // Build an argv-style array of tokens
    for (c = 0, p = string; (token[c] = strsep(&p, " \t")) != NULL; ++c)
        ;
    token[c] = NULL;
```

---

**Caution** The `strsep()` function is destructive. It replaces the delimiters with null bytes in the original string.

---

Addendum: The similar and older `strtok()` function is considered obsolete.

### 22.2.9 Addendum: More Information

You can find out more about string functions by checking the SEE ALSO sections of string function man pages, using **man string** on some systems, and by looking at header files, e.g. **more /usr/include/string.h**

### 22.2.10 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Why should we avoid copying strings (or other arrays)?
2. What is the problem with `strcpy()`?
3. Can we use the assignment operator, =, to copy strings?
4. How can we copy a string, ensuring that the target array is the correct size? Are there any risks?
5. What is the down side of using `strlen()` and how can we avoid it?
6. Why doesn't C support string comparison with ==?
7. Where can we find more information about string functions?

## 22.3 Classifying Characters: The ctype Functions

The header `ctype.h` contains numerous macros for classifying characters.

```
int     isalnum(int);
int     isalpha(int);
int     iscntrl(int);
int     isdigit(int);
int     isgraph(int);
int     islower(int);
int     isprint(int);
int     ispunct(int);
int     isspace(int);
int     isupper(int);
int     isxdigit(int);
int     tolower(int);
int     toupper(int);
```

```
    if ( isalpha(ch) )
    {
        // ch is a letter
    }
```

Upper and lower case letters in the ASCII/ISO character sets differ by 32 ($2^5$), e.g. 'A' is 01000001 (65 decimal) while 'a' is 01100001 (97 decimal). The `toupper()` and `tolower()` macros simply need to toggle bit 5:

```
#define toupper(ch) (islower(ch) ? (ch) & 0xDF : (ch))  // Mask = 11011111
#define tolower(ch) (isupper(ch) ? (ch) | 0x20 : (ch))  // Mask = 00100000
```

Addendum: Modern POSIX-compliant implementations will not affect non-letters.

### 22.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What can we find in the header `ctype.h`?

## 22.4 Pattern Matching Functions

**Note** Students will not be expected to write code using pattern matching functions on quizzes or exams. They need only understand what regular expressions and globbing patterns are and be familiar with the functions for handling them.

### 22.4.1 The Regex Functions

Regular expressions are patterns that can be used to match many different strings.

| 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) |
| ^ | Beginning of the line |
| $ | End of the line |
| .* | 0 or more of any character |
| [a-z]* | 0 or more lower-case letters |

Table 22.1: RE Patterns

To use regular expressions in C, we first "compile" them using `regcomp()`. This converts the character string pattern to a structure that can be compared much more efficiently by `regexec()`.

```
char    *string = "I'm Joe and I'm 10.5 years old.",
        *pattern = "[0-9]+\.[0-9]+";    // Real number pattern
regex_t compiled_regex;
regmatch_t  match[1];

if ( regcomp(&compiled_regex, pattern, REG_EXTENDED) == 0 )
{
    if ( regexec(&compiled_reg, string, 1, match, 0) == 0 )
        printf("Match found at character %lu.\n",
                (unsigned long)match[0].rm_so);
}
regfree(&compiled_regex);
```

These are the functions used by Unix commands such as **grep**, **sed**, **awk**, etc., which search and manipulate files containing regular expressions. For more information, read the man pages on `regcomp()`, `regexec()`, and related functions.

### 22.4.2 File Specification Matching: `fnmatch()` and `glob()`

Globbing patterns, as often used in Unix commands, look similar to regular expressions, but are not the same and do not serve the same purpose.

A globbing pattern always matches existing path names in a filesystem, whereas regular expressions look for patterns in a character string. Globbing is covered here rather than in the files chapter in order to contrast it to regular expressions.

Globbing patterns are covered in Section 4.7.2.

```
int fnmatch(const char *pattern, const char *string, int options);
int glob(char *pattern, int options, func_t errfunc, glob_t *pathlist);
```

The `fnmatch()` function returns 0 if the given string (normally a pathname) matches `pattern`.

The `glob()` function allocates and populates `pathlist` with a list of pathnames in the CWD that match `pattern`.

```
    code = glob("*.c",0,NULL,&paths);
    if (code == 0)
    {
        for (p=paths.gl_pathv; *p != NULL; ++p)
            puts(*p);
        globfree(&paths);
    }
```

### 22.4.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Show a regular expression that matches an octal integer constant.
2. Show a globbing pattern that matches all of the object files in the directory Program1.
3. What is the difference between '*' in a regular expression (RE) and a globbing pattern?

## 22.5 Bulk Memory Manipulation

### 22.5.1 Copying Blocks: `memcpy()` and `memmove()`

These functions copy fixed-length blocks of memory as efficiently as possible. They are often more efficient than using a loop, since implementations can take advantage of specific hardware features such as vector instructions.

They are identical, except that `memmove()` is safe to use when the target overlaps the source. The `memcpy()` function may overwrite bytes in the source string before they are copied to the target, resulting in an incorrect copy.

```
|------------ source ------------|
    |------------ target ------------|
     ^                           ^
     If we copy from left to right, bytes in this range are overwritten
     before they are copied from the source.
```

```
    car_t   cars1[MAX_CARS], cars2[MAX_CARS];
    size_t  c;

    // Ordinary loop method
    for (c = 0; c < MAX_CARS; ++c)
        cars2[c] = cars1[c];

    // This may be faster, depending on the library and processor
    memcpy(cars2, cars1, MAX_CARS * sizeof(car_t));
```

### 22.5.2 Comparing Blocks: `memcmp()`

The `memcmp()` function compares memory byte-by-byte like `strcmp()`, but using a fixed length rather then to a null terminator.

```
if ( memcmp(string1, string2, 10) == 0 )
{
    // The first 10 bytes are the same
}
```

The `memcmp()` function is significantly faster the `strcmp()` on some platforms, since it does not need to compare one byte at a time while searching for a null terminator. On a 64-bit computer, it can compare 64 bits (8 bytes) at a time, possibly more using vector operations available on some CPUs.

### 22.5.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is the difference between `memcpy()` and `memmove()`?
2. Does `memcmp()` have any advantages over `strcmp()`?

# Chapter 23

# Odds and Ends

Coverage of this topic will be brief and limited to key concepts and awareness of available functions. Most of the information you need can be found in the man pages, e.g. **man qsort()**.

Students are not expected to memorize details about library functions. They should be aware of the common functions and their basic use, and how to find more information in the man pages.

## 23.1 Math Functions

Math functions, as we have seen, mostly reside in the standard math library, `libm.a` or `libm.so`. Programs must be linked with the `-lm` flag to search `libm.*`. Most other standard library functions reside in `libc.*`, which is always searched by the linker. I.e., we do not need to use `-lc`.

Prototypes, macros, and related derived types are defined in `/usr/include/math.h`.

On some systems, such as FreeBSD, we can obtain reference information about available math functions by running **man math**.

On any system, we can browse the header files, e.g.

