ZED SHAW'S HARD WAY SERIES

Learn C the HARD WAY

Practical Exercises on the Computational Subjects You Keep Avoiding (Like C)

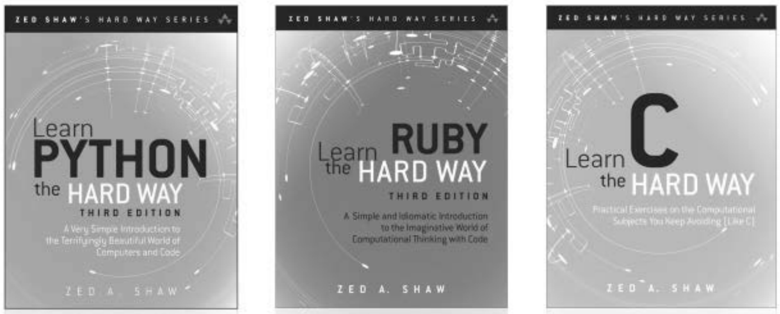
ZED A. SHAW

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

LEARN C
THE HARD WAY

Zed Shaw's Hard Way Series


ZED SHAW'S HARD WAY SERIES
Learn PYTHON the HARD WAY
THIRD EDITION
A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code
ZED A. SHAW
Learn RUBY the HARD WAY
THIRD EDITION
A Simple and Idiomatic Introduction to the Imaginative World of Computational Thinking with Code
ZED A. SHAW
Learn C the HARD WAY
Practical Exercises on the Computational Subjects You Keep Avoiding (Like C)
ZED A. SHAW

Addison-Wesley

Visit **informit.com/hardway** for a complete list of available publications.

Zed Shaw's Hard Way Series emphasizes instruction and *making* things as the best way to get started in many computer science topics. Each book in the series is designed around short, understandable exercises that take you through a course of instruction that creates working software. All exercises are thoroughly tested to verify they work with real students, thus increasing your chance of success. The accompanying video walks you through the code in each exercise. Zed adds a bit of humor and inside jokes to make you laugh while you're learning.

Make sure to connect with us!
informit.com/socialconnect

informIT.com
the trusted technology learning source
| Addison-Wesley | Safari

ALWAYS LEARNING PEARSON

LEARN C THE HARD WAY

Practical Exercises on the
Computational Subjects You Keep
Avoiding (Like C)

Zed A. Shaw

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Shaw, Zed, author.
  Learn C the hard way : practical exercises on the computational subjects you keep avoiding (like C) /
Zed A. Shaw.
    pages   cm
  Includes index.
  ISBN 978-0-321-88492-3 (pbk. : alk. paper)—ISBN 0-321-88492-2 (pbk. : alk. paper)
  1. C (Computer program language)—Problems, exercises, etc. I. Title.
  QA76.73.C15S473 2016
  005.13'3—dc23
                                                                                    2015020858

Copyright © 2016 Zed A. Shaw

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 200 Old Tappan Road, Old Tappan, New Jersey 07657, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-88492-3
ISBN-10: 0-321-88492-2

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, August 2015

Contents

Acknowledgments xiv

This Book Is Not Really about C xv

- The Undefined Behaviorists xvi
- C Is a Pretty and Ugly Language xvii
- What You Will Learn xviii
- How to Read This Book xviii
- The Videos xix
- The Core Competencies xix
  - Reading and Writing xix
  - Attention to Detail xx
  - Spotting Differences xx
  - Planning and Debugging xx

Exercise 0 The Setup 2

- Linux 2
- Mac OS X 2
- Windows 3
- Text Editor 3
  - Do Not Use an IDE 4

Exercise 1 Dust Off That Compiler 6

- Breaking It Down 6
- What You Should See 7
- How to Break It 8
- Extra Credit 8

Exercise 2 Using `Makefiles` to Build 10

- Using `Make` 10
- What You Should See 11
- How to Break It 12
- Extra Credit 12