```
shell-prompt: more /usr/include/math.h
```

### 23.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. Where do standard math functions reside, and what link option to we need to use them in our programs?
2. What is the portable way to find out what math functions are available on our system?

## 23.2 Data Conversion Functions

The standard libraries contain many functions for converting between strings and numbers. Among the most generic is `strtol()`, which converts a string to an integer using any base. The `strtol()` function could be used by `scanf()` to support the "%ld" and "%lx" specifiers, for example.

```
    char    *string = "-8234", *end;
    long    line_count;

    // strtol() returns the integer value converted from the string.
    // end is set to the address of the first character that
    // could not be converted as part of the number.  If the
    // entire string is supposed to be a number, this should point
    // to a null terminator ('\0') after the conversion.  If
    // end == string after calling strtol(), then no digits were found.
    line_count = strtol(string, &end, 10);
    if ( *end != '\0' )
    {
        fprintf(stderr, "Invalid argument: %s.  Expected an integer.\n",
                string);
        exit(EX_USAGE);
    }
```

The `strtoll()` converts to `long long`. For smaller integers such as `short` and `int`, we just use `strtol()`.

The `strtof()`, `strtod()`, and `strtold()` functions convert to a `float`, `double`, and `long double` (base 10 only).

Converting numbers to strings is usually done using `snprintf()`, which writes the string to a character array rather than a file stream like `printf()` or `fprintf()`.

```
// Returns the number of characters converted, or a negative
// value if there was an error.
if ( snprintf(string, MAX_DIGITS + 1, "%d", -8234) < 0 )
{
    // Error out
}
```

Run **man strtol** or **man snprintf**()for details, and check the "SEE ALSO" section for related functions.

### 23.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What function can we use to convert a string to an integer?
2. What function can we use to convert an integer to a string?

## 23.3 Random Numbers

The standard libraries contain functions for generating *pseudorandom* numbers ("pseudo-" = false). This begs the question: What does random mean? How does a pseudorandom number differ from a "true" random number?

Everything that happens in the universe has a cause, and is therefore predictable in theory, though not always in practice. We can't predict where an electron will be in its orbit around the nucleus an atom because we are unable to observe its current position and trajectory. The best we can do is define the typical bounds of its orbit, known as the *electron cloud*.

"Random" is really just a synonym for unpredictable *in practice*. The goal of pseudorandom number generators is to make it as difficult as possible to predict the next number in the sequence.

The `random()` function returns a pseudorandom number with a uniform distribution between 0 and $2^{31}$ - 1 that is good enough for most purposes.

```
    // Print a random number
    printf("%ld\n", random());
```

Unless we initialize the random number generator by calling `srandom()` with a different seed each time, `random()` will always generate the same sequence. One simple trick is to use the current time, which will provide a new seed as long as at least 1 second has passed since the last call to `srandom()`.

```
    srandom(time(NULL));
```

### 23.3.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What does "random" mean?
2. What should we do before using the `random()` function to ensure somewhat unpredictable results?

## 23.4 Basic Process Control

### 23.4.1 Normal Termination: `exit()`

The `exit()` function terminates a process in basically the same way as a `return` statement in `main()`, but `exit()` can be called from within any function. Don't use this in `main()`, just use `return`.

```
    if ( scanf("%zu", &count) != 1 )
    {
        fputs("Error reading count.\n", stderr);
        exit(EX_DATAERR);
    }
```

### 23.4.2 Last Requests: `atexit()`

Skip for now.

### 23.4.3 Creating a Core File: `abort()`

The `abort()` function aborts the current process and generates a core file, which can be used with a debugger to pinpoint problems in the program. The same effect can be accomplished by sending a `SIGABRT` signal to the process.

```
shell-prompt: kill -ABRT 413    // Abort process with PID 413
```

```
    abort();
```

### 23.4.4 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What does `exit()` take as an argument?

## 23.5 Manipulating the Environment

### 23.5.1 Reading the Environment: `getenv()`

```
    char    *ostype;

    // Get OSTYPE environment variable.
    ostype = getenv("OSTYPE");
```

### 23.5.2 Writing to the Environment

```
    // Set environment variable EDITOR to "nano", overwriting if it
    // is already set.  0 in the third argument prevents overwriting (clobbering).
    setenv("EDITOR", "nano", 1);
```

### 23.5.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How can a C program find out the value of the `TERM` environment variable and assign it to a string variable called `term_type`?

## 23.6 Sorting and Searching

### 23.6.1 Sorting: `qsort()`

This is a *polymorphic* function, i.e. it can sort an array of any data type. This is made possible by passing the size of each object and a pointer to a comparison function as arguments. This allows `qsort()` to compare and swap elements of any type. The comparison function must have the same interface as `strcmp()`.

```
    char    strings[MAX_STRINGS][MAX_STRING_LEN + 1];

    // strcmp with no arguments represents the address of the function
    // (much like an array name with no subscript)
    // Case strcmp to take two void pointers to silence compiler warnings
    qsort(strings, MAX_STRINGS, MAX_STRING_LEN + 1,
            (int (*)(const void *, const void *))strcmp);
```

Moving entire strings around this way is horrifically inefficient. Sorting pointer arrays is an order of magnitude faster than arrays of large objects (strings, structures). For this we need a compare function that takes pointers to pointers and compares the strings.

```
// This compare function and several others suitable for qsort() are
// available in libxtend
int     strptrcmp(const char **p1, const char **p2)

{
    return strcmp(*p1, *p2);
}
```

Quicksort is known to be an unstable algorithm, e.g. performance is poor if the list is partially sorted and the pivot is not carefully selected. Basic quicksort algorithm becomes selection sort if the list is sorted and the first element in the list is chosen as the pivot. This is easily avoided by choosing the middle element as the pivot instead. Most library implementations should be stable.

**Note** FreeBSD also has `heapsort()` and `mergesort()` in the standard library. These are not portable to Linux and some other Unix-like systems.

### 23.6.2 Searching: `bsearch()`

Normally, inhaling large amounts of data into memory should be avoided. It hurts memory performance (hierarchy) and limits programs to available memory. However, if we want to search a list many times, loading it into memory and using a binary search will likely pay off.

Uses same compare functions as `qsort()`.

```
char    *result;

result = bsearch("aardvark", strings, MAX_STRINGS, MAX_STRING_LEN + 1,
        (int (*)(const void *, const void *))strcmp);
if ( result == NULL )
{
    // String not found
}
```

### 23.6.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. How do functions like `qsort()` and `bsearch()` manage to be polymorphic?

## 23.7 Functions with Variable Argument Lists

Skip for now.

### 23.7.1 ANSI Form: `stdarg.h`

### 23.7.2 Unix Form: `varargs.h`

## 23.8 Addendum: Advanced: `tgmath.h` (C99)

Skip for now.

# Chapter 24

# Working with the Unix Filesystem

Coverage of this topic will be brief and limited to key concepts and awareness of available functions. Most of the information you need can be found in the man pages, e.g. **man -a chown()**.

Students are not expected to memorize details about library functions. They should be aware of the common functions and their basic use, and how to find more information in the man pages.

## 24.1 File Information: `stat()` and `fstat()`

The `stat()` function reads *inode* information (file metadata) for a given pathname.

There is also a **stat** Unix command, so run **man 2 stat** to learn about the function. **man -a stat** will display *all* man pages for stat commands and functions.

```
#include <sys/stat.h>

...

    struct stat st;
    char    *path = "input.txt";

    if ( stat(path, &st) == 0 )
    {
        printf("File size is %lu.\n", st.st_size);
        printf("Owner UID is %u.\n", st.st_uid);
        printf("Permissions and other mode info is %o.\n", st.st_mode);
        if ( st.st_mode & S_IFREG )
            printf("%s is a regular file.\n", path);
    }
    else
    {
        // Error out
    }
```

The **fstat()** function provides the same information, but takes a file descriptor rather than a pathname.

### 24.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What function can we use to find out about the permissions on a file? A file descriptor?
2. How can we find out more about the `stat()` function?

## 24.2 Changing File Information

Many Unix commands have similar counterparts in the C library.

### 24.2.1 Changing Ownership: `chown()`

Skip for now. Can only be used by root user.

### 24.2.2 Changing Permissions: `chmod()`

As discussed in Section 4.3.3, Unix permissions are made up of 9 bits, read, write, and execute bits for "user", "group", and "other".

The header file `sys/stat.h` defines named constants for each of these bits, which we can and should use to make our code self-documenting:

```
#define S_IRWXU 0000700                 /* RWX mask for owner */
#define S_IRUSR 0000400                 /* R for owner */
#define S_IWUSR 0000200                 /* W for owner */
#define S_IXUSR 0000100                 /* X for owner */

#define S_IRWXG 0000070                 /* RWX mask for group */
#define S_IRGRP 0000040                 /* R for group */
#define S_IWGRP 0000020                 /* W for group */
#define S_IXGRP 0000010                 /* X for group */

#define S_IRWXO 0000007                 /* RWX mask for other */
#define S_IROTH 0000004                 /* R for other */
#define S_IWOTH 0000002                 /* W for other */
#define S_IXOTH 0000001                 /* X for other */
```

```
    if ( stat(path, &st) == NULL )
    {
        // Add group write permissions by ORing in the group write bit
        // S_IWGRP  = 0000020 octal = 000... 000 010 000 binary
        st.st_mode |= S_IWGRP;

        // Remove all permissions for "other" using AND with the
        // complement of all the "other" permission bits
        // S_IROTH|S_IWOTH|S_IXOT       = 0000007 octal
        // ~(S_IROTH|S_IWOTH|S_IXOT)    = 7777770 octal
        st.st_mode &= ~(S_IROTH|S_IWOTH|S_IXOTH)

        chmod(path, st.st_mode);
    }
```

### 24.2.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Briefly describe the process of adding read permissions for "other" on a file from a C program. Or write the code if you wish.

## 24.3 Accessing Directories

Skip for now.

### 24.3.1 Reading Directories

### 24.3.2 Creating Directories

# Chapter 25

# Low-Level I/O

Coverage of this topic will be brief and limited to key concepts and awareness of available functions. Most of the information you need can be found in the man pages, e.g. **man -a open**().

Students are not expected to memorize details about library functions. They should be aware of the common functions and their basic use, and how to find more information in the man pages.

## 25.1 Why Use Low-level I/O

Low-level I/O is a direct interface to the kernel's input and output routines. Unlike `FILE` streams, there is no buffering, no individual character input or output like `getc()` and `putc()`, and no numeric input or output like `fprintf()` and `fscanf()`.

---

**Note** ALL input and output to/from any file or device is ultimately performed by the low-level I/O functions. FILE streams are an abstraction built on top of low-level I/O.

---

With low-level I/O, we simply read and write fixed size blocks of bytes. By eliminating `getc()` and `putc()`, and the buffers they manage, we use far less CPU time than `FILE` streams, and may achieve better I/O throughput, but only when reading or writing blocks. FILE streams make it easier and more efficient to read or write smaller pieces of data like individual characters.

Most communication via pipes and sockets (covered later) is done using low-level I/O. `FILE` streams are primarily used for terminals and human-readable files.

### 25.1.1 Cat: A Bad Example

Suppose we write a simple **cat** command with no features that require inspecting individual characters. We might do it as follows:

```
#include <stdio.h>
#include <sysexits.h>
#include <string.h>
#include <errno.h>

int     main(int argc,char *argv[])

{
    FILE    *fpin;
    char    *filename;
    int     ch;

    switch(argc)
    {
```

```
        case    1:
            fpin = stdin;
            filename = "stdin";
            break;

        case    2:
            filename = argv[1];
            if ( (fpin = fopen(filename,"r")) == NULL )
            {
                fprintf(stderr,"Could not open file: %s: %s\n",
                        filename, strerror(errno));
                return EX_NOINPUT;
            }
            break;

        default:
            fprintf(stderr,"Usage: %s [file]\n",
                    argv[0]);
            return EX_USAGE;
    }

    while ( (ch=getc(fpin)) != EOF )
        putchar(ch);

    if ( !feof(fpin) )
    {
        fprintf(stderr, "Error reading %s: %s\n",
                filename, strerror(errno));
        return EX_DATAERR;
    }

    fclose(fpin);

    return EX_OK;
}
```

The problem here, is that this program loads a block of data into the `fpin FILE` buffer, then copies it to the `stdout FILE` buffer one character at a time using `getc()` and `putc()`, and finally writes the `stdout FILE` buffer to the standard output. Given that the program does not care what any of the characters are, this buffer copying is an enormous waste of time.

Low-level I/O can do this much more efficiently, as we will see shortly.

### 25.1.2 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is the advantage of using low-level I/O over FILE streams?
2. What is the limitation of low-level I/O compared with FILE streams?

## 25.2 Basic Input and Output

### 25.2.1 Opening Files: `open()`

The `open()` function works like `fopen()`, but returns a low-level integer file descriptor instead of a `FILE` structure.

The `fopen()` function actually creates a FILE structure and calls `open()`. Each FILE structure contains the file descriptor returned by `open()` for performing low-level reads and writes of the buffer contents. For example, when a FILE stream write buffer is full, its contents are written out using a low-level `write()` to the file descriptor returns by `open()`.

Descriptor 0 is the standard input, 1 is the standard output, and 2 is the standard error. The `open()` function returns successively higher values, or -1 if an error occurred. E.g., the first descriptor returned by `open()` will be 3, assuming all the standard streams are already open.

The open mode for `open()` is given by a single integer argument made up of bit flags. Constants are defined in `fcntl.h`, which we can OR together to define the open mode. A few of them are described in Table 25.1. These named constants *must* be used for both portability and readability. We must never specify the open mode as a hard-coded integer constant.

---

**Note** Bit flags are a very common theme in systems programming. We saw them in `chmod()` and now in `open()`. There are numerous other standard library functions that use the same technique to receive multiple Boolean values through one integer argument.

---

| Bit flag | Meaning |
|---|---|
| O_RDONLY | Open for reading only |
| O_WRONLY | Open for writing only |
| O_CREAT | Create the file if it does not exist. open() will fail without this flag. |
| O_RDWR | Open for reading and writing |
| O_APPEND | Append rather than overwrite |
| O_TRUNC | Truncate existing file on open |

Table 25.1: Mode bits for open()