Exercise 3 Formatted Printing 14

- What You Should See 15
- External Research 15

How to Break It . . . 15
Extra Credit . . . 16
Exercise 4 Using a Debugger . . . 18
GDB Tricks . . . 18
GDB Quick Reference . . . 18
LLDB Quick Reference . . . 19
Exercise 5 Memorizing C Operators . . . 20
How to Memorize . . . 20
The List of Operators . . . 21
Exercise 6 Memorizing C Syntax . . . 26
The Keywords . . . 26
Syntax Structures . . . 27
A Word of Encouragement . . . 30
A Word of Warning . . . 31
Exercise 7 Variables and Types . . . 32
What You Should See . . . 34
How to Break It . . . 34
Extra Credit . . . 34
Exercise 8 `If`, `Else-If`, `Else` . . . 36
What You Should See . . . 37
How to Break It . . . 37
Extra Credit . . . 38
Exercise 9 `While`-Loop and Boolean Expressions . . . 40
What You Should See . . . 40
How to Break It . . . 41
Extra Credit . . . 41
Exercise 10 Switch Statements . . . 42
What You Should See . . . 43
How to Break It . . . 44
Extra Credit . . . 44
Exercise 11 Arrays and Strings . . . 46
What You Should See . . . 47
How to Break It . . . 48
Extra Credit . . . 48
Exercise 12 Sizes and Arrays . . . 50
What You Should See . . . 51

How to Break It . . . 52
Extra Credit . . . 53
Exercise 13 `For`-Loops and Arrays of Strings . . . 54
What You Should See . . . 56
Understanding Arrays of Strings . . . 56
How to Break It . . . 57
Extra Credit . . . 57
Exercise 14 Writing and Using Functions . . . 58
What You Should See . . . 59
How to Break It . . . 60
Extra Credit . . . 60
Exercise 15 Pointers, Dreaded Pointers . . . 62
What You Should See . . . 64
Explaining Pointers . . . 65
Practical Pointer Usage . . . 66
The Pointer Lexicon . . . 66
Pointers Aren't Arrays . . . 67
How to Break It . . . 67
Extra Credit . . . 67
Exercise 16 Structs and Pointers to Them . . . 68
What You Should See . . . 71
Explaining Structures . . . 71
How to Break It . . . 72
Extra Credit . . . 72
Exercise 17 Heap and Stack Memory Allocation . . . 74
What You Should See . . . 79
Heap versus Stack Allocation . . . 80
How to Break It . . . 81
Extra Credit . . . 82
Exercise 18 Pointers to Functions . . . 84
What You Should See . . . 88
How to Break It . . . 88
Extra Credit . . . 89
Exercise 19 Zed's Awesome Debug Macros . . . 90
The C Error-Handling Problem . . . 90
The Debug Macros . . . 91
Using `dbg.h` . . . 93

What You Should See 95
How the CPP Expands Macros 96
Extra Credit 98
Exercise 20 Advanced Debugging Techniques 100
Debug Printing versus GDB 100
A Debugging Strategy 101
Extra Credit 102
Exercise 21 Advanced Data Types and Flow Control 104
Available Data Types 104
Type Modifiers 104
Type Qualifiers 105
Type Conversion 105
Type Sizes 106
Available Operators 108
Math Operators 108
Data Operators 109
Logic Operators 109
Bit Operators 109
Boolean Operators 110
Assignment Operators 110
Available Control Structures 110
Extra Credit 111
Exercise 22 The Stack, Scope, and Globals 112
`ex22.h` and `ex22.c` 112
`ex22_main.c` 114
What You Should See 117
Scope, Stack, and Bugs 118
How to Break It 119
Extra Credit 119
Exercise 23 Meet Duff's Device 120
What You Should See 124
Solving the Puzzle 124
Why Bother? 124
Extra Credit 125

Exercise 24 Input, Output, Files 126
What You Should See 128
How to Break It 129
The I/O Functions 129
Extra Credit 130
Exercise 25 Variable Argument Functions 132
What You Should See 135
How to Break It 136
Extra Credit 136
Exercise 26 Project `logfind` 138
The `logfind` Specification 138
Exercise 27 Creative and Defensive Programming 140
The Creative Programmer Mind-Set 140
The Defensive Programmer Mind-Set 141
The Eight Defensive Programmer Strategies 141
Applying the Eight Strategies 142
Never Trust Input 142
Prevent Errors 145
Fail Early and Openly 146
Document Assumptions 147
Prevention over Documentation 147
Automate Everything 148
Simplify and Clarify 148
Question Authority 149
Order Is Not Important 149
Extra Credit 150
Exercise 28 Intermediate `Makefile`s 152
The Basic Project Structure 152
`Makefile` 153
The Header 154
The Target Build 155
The Unit Tests 156
The Cleaner 157
The Install 158
The Checker 158