```
// File descriptor, part of the FILE structure
int     infd;

if ( (fd = open(path, O_RDONLY)) == -1 )
{
    fprintf(stderr, "Could not open %s: %s\n",
            path, strerror(errno));
    return EX_NOINPUT;
}
```

```
// File descriptor, part of the FILE structure
int     outfd;

// When creating a file, we also specify the permissions as a 3rd argument
if ( (fd = open(path, O_WRONLY|O_CREAT, 0644)) == -1 )
{
    fprintf(stderr, "Could not open %s: %s\n",
            path, strerror(errno));
    return EX_CANTCREAT;
}
```

### 25.2.2 Reading Files: `read()`

The `read()` function reads as many bytes as requested in the third argument, or fewer if it reaches the end of the file. It returns the actual number of bytes read, or -1 if there was an error.

---

**Note** The `read()` function is called by `getc()` and some other FILE stream functions to fill the stream buffer when it is empty.

---

```
    ssize_t bytes;  // Signed size_t to allow for negative error codes
    char    buff[BUFF_SIZE];

    if ( (bytes = read(infd, buff, BUFF_SIZE)) == -1 )
    {
        fprintf(stderr, "Could not read %s: %s\n",
                path, strerror(errno));
        return EX_DATAERR;
    }
```

### 25.2.3 Writing Files: `write()`

The `write()` function writes as many bytes as requested in the third argument, or fewer if it the medium becomes full. It returns the actual number of bytes written, or -1 if there was an error.

---

**Note** The `write()` function is called by `putc()` and some other FILE stream functions to flush the stream buffer when it is full.

---

```
    ssize_t bytes;  // Signed size_t to allow for negative error codes

    if ( (bytes = write(outfd, buff, BUFF_SIZE)) == -1 )
    {
        fprintf(stderr, "Could not write %s: %s\n",
                path, strerror(errno));
        return EX_DATAERR;
    }
```

### 25.2.4 Closing Files: `close()`

---

**Note** The `close()` function is called by `fclose()`, which then deallocates the FILE structure created by `fopen()`.

---

```
    close(infd);
```

### 25.2.5 Moving within a File: `lseek()`

```
    lseek(infd, 0, SEEK_SET);   // Rewind to beginning of file
```

### 25.2.6 Cat: A Better Example

Below is a simple **cat** command that uses low-level I/O for greater efficiency. Note that in order to add features such as highlighting control characters, we would need to examine each character, and we might as well use FILE streams and `getc()`.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sysexits.h>
#include <string.h>
#include <errno.h>
```

```c
/* Relatively large buffer will mean few read and write calls */
#define BUFF_SIZE   65536

int     main(int argc,char *argv[])

{
    char    buff[BUFF_SIZE+1], *filename;
    int     infd;
    size_t  bytes;

    switch(argc)
    {
        case    1:  /* Used as filter */
            infd = fileno(stdin);
            filename = "stdin";
            break;

        case    2:  /* Open the given filename */
            filename = argv[1];
            infd = open(filename, O_RDONLY);
            if ( infd == -1 )
            {
                fprintf(stderr, "Could not open file: %s: %s\n",
                        argv[1], strerror(errno));
                return EX_NOINPUT;
            }
            break;

        default:
            fprintf(stderr,"Usage: %s [file]\n",
                    argv[0]);
            return EX_USAGE;
    }

    /* Read and write 64k blocks */
    while ( (bytes=read(infd,buff,BUFF_SIZE)) > 0 )
        write(fileno(stdout),buff,bytes);

    close(infd);

    if ( bytes == -1 )
    {
        fprintf(stderr, "Error reading %s: %s\n",
                filename, strerror(errno));
        return EX_DATAERR;
    }

    return EX_OK;
}
```

### 25.2.7 Choosing the Right Buffer Size

Skip for now. See book.

### 25.2.8 Handling Multiple Files: `select()`

Skip for now. See book.

### 25.2.9 Controlling File Descriptors: `fcntl()`

Skip for now. See book.

### 25.2.10 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What does `open()` return? How does it relate to what `fopen()` returns?
2. How does `write()` relate to `putc()` and other FILE stream functions?

# Chapter 26

# Controlling I/O Device Drivers

Skip for now. Will come back to it if we run out of material, which is unlikely.

## 26.1 Termios

Interface based on RS-232 serial port / modem protocol. This was the standard for dumb terminals used with Unix systems until the 1990s, when Ethernet took over.

Also commonly used to communicate with embedded microcontrollers. RoboCTL.

### 26.1.1 Input Flags

### 26.1.2 Output Flags

### 26.1.3 Control Flags

### 26.1.4 Local Flags

### 26.1.5 Control Characters

### 26.1.6 The Termios Functions

### 26.1.7 Alternatives to the Termios Interface

#### Other Low-evel Methods

#### High-level Libraries

Curses, twintk

Addendum: Commercial libraries like Vermont Views are defunct.

# Chapter 27

# Unix Processes

## 27.1 Creating Processes

A *process*, as discussed in Section 4.2.2, is the execution of a program. If Joe and Sarah are both running the same program, there is one program, but two processes.

We can list currently running processes using **ps**:

```
FreeBSD moray.acadix  bacon ~ 999: ps ax
  PID TT  STAT        TIME COMMAND
    0  -  DLs      7:26.21 [kernel]
    1  -  SLs      0:00.28 /sbin/init
    2  -  DL       0:00.00 [KTLS]

[ numerous processes omitted ]

71502  0  Is       0:00.04 /bin/tcsh
72155  0  S+       0:00.03 ape processes.dbk
72216  1  Ss       0:00.05 /bin/tcsh
72241  1  R+       0:00.00 ps ax
```

A very convenient way to run another process from within a C program is the `system()` function, which takes a shell command as a string argument:

```
    system("ls -als /etc | more");
```

This is very convenient, since it provides access to all the features built into the Bourne shell, such as redirection, pipes, etc. This approach has a lot of overhead, however, since it starts up a Bourne shell process, and then passes the command to the shell process.

For more efficient and tunable process creation, we turn to the traditional `fork()` and `exec()` approach, or the newer *POSIX spawn* interface.

### 27.1.1 Creating Processes: `fork()`

Unix has a special function that is called from one place, but returns to two places. This is possible, because the function clones the process that called it. This function returns 0 to the new process (the child process) and returns the process ID (PID) of the child process to the original process that called it (the parent process).

```
            /*
             *  Clone the calling process.  The ONLY difference between
             *  the two processes immediately after fork() is their
             *  PIDs and the return value they receive from fork().
```

```
 */

if ( fork() == 0 )
{
    // This clause is only run by the child process
}
else
{
    // This clause is only run by the parent process
}
```

### 27.1.2 Transforming Processes: `execve()`

Of course, having two processes doing exactly the same thing to the same input data is of no use. So, after the parent process is cloned, one of them must evolve so that it can do something that the other is not doing.

The *exec* family of functions replace the program being run by calling process. These functions load a new program into the calling process, replacing the program that called them, and call the `main()` function of the new program.

The `execl()` function and derivatives take the path of a program followed by a list of separate arguments, which are combined into an `argv` style pointer array and passed to `main()` of the new program. The `execlp()` function uses the `PATH` environment variable to find the command.

```
int     execl(const char *path, const char *argv0, const char *argv1, ...);
```

These functions take a variable number of arguments, like `printf()`. To indicate the end of the argument list, we pass `NULL` as the last argument.

```
execl("/bin/ls", "ls", NULL);
```

---

**Note** The command is given twice, once as the path, and again as `argv0`.

---

```
if ( fork() == 0 )
{
    // This clause is only run by the child process
    execlp("ls", "ls", "-als", "/etc", NULL);

    // An exec function should not return
    // If we're running this, the exec failed
    fprintf(stderr, "exec failed: %s\n", strerror(errno));
    exit(EX_OSERR);
}
else
{
    // This clause is only run by the parent process
}
```

If we have the arguments for the new program already in an `argv` style array, we can use the `execv()` family of functions instead.

```
if ( fork() == 0 )
{
    // This clause is only run by the child process
    char    *new_argv[] = { "ls", "-als", "/etc", NULL };

    execvp("ls", new_argv);
```

```
            // An exec function should not return
            // If we're running this, the exec failed
            fprintf(stderr, "exec failed: %s\n", strerror(errno));
            exit(EX_OSERR);
        }
        else
        {
            // This clause is only run by the parent process
        }
```

---

**Note**

There is also an internal **exec** command in Unix shells that replaces the shell process with a new process running a different program. This simply causes the shell not to call `fork()` before calling an `exec` function.

```
# Runs ls in place of the shell process instead of under it
# Once ls is done, this shell session is finished
shell-prompt: exec ls
```

---

### 27.1.3 Waiting for Godot: `wait()`

The `wait()` function causes a parent process to pause until one of its children terminates. The PID of the child process is returned, and the exit status is placed at the address provided by the argument.

```
        if ( fork() == 0 )
        {
            // This clause is only run by the child process
            char    *new_argv[] = "ls", "-als", "/etc", NULL };

            execvp("ls", new_argv);

            // If we're running this, the exec failed
            fprintf(stderr, "exec failed: %s\n", strerror(errno));
            exit(EX_OSERR);
        }
        else
        {
            // This clause is only run by the parent process
            int     status;

            pid = wait(&status);

            printf("Child process %d returned %d.\n", pid, status);
        }
```

### 27.1.4 A Complete Example

The one-page **skel-shell** program below actually implements a functional Unix shell that can run any external command in the `PATH`, and implements one internal command, **exit**.

This can be extended to add features such as redirection and pipes, more internal commands such as **cd**, command-line editing, etc.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
```

```c
#include <sysexits.h>

#define COMMAND_LEN 1024
#define MAX_ARGS    1024

int     main()

{
    char    command[COMMAND_LEN+1], *command_ptr,
            *new_argv[MAX_ARGS], **arg_ptr, *path;
    int     status;

    /* Input commands until "exit" or Ctrl+d is entered */
    do
    {
        fputs("skel-shell: ", stdout);
        if ( fgets(command,COMMAND_LEN,stdin) == NULL )
            return EX_OK;   // Ctrl+d = EOF

        /* Set up argv array for execvp() */
        arg_ptr = new_argv, command_ptr = command;
        while ( ((*arg_ptr = strsep(&command_ptr, " \t\n")) != NULL) &&
                (arg_ptr < &new_argv[MAX_ARGS]) )
            if ( **arg_ptr != '\0' )
                ++arg_ptr;
        path = new_argv[0];

        /* If it's an external command, fork and exec */
        if ( (path != NULL) && (strcmp(path, "exit") != 0) )
        {
            /* Create child process */
            if ( fork() == 0 )
            {
                /* If child, run command */
                execvp(path, new_argv);

                /* Error: execvp() should never return: kill child */
                fprintf(stderr, "Could not execute %s: %s\n",
                        command, errno);
                return EX_SOFTWARE;
            }
            else
            {
                /* If parent, wait for child */
                wait(&status);
            }
        }
    }   while ( strcmp(command, "exit") != 0 );

    return EX_OK;
}
```

### 27.1.5 Addendum: The POSIX Spawn Interface

The POSIX spawn interface provides a higher level interface for creating child processes than `fork()` and `exec()`.

The `posix_spawn()` and `posix_spawnp()` functions create a new process and exec a new program in one function call, while also accepting two structure pointers containing a wealth of information about how to initialize and modify the new process. Both of these structure pointers can be `NULL` for the most basic use. Below is a modification of the **skel-shell** program using basic `posix_spawnp()`.

```c
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
#include <sysexits.h>
#include <spawn.h>

#define COMMAND_LEN 1024
#define MAX_ARGS    1024

int     main(int argc, char *argv[], char *envp[])

{
    char    command[COMMAND_LEN+1], *command_ptr,
            *new_argv[MAX_ARGS], **arg_ptr, *path;
    int     status;
    pid_t   child_pid;

    /* Input commands until "exit" or Ctrl+d is entered */
    do
    {
        fputs("skel-shell: ", stdout);
        if ( fgets(command,COMMAND_LEN,stdin) == NULL )
            return EX_OK;   // Ctrl+d = EOF

        /* Set up argv array for execvp() */
        arg_ptr = new_argv, command_ptr = command;
        while ( ((*arg_ptr = strsep(&command_ptr, " \t\n")) != NULL) &&
                (arg_ptr < &new_argv[MAX_ARGS]) )
            if ( **arg_ptr != '\0' )
                ++arg_ptr;
        path = new_argv[0];

        /* If it's an external command, fork and exec */
        if ( (path != NULL) && (strcmp(path, "exit") != 0) )
        {
            /* Create child process */
            if ( posix_spawnp(&child_pid, path, NULL, NULL, new_argv, envp) != 0 )
            {
                /* Error: execvp() should never return: kill child */
                fprintf(stderr, "Could not execute %s: %s\n",
                        command, errno);
                return EX_SOFTWARE;
            }
            else
            {
                /* If parent, wait for child */
                wait(&status);
            }
        }
    }   while ( strcmp(command, "exit") != 0 );

    return EX_OK;
}
```

### 27.1.6 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Describe one pro and one con of using the `system()` function?
2. How does a program know if it is running the parent process or the child process after calling `fork()`?
3. What do the `exec` family of functions return upon successful completion?
4. How does a shell program know when the foreground command we ran under it is finished?
5. What is the advantage of the POSIX spawn interface over `fork()` and `exec()`?

## 27.2 Redirection

### 27.2.1 Simple Redirection

Redirection, as performed by the shell using the < and > operators in a Unix command, involves closing one of the standard file descriptors (0 = standard input, 1 = standard output, 2 = standard error) and then simply opening another file or device using `open()`.

This depends on the fact that the `open()` function always chooses the lowest available file descriptor. If descriptors 0, 1, and 2 are already open (as is usually the case), and no other files are open, then `open()` will return 3. If we close descriptor 0 (standard input), then `open()` will attach the next file opened to descriptor 0.