What You Should See 159
Extra Credit 159
Exercise 29 Libraries and Linking 160
Dynamically Loading a Shared Library 161
What You Should See 163
How to Break It 164
Extra Credit 164
Exercise 30 Automated Testing 166
Wiring Up the Test Framework 167
Extra Credit 171
Exercise 31 Common Undefined Behavior 172
UB 20 173
Common UBs 173
Exercise 32 Double Linked Lists 174
What Are Data Structures 178
Making the Library 178
Doubly Linked Lists 179
Definition 180
Implementation 181
Tests 185
What You Should See 187
How to Improve It 188
Extra Credit 188
Exercise 33 Linked List Algorithms 190
Bubble and Merge Sort 190
The Unit Test 191
The Implementation 193
What You Should See 195
How to Improve It 196
Extra Credit 197
Exercise 34 Dynamic Array 198
Advantages and Disadvantages 205
How to Improve It 206
Extra Credit 206

Exercise 35 Sorting and Searching 208
Radix Sort and Binary Search 211
C Unions 212
The Implementation 213
`RadixMap_find` and Binary Search 219
`RadixMap_sort` and `radix_sort` 220
How to Improve It 221
Extra Credit 222
Exercise 36 Safer Strings 224
Why C Strings Were a Horrible Idea 224
Using `bstrlib` 225
Learning the Library 226
Exercise 37 Hashmaps 228
The Unit Test 235
How to Improve It 238
Extra Credit 238
Exercise 38 Hashmap Algorithms 240
What You Should See 245
How to Break It 246
Extra Credit 247
Exercise 39 String Algorithms 248
What You Should See 255
Analyzing the Results 257
Extra Credit 258
Exercise 40 Binary Search Trees 260
How to Improve It 273
Extra Credit 273
Exercise 41 Project devpkg 274
What Is devpkg? 274
What We Want to Make 274
The Design 275
The Apache Portable Runtime 275
Project Layout 277
Other Dependencies 277

The `Makefile` . . . 277
The Source Files . . . 278
The DB Functions . . . 279
The Shell Functions . . . 283
The Command Functions . . . 287
The devpkg Main Function . . . 292
The Final Challenge . . . 295
Exercise 42 Stacks and Queues . . . 296
What You Should See . . . 299
How to Improve It . . . 299
Extra Credit . . . 299
Exercise 43 A Simple Statistics Engine . . . 300
Rolling Standard Deviation and Mean . . . 300
Implemention . . . 301
How to Use It . . . 306
Extra Credit . . . 307
Exercise 44 Ring Buffer . . . 310
The Unit Test . . . 313
What You Should See . . . 313
How to Improve It . . . 314
Extra Credit . . . 314
Exercise 45 A Simple TCP/IP Client . . . 316
Augment the `Makefile` . . . 316
The `netclient` Code . . . 316
What You Should See . . . 320
How to Break It . . . 320
Extra Credit . . . 321
Exercise 46 Ternary Search Tree . . . 322
Advantages and Disadvantages . . . 330
How to Improve It . . . 331
Extra Credit . . . 331
Exercise 47 A Fast URL Router . . . 332
What You Should See . . . 335
How to Improve It . . . 335
Extra Credit . . . 336

Exercise 48 A Simple Network Server 338

The Specification 338

Exercise 49 A Statistics Server 340

Specification 340

Exercise 50 Routing the Statistics 342

Exercise 51 Storing the Statistics 344

The Specification 344

Exercise 52 Hacking and Improving Your Server 346

Next Steps 348

Index 349

Acknowledgments

I would like to thank three kinds of people who helped make this book what it is today: the haters, the helpers, and the painters.

The haters helped make this book stronger and more solid through their inflexibility of mind, irrational hero worship of old C gods, and complete lack of pedagogical expertise. Without their shining example of what not to be, I would have never worked so hard to make this book a complete introduction to becoming a better programmer.

The helpers are Debra Williams Cauley, Vicki Rowland, Elizabeth Ryan, the whole team at Addison-Wesley, and everyone online who sent in fixes and suggestions. Their work producing, fixing, editing, and improving this book has formed it into a more professional and better piece of writing.

The painters, `Brian`, `Arthur`, `Vesta`, and `Sarah`, helped me find a new way to express myself and to distract me from deadlines that Deb and Vicki clearly set for me but that I kept missing. Without painting and the gift of art these artists gave me, I would have a less meaningful and rich life.

Thank you to all of you for helping me write this book. It may not be perfect, because no book is perfect, but it's at least as good as I can possibly make it.