```
if ( fork() == 0 )
{
    // This clause is only run by the child process
    char    *new_argv[] = "ls", "-als", "/etc", NULL },
            *output_file = "output.txt";

    // Redirect standard output to output.txt
    close(1);
    if ( open(output_file, O_WRONLY|O_CREAT) == -1 )
    {
        fprintf(stderr, "Could not open %s: %s\n",
                output_file, strerror(errno));
        exit(EX_CANTCREAT);
    }

    execvp("ls", new_argv);

    // If we're running this, the exec failed
    fprintf(stderr, "exec failed: %s\n", strerror(errno));
    exit(EX_OSERR);
}
else
{
    // This clause is only run by the parent process
    int     status;

    pid = wait(&status);

    printf("Child process %d returned %d.\n", pid, status);
}
```

### 27.2.2 Redirection and Restoration

Skip for now. See the book.

### 27.2.3 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Briefly describe the process of redirecting the standard input of a child process.

# Chapter 28

# Interprocess Communication (IPC)

One of the most powerful features of multitasking operating systems is the ability of processes to communicate with each other. There are numerous ways for processes to send information to other processes, in addition to passing arguments via `exec` functions. The most common ones are covered in the sections below.

## 28.1 The Environment

We have already seen in Section 4.10 that child processes inherit all the environment variables of their parent. This is a very simple form of IPC that a process can use to send a message of any kind to its children. It only works in one direction, however. Children cannot talk back to their parents via the environment.

```
    // This will be inherited by all child processes
    setenv("EDITOR", "ape", 1);
```

### 28.1.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What is the major limitation of the environment as an IPC mechanism?

## 28.2 Signals

As we saw in Section 4.9, we can terminate the foreground process running under a Unix shell by typing "Ctrl+c", or suspend it by typing "Ctrl+z". Recall from Chapter 4 that the foreground process is the one receiving input from the keyboard. There may be other "background" processes running under the same shell, but they do not receive input from the keyboard.

How do Ctrl+c and Ctrl+z work? Signals are simple integer values that any Unix process can send to any other Unix process, as long as it knows their process ID, or PID. We can see a list of running processes from the Unix shell by running **ps**. The PID is the number in the first column.

```
shell-prompt: ps
  PID TT  STAT       TIME COMMAND
49557  0  Is      0:00.13 /bin/tcsh
50145  0  S+      0:00.27 ape ipc.dbk
41704  2- I      19:30.22 /usr/local/lib/virtualbox/VBoxHeadless --startvm Alma
 6050  1- S    2269:35.77 /usr/local/lib/virtualbox/VBoxHeadless --startvm NetB
50012  3  Is+     0:00.07 /bin/tcsh
```

```
50421  4  Ss      0:00.06 /bin/tcsh
50446  4  R+      0:00.00 ps
```

When we type Ctrl+c, the shell process normally sends a SIGINT signal (signal 2) to the foreground process. When we type Ctrl+z, it sends a SIGTSTP (signal 18).

We could accomplish the same thing using the **kill** command, which is confusingly named, because it can send any signal, most of which do not terminate the process. For example, to send a SIGINT signal to the **ape** process shown above, we could run:

```
shell-prompt: kill -INT 50145
```

To suspend it, we could run:

```
shell-prompt: kill -TSTP 50145
```

There is also a `kill()` function in the standard library, so we can achieve the same effect from within a C program directly (rather than clumsily using `system()` to run a shell command).

```
    system("kill -INT 50145);   // A clumsy approach

    kill(50145, SIGINT);        // A cleaner approach
```

There are many different signal types with many different effects on the target process. Run **man signal** for a summary.

C programs can choose how to respond to all signals except signal 9 (SIGKILL), which always terminates a process immediately. (This is also what a "Force quit" does on macOS).

To alter the default response to a signal, C programs can call `signal()` or the more flexible `sigaction()` to specify a function to run in response to a given signal. Such a function is called a *signal handler*.

```
#include <stdio.h>
#include <signal.h>
#include <sysexits.h>

void    catch_sigint(void);

int     main(int argc,char *argv[])

{
    struct sigaction new, old;
    int     ch;

    /* Install new signal handler */
    new.sa_handler = (void (*)())catch_sigint;
    new.sa_flags = 0;
    sigaction(SIGINT,&new,&old);
    while ( (ch = getchar()) != 'q' )
        printf("%d\n",ch);

    /* Restore old action, just for demonstration */
    sigaction(SIGINT,&old,NULL);

    return EX_OK;
}


void    catch_sigint()

{
    puts("Caught a SIGINT signal!");
}
```

### 28.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What Unix command do we use to send a SIGCONT signal (which tells a stopped process to resume running) to a process. What C function?

   If you were grading a program where the student was tasked with writing a function that sends any signal to any process, how many points would you award for descriptive naming to a student who gave the function this name?

2. What is a signal handler? How do we use one?

## 28.3 Pipes

We saw how to use pipes from the Unix shell in Section 4.8.3. Like most things in the Unix shell, this feature is built from C and C library functions.

The standard library provides a very convenient way to run another process from within a C program and pipe output to it or pipe input from it. The function looks somewhat like a mutant chimera of `system()` and `fopen()`, from a scientific experiment gone wrong in an X-files episode.

The `popen()` function looks like `fopen()`, but takes a *shell command* like `system()` instead of a filename. If the open mode is "r", then the standard output from the child process is redirected to the input file stream created. If the open mode is "w", then then output file stream created is sent to the standard input of the child process.

**Note** We must use `pclose()`, not `fclose()`, to close a FILE stream opened with `popen()`.

```
#include <stdio.h>
#include <string.h>
#include <sysexits.h>

#define MAX_LIST_SIZE       1000
#define MAX_FILENAME_LEN    128

int     main()

{
    FILE    *infile;
    char    *list[MAX_LIST_SIZE],
            file[MAX_FILENAME_LEN+1];
    size_t  list_size = 0;

    infile = popen("ls","r");
    if ( infile != NULL )
    {
        while ( (list_size < MAX_LIST_SIZE) &&
                (fscanf(infile,"%s",file) == 1) )
        {
            list[list_size++] = strdup(file);
            puts(file);
        }
    }
    pclose(infile);

    return EX_OK;
}
```

A more general approach to creating a pipe is used in conjunction with `fork()` and `exec()`. The `pipe()` function opens *two* file descriptors, placed in the array of two integers passed as an argument. We create the pipe *before* calling `fork()`. The child process inherits all file descriptors from the parent, so that both parent and child can send and receive messages through the pipe.

This allows for bidirectional communication between the parent and child process. Each of them reads from `fd[0]` and writes to `fd[1]`. I.e., `fd[0]` is the read end of the pipe and `fd[1]` is the write end. If the pipe is only to be used for one-way communication, each process simply closes the descriptor it doesn't need.

There is one pipe, which both processes can read and write. It is possible for a process to end up talking to itself via this pipe. It is up to us to ensure that anything written to the pipe is not read back by the same process.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sysexits.h>

#define BUFF_SIZE   40

int     main(int argc,char *argv[])

{
    int     fd[2];
    char    buff[BUFF_SIZE+1];

    if ( pipe(fd) == 0 )
    {
        if ( fork() == 0 )
        {
            /* Child: writer */
            close(fd[0]);

            /* Get a message to send */
            fputs("Enter a message to send: ",stdout);
            fgets(buff,BUFF_SIZE,stdin);

            /* Send message, including nul */
            write(fd[1],buff,strlen(buff));
            close(fd[1]);
        }
        else
        {
            /* Parent: reader */
            close(fd[1]);

            /* Read message from parent */
            read(fd[0],buff,BUFF_SIZE);
            printf("Got the message: %s.",buff);
            close(fd[0]);
        }
    }

    return EX_OK;
}
```

### 28.3.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Show how to open a FILE stream that it piped to the standard input of a new process running the **more** command.

2. Where do we use the `pipe()` function in order to set up communication between a parent and child process?

## 28.4 Sockets

Sockets are a communication construct to which processes can "plug in". They are similar to pipes, and in fact pipes can be implemented as sockets under the hood. Sockets, however, are lower-level and more flexible than simple pipes.

One major difference is that sockets can be used by processes *on different computers* to communicate across a network. All other IPC methods discussed here only work for processes running on the same computer.

Sockets are responsible for virtually all communication over the Internet (or local networks based on Ethernet and similar network technologies).

Socket programming is complex, so there may not be time to get into the details in a typical undergraduate course. Knowing the concepts may have to suffice.

Below is a simple example program like the pipes example above, but using its own function to implement the pipe using "Unix" sockets to establish communication between processes on the same computer. Other types of sockets are needed for network communication, along with access to two machines where we can run the processes that would talk to each other.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <errno.h>
#include <sysexits.h>

#define BUFF_SIZE   40
#define BACKLOG     1       // Max queue length for socket

int     home_made_pipe(int fd[]);

int     main(int argc, char *argv[])

{
    int     fd[2];
    char    buff[BUFF_SIZE + 1];

    if ( home_made_pipe(fd) == EX_OK )
    {
        if ( fork() == 0 )
        {
            // Child: writer, don't need the read descriptor
            close(fd[0]);

            // Get a message to send
            fputs("Enter a message to send: ", stdout);
            fgets(buff, BUFF_SIZE, stdin);

            // Send message, including nul byte
            write(fd[1], buff, strlen(buff) + 1);
            close(fd[1]);
        }
        else
        {
            // Parent: reader, don't need the write descriptor
            close(fd[1]);

            // Read message from parent
            read(fd[0], buff, BUFF_SIZE);
```

```
            printf("Got the message: %s", buff);
            close(fd[0]);
        }
    }
    else
        perror("home_made_pipe() failed");

    return EX_OK;
}


/*
 * Home-made pipe function with informative output.
 * This function creates a pair of connected sockets
 * for local interprocess communication.
 */

int     home_made_pipe(int fd[2])

{
    // Using named initializers for sockaddr structure, since it may differ
    // across platforms
    struct sockaddr name0 = {.sa_family = AF_UNIX, .sa_data = "Barny"};
    struct sockaddr name1 = {.sa_family = AF_UNIX, .sa_data = "Fred"};
    struct sockaddr actual_name = {.sa_family = AF_UNIX, .sa_data = "Nobody"};

    // Socket file descriptors
    int     s0, s1, connected_socket;

    // At least as long as the strings above, may be altered by getcoskname()
    socklen_t   namelen0 = 14, namelen1 = 14, actual_namelen = 14;

    // Create a pair of sockets
    if ( (s0 = socket(AF_UNIX, SOCK_STREAM, 0)) == -1 )
    {
        fprintf(stderr, "Failed to create socket s0: %s.\n", strerror(errno));
        return -1;
    }

    if ( (s1 = socket(AF_UNIX, SOCK_STREAM, 0)) == -1 )
    {
        fprintf(stderr, "Failed to create socket s1: %s.\n", strerror(errno));
        return -1;
    }

    // Name the sockets
    if ( bind(s0, &name0, namelen0) == -1 )
    {
        fprintf(stderr, "Failed to bind name0: %s.\n", strerror(errno));
        return -1;
    }

    if ( bind(s1, &name1, namelen1) == -1 )
    {
        fprintf(stderr, "Failed to bind name1: %s.\n", strerror(errno));
        return -1;
    }

    // Debug code
    if ( getsockname(s1, &actual_name, &actual_namelen) == -1 )
    {
        fprintf(stderr, "Failed to get socketname of s1: %s\n", strerror(errno));
```

```
        return -1;
    }
    fprintf(stderr, "Socket name = %s, actual namelen = %d.\n",
            actual_name.sa_data, actual_namelen);

    // Set socket s1 to listen for connect requests
    if ( listen(s1, BACKLOG) == -1 )
    {
        perror("listen() failed");
        return -1;
    }

    // Send connection request from s0 to s1
    if ( connect(s0, &name1, namelen1) == -1 )
    {
        perror("connect() failed");
        return -1;
    }

    // Accept first connection request sent to s1
    if ( (connected_socket = accept(s1, &actual_name, &actual_namelen)) == -1 )
    {
        perror("accept() failed.");
        return -1;
    }
    else
    {
        printf("Accepted connection from %s.\n",
                actual_name.sa_data);
        close(s1);
    }

    // Return socket pair through fd[] argument
    fd[0] = s0;
    fd[1] = connected_socket;