This Book Is Not Really about C

Please don't feel cheated, but this book is not about teaching you C programming. You'll learn to write programs in C, but the most important lesson you'll get from this book is *rigorous defensive programming*. Today, too many programmers simply assume that what they write works, but one day it will fail catastrophically. This is especially true if you're the kind of person who has learned mostly modern languages that solve many problems for you. By reading this book and following my exercises, you'll learn how to create software that defends itself from malicious activity and defects.

I'm using C for a very specific reason: C is broken. It is full of design choices that made sense in the 1970s but make zero sense now. Everything from its unrestricted, wild use of pointers to its severely broken NUL terminated strings are to blame for nearly all of the security defects that hit C. It's my belief that C is so broken that, while it's in wide use, it's the most difficult language to write securely. I would fathom that Assembly is actually easier to write securely than C. To be honest, and you'll find out that I'm very honest, I don't think that anybody should be writing new C code.

If that's the case, then why am I teaching you C? Because I want you to become a better, stronger programmer, and there are two reasons why C is an excellent language to learn if you want to get better. First, C's lack of nearly every modern safety feature means you have to be more vigilant and more aware of what's going on. If you can write secure, solid C code, you can write secure, solid code in any programming language. The techniques you learn will translate to every language you use from now on. Second, learning C gives you direct access to a mountain of legacy code, and teaches you the base syntax of a large number of descendant languages. Once you learn C, you can more easily learn C++, Java, Objective-C, and JavaScript, and even other languages become easier to learn.

I don't want to scare you away by telling you this, because I plan to make this book incredibly fun, easy, and devious. I'll make it fun to learn C by giving you projects that you might not have done in other programming languages. I'll make this book easy by using my proven pattern of exercises that has you *doing* C programming and building your skills slowly. I'll make it devious by teaching you how to break and then secure your code so you understand why these issues matter. You'll learn how to cause stack overflows, illegal memory access, and other common flaws that plague C programs so that you know what you're up against.

Getting through this book will be challenging, like all of my books, but when you're done you will be a far better and more confident programmer.

The Undefined Behaviorists

By the time you're done with this book, you'll be able to debug, read, and fix almost any C program you run into, and then write new, solid C code should you need to. However, I'm not really going to teach you official C. You'll learn the language, and you'll learn how to use it well, but official C isn't very secure. The vast majority of C programmers out there simply don't write solid code, and it's because of something called *Undefined Behavior* (UB). UB is a part of the American National Standards Institute (ANSI) C standard that lists all of the ways that a C compiler can disregard what you've written. There's actually a part of the standard that says if you write code like this, then all bets are off and the compiler doesn't have to do anything consistently. UB occurs when a C program reads off the end of a string, which is an incredibly common programming error in C. For a bit of background, C defines strings as blocks of memory that end in a NUL byte, or a 0 byte (to simplify the definition). Since many strings come from outside the program, it's common for a C program to receive a string without this NUL byte. When it does, the C program attempts to read past the end of this string and into the memory of the computer, causing your program to crash. Every other language developed after C attempts to prevent this, but not C. C does so little to prevent UB that every C programmer seems to think it means they don't have to deal with it. They write code full of potential NUL byte overruns, and when you point them out to these programmers, they say, "Well that's UB, and I don't have to prevent it." This reliance on C's large number of UBs is why most C code is so horribly insecure.

I write C code to try to avoid UB by either writing code that doesn't trigger it, or writing code that attempts to prevent it. This turns out to be an impossible task because there is *so much* UB that it becomes a Gordian knot of interconnected pitfalls in your C code. As you go through this book, I'll point out ways you can trigger UB, how to avoid it if you can, and how to trigger it in other people's code if possible. However, you should keep in mind that avoiding the nearly random nature of UB is almost impossible, and you'll just have to do your best.

WARNING! You'll find that hardcore C fans frequently will try to beat you up about UB. There's a class of C programmers who don't write very much C code but have memorized all of the UB just so they could beat up a beginner intellectually. If you run into one of these abusive programmers, please ignore them. Often, they aren't practicing C programmers, they are arrogant, abusive, and will only end up asking you endless questions in an attempt to prove their superiority rather than helping you with your code. Should you ever need help with your C code, simply email me at help@learncodethehardway.org, and I will gladly help you.

C Is a Pretty and Ugly Language

The presence of UB though is one more reason why learning C is a good move if you want to be a better programmer. If you can write good, solid C code in the way I teach you, then you can survive *any* language. On the positive side, C is a really elegant language in many ways. Its syntax is actually incredibly small given the power it has. There's a reason why so many other languages have copied its syntax over the last 45 or so years. C also gives you quite a lot using very little technology. When you're done learning C, you'll have an appreciation for a something that is very elegant and beautiful but also a little ugly at the same time. C is old, so like a beautiful monument, it will look fantastic from about 20 feet away, but when you step up close, you'll see all the cracks and flaws it has.

Because of this, I'm going to teach you the most recent version of C that I can make work with recent compilers. It's a practical, straightforward, simple, yet complete subset of C that works well, works everywhere, and avoids many pitfalls. This is the C that I use to get real work done, and not the encyclopedic version of C that hardcore fans try and fail to use.

I know the C that I use is solid because I spent two decades writing clean, solid C code that powered large operations without much failure at all. My C code has probably processed trillions of transactions because it powered the operations of companies like Twitter and airbnb. It rarely failed or had security attacks against it. In the many years that my code powered the Ruby on Rails Web world, it's run beautifully and even prevented security attacks, while other Web servers fell repeatedly to the simplest of attacks.

My style of writing C code is solid, but more importantly, my mind-set when writing C is one every programmer should have. I approach C, and any programming, with the idea of preventing errors as best I can and assuming that nothing will work right. Other programmers, even supposedly good C programmers, tend to write code and assume everything will work, but rely on UB or the operating system to save them, neither of which will work as a solution. Just remember that if people try to tell you that the code I teach in this book isn't "real C." If they don't have the same track record as me, maybe you can use what I teach you to show them why their code isn't very secure.

Does that mean my code is perfect? No, not at all. This is C code. Writing perfect C code is impossible, and in fact, writing perfect code in any language is impossible. That's half the fun and frustration of programming. I could take someone else's code and tear it apart, and someone could take my code and tear it apart. All code is flawed, but the difference is that I try to assume my code is always flawed and then prevent the flaws. My gift to you, should you complete this book, is to teach you the *defensive programming* mind-set that has served me well for two decades, and has helped me make high-quality, robust software.

What You Will Learn

The purpose of this book is to get you strong enough in C that you'll be able to write your own software with it or modify someone else's C code. After this book, you should read Brian Kernighan and Dennis Ritchie's *The C Programming Language*, Second Edition (Prentice Hall, 1988), a book by the creators of the C language, also called *K&R C*. What I'll teach you is

- The basics of C syntax and idioms
- Compilation, `make` files, linkers
- Finding bugs and preventing them
- Defensive coding practices
- Breaking C code
- Writing basic UNIX systems software

By the final exercise, you will have more than enough ammunition to tackle basic systems software, libraries, and other smaller projects.

How to Read This Book

This book is intended for programmers who have learned at least one other programming language. I refer you to my book *Learn Python the Hard Way* (Addison-Wesley, 2013) if you haven't learned a programming language yet. It's meant for beginners and works very well as a first book on programming. Once you've completed *Learn Python the Hard Way*, then you can come back and start this book.

For those who've already learned to code, this book may seem strange at first. It's not like other books where you read paragraph after paragraph of prose and then type in a bit of code here and there. Instead, there are videos of lectures for each exercise, you code right away, and then I explain what you just did. This works better because it's easier for me to explain something you've already done than to speak in an abstract sense about something you aren't familiar with at all.

Because of this structure, there are a few rules that you *must* follow in this book:

- Watch the lecture video first, unless the exercise says otherwise.
- Type in all of the code. Don't copy-paste!
- Type in the code exactly as it appears, even the comments.
- Get it to run and make sure it prints the same output.
- If there are bugs, fix them.

- Do the Extra Credit, but it's all right to skip anything you can't figure out.
- Always try to figure it out first before trying to get help.

If you follow these rules, do everything in the book, and still can't code C, then you at least tried. It's not for everyone, but just trying will make you a better programmer.

The Videos

Included in this course are videos for every exercise, and in many cases, more than one video for an exercise. These videos should be considered essential to get the full impact of the book's educational method. The reason for this is that *many* of the problems with writing C code are interactive issues with failure, debugging, and commands. C requires much more interaction to get the code running and to fix problems, unlike languages like Python and Ruby where code just runs. It's also much easier to show you a video lecture on a topic, such as pointers or memory management, where I can demonstrate how the machine is actually working.