    // Must remove names before next listen()
    unlink("Barny");
    unlink("Fred");

    return EX_OK;
}
```

### 28.4.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. What is the major advantage of sockets over other IPC mechanisms?
2. What is the major disadvantage of sockets when compared with other IPC mechanisms?

## 28.5 Shared Memory

Shared memory allows two processes running on the same computer to directly access the same memory (i.e. variables, or objects).

Shared memory can make communication between processes faster and easier, since it does not involve system calls that make processes wait for input or output, often giving up their time slot on the CPU and going back to the end of the CPU queue.

Instead, one process writes to a memory address, and other processes can read it whenever they want.

---

**Note** Regardless of how communication occurs between processes, process synchronization is always a potential issue. This is inherent in the parallel processing algorithms and has nothing to do with how communication is implemented. While shared memory eliminates some delays, processes reading from a shared memory object still must ensure that it is up-to-date. The details of this are not covered here, but saved for a course in parallel/concurrent programming.

---

### 28.5.1 System V Shared Memory

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <sysexits.h>

#define BLOCK_SIZE  40

int     main(int argc,char *argv[])

{
    static char    *buff;
    int     shmid,
            mode = SHM_R|SHM_W,
            status;

    /* Create shared memory object */
    if ( (shmid=shmget(IPC_PRIVATE,
                       BLOCK_SIZE,mode)) == -1 )
        perror("shmget() failed");

    /* Map object into this process' memory space */
    if ( (void *)(buff = shmat(shmid,0,0)) == (void *)-1 )
        perror("shmat() failed");

    /* Create second process to share memory with */
    if ( fork() == 0 )
    {
        /* Allow writer time to write message */
        sleep(1);

        /* Print message stored in shared memory */
        printf("Reader read %s from shared memory.\n",buff);

        /* Detach shared memory from process */
        shmdt(buff);

        /* Delete shared memory object from system */
        shmctl(shmid,IPC_RMID,NULL);
    }
    else
    {
        char    *message = "Hello, clone!";
```

```
        /* Write message to shared memory, including '\0' terminator */
        memcpy(buff, message, strlen(message) + 1);
        printf("Writer wrote %s to shared memory.\n",buff);

        /* Detach shared memory from process */
        shmdt(buff);

        wait(&status);
    }

    return EX_OK;
}
```

### 28.5.2 Process Synchronization: Semaphores

Skip for now, see book.

### 28.5.3 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. How do processes communicate using shared memory?
2. Does shared memory eliminate the need for process synchronization?

# Chapter 29

# Threads

Prerequisite: Unix processes (Chapter 27) and shared memory (Section 28.5).

## 29.1 Overview

*Threads*, also known as *lightweight processes*, are a feature of modern Unix systems aimed at better utilizing multiple cores, or achieving better utilization of a single core with processes that would not fully utilize the core individually.

Threads differ from *heavyweight* processes (those created using `fork()` or `posix_spawn()`) in a number of ways, including, but not limited to the following:

- The creation of heavyweight processes involves significant overhead cost, while threads can be created very quickly.
- Heavyweight processes usually run different programs, while threads usually run the same program on different data. The text segment (machine code) of a program is typically shared by multiple threads, while each thread has its own data, stack, and heap segments. ( See Table 11.3 to review. )

  Both heavyweight processes and threads are created by cloning a process. Threads, however, don't typically exec a different program. The multiple threads generally continue to run the same program, sharing some variables among all threads, and having private copies of others for each thread.
- Programs often use different numbers of threads for different parts of the program. E.g., multiple threads may be used to speed up a particular loop, while the rest of the program runs in a single thread.
- Heavyweight processes cooperating with each other may run on the same machine or on different machines, and communicate using a variety of different mechanisms such as environment variables, signals, pipes, sockets, or shared memory. The *MPI (Message Passing Interface)* is a system based on sockets, often used by heavyweight processes cooperating as a parallel job across multiple machines on a network.

  Threads that are part of the same job normally run on the same machine and communicate entirely via shared memory.

Threads are often used to speed up scientific programs that need to process large amounts of data. If the data can be subdivided into segments that are processed independently, threads are an easy way to utilize multiple cores to speed up processing.

The FreeBSD NFS (Network File System) server uses multiple threads to listen for and process requests from client machines. This improves throughput by allowing other threads to immediately process requests rather than let them wait for a single process to finish one request before responding to the next. The server automatically spawns additional threads when the number of requests increases. Some other POSIX systems use a fixed number of heavyweight processes to service NFS clients.

### 29.1.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. How do threads differ from heavyweight processes?
2. What are some examples how how threads can be used? Cite examples from the text and some of your own if possible.

## 29.2 Addendum: POSIX Threads

*POSIX threads (pthreads)* is one of the first standardized threads interfaces and is still widespread today. Countless programs written since the 1990s use pthreads. It uses highly standardized library functions to create and destroy threads.

However, for most new code, POSIX threads have given way to simpler interfaces such as *OpenMP* (Open MultiProcessing).

We will leave the reader to investigate POSIX threads on their own and focus on OpenMP in this text.

### 29.2.1 Practice

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

1. Why are POSIX threads important to know about?

## 29.3 Addendum: OpenMP

The *OpenMP* (Open MultiProcessing) system is a feature of modern compilers that facilitates creation and coordination of threads in C, C++, Fortran, and some other languages.

In C and C++, we use the `#pragma` preprocessor directive to create threads.

```
/*****************************************************************************
 *  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;
}
```

To use OpenMP, we must include `omp.h` in the code and compile with `-fopenmp`.

```
shell-prompt: cc -O -Wall -fopenmp myprog.c -o myprog
```

One of the most convenient features is the ability to parallelize a loop with no modification to the code besides the pragma. If each iteration of a loop is independent of other iterations, then OpenMP will execute multiple iterations at the same time on different cores. For example, if the computer running the program below has 4 cores, then iterations of the loop where c is 0, 1, 2, and 3 can all run at the same time.

```
/*****************************************************************************
 *  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;
}
```

The OpenMP system has rules for automatically determining whether a variable should be shared among threads, or duplicated so that each thread has a private copy that can contain a different value than other threads. Generally, a variable defined outside a parallel region is shared by default, and one defined inside a parallel region is private by default. We can explicitly control the nature each each variable using additional parameters in the pragma directive.

OpenMP also allows us to tag code as *atomic*, meaning that only one thread can be running any part of the code at a given time. This is important to ensure the the code produces correct results. It may seem unnecessary below since `sum += c_squared;` is a single statement. In reality, however, this statement translates to multiple machine instructions, and if one thread starts the sequence before another finishes it, the results may be incorrect. Such a sequence of machine instructions is called a *critical section*.

```
/*****************************************************************************
 *  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;
}
```

The OpenMP system uses all available cores by default, but we can control the number of threads using the `OMP_NUM_THREADS` environment variable.

```
shell-prompt: env OMP_NUM_THREADS=4 ./myprog
```

### 29.3.1 Practice

---

**Note** Be sure to thoroughly review the instructions in Section 0.2.3 before doing the practice problems below.

---

1. What criteria must be satisfied in order to use an OpenMP parallel `for` loop?
2. How do we use OpenMP in a C program?
3. Write a C program that prints the squares of all integers from 1 to 100, using OpenMP to utilize all available cores on the computer. Do you notice anything odd about the output?

# Chapter 30

# Unix Graphics: X Windows

## 30.1 How X11 Works

Inherently networked

### 30.1.1 The X Server

### 30.1.2 X Clients

### 30.1.3 Addendum: DRI

## 30.2 Programming with Xlib

## 30.3 Programming with the Xt Toolkit

## 30.4 Addendum: OpenGL 3D Graphics

## 30.5 Addendum: QT, GTK

# Part IV

# The C++ Programming Language

# Chapter 31

# Introduction to C++

# Chapter 32

# Index

**A**
alpha testing, 36

**B**
beta testing, 36

**S**
software life cycle, 35