I recommend that as you go through the course, you plan to watch the videos first, and then do the exercises unless directed to do otherwise. In some of the exercises, I use one video to present a problem and then another to demonstrate the solution. In most of the other exercises, I use a video to present a lecture, and then you do the exercise and complete it to learn the topic.

The Core Competencies

I'm going to guess that you have experience using a *lesser* language. One of those *usable* languages that lets you get away with sloppy thinking and half-baked hackery like Python or Ruby. Or, maybe you use a language like LISP that pretends the computer is some purely functional fantasy land with padded walls for little babies. Maybe you've learned Prolog, and you think the entire world should just be a database where you walk around in it looking for clues. Even worse, I'm betting you've been using an integrated development environment (IDE), so your brain is riddled with memory holes, and you can't even type an entire function's name without hitting CTRL-SPACE after every three characters.

No matter what your background is, you could probably use some improvement in these areas:

Reading and Writing

This is especially true if you use an IDE, but generally I find programmers do too much skimming and have problems reading for comprehension. They'll just skim code that they need to understand in detail without taking the time to understand it. Other languages provide tools that let programmers avoid actually writing any code, so when faced with a language like C, they break down. The simplest thing to do is just understand that *everyone* has this problem, and you can fix it by forcing yourself to slow down and be meticulous about your reading and writing. At first, it'll feel painful and annoying, but take frequent breaks, and then eventually it'll be easier to do.

Attention to Detail

Everyone is bad at this, and it's the biggest cause of bad software. Other languages let you get away with not paying attention, but C demands your full attention because it's right in the machine, and the machine is very picky. With C, there is no "kind of similar" or "close enough," so you need to pay attention. Double check your work. Assume everything you write is wrong until you prove it's right.

Spotting Differences

A key problem that people who are used to other languages have is that their brains have been trained to spot differences in *that* language, not in C. When you compare code you've written to my exercise code, your eyes will jump right over characters you think don't matter or that aren't familiar. I'll be giving you strategies that force you to see your mistakes, but keep in mind that if your code is not *exactly* like the code in this book, it's wrong.

Planning and Debugging

I love other, easier languages because I can just hang out. I can type the ideas I have into their interpreter and see results immediately. They're great for just hacking out ideas, but have you noticed that if you keep doing *hack until it works*, eventually nothing works? C is harder on you because it requires you to first plan out what you want to create. Sure, you can hack for a bit, but you have to get serious much earlier in C than in other languages. I'll be teaching you ways to plan out key parts of your program before you start coding, and this will likely make you a better programmer at the same time. Even just a little planning can smooth things out down the road.

Learning C makes you a better programmer because you are forced to deal with these issues earlier and more frequently. You can't be sloppy about what you write or nothing will work. The advantage of C is that it's a simple language that you can figure out on your own, which makes it a great language for learning about the machine and getting stronger in these core programming skills.

This page intentionally left blank

EXERCISE 1

Dust Off That Compiler

After you have everything installed, you need to confirm that your compiler works. The easiest way to do that is to write a C program. Since you should already know at least one programming language, I believe you can start with a small but extensive example.

ex1.c

```
 1    #include <stdio.h>
 2
 3    /* This is a comment. */
 4    int main(int argc, char *argv[])
 5    {
 6        int distance = 100;
 7
 8        // this is also a comment
 9        printf("You are %d miles away.\n", distance);
10
11        return 0;
12    }
```

If you have problems getting the code up and running, watch the video for this exercise to see me do it first.

Breaking It Down

There are a few features of the C language in this code that you might or might not have figured out while you were typing it. I'll break this down, line by line, quickly, and then we can do exercises to understand each part better. Don't worry if you don't understand everything in this breakdown. I am simply giving you a quick dive into C and *promise* you will learn all of these concepts later in the book.

Here's a line-by-line description of the code:

ex1.c:1 An `include`, and it is the way to import the contents of one file into this source file. C has a convention of using `.h` extensions for *header* files, which contain lists of functions to use in your program.

ex1.c:3 This is a multiline `comment`, and you could put as many lines of text between the opening `/*` and closing `*/` characters as you want.

ex1.c:4 A more complex version of the `main` function you've been using so far. How C programs work is that the operating system loads your program, and then it runs the function named `main`. For the function to be totally complete it needs to return an `int` and take two parameters: an `int` for the argument count and an array of `char *` strings for the arguments. Did that just fly over your head? Don't worry, we'll cover this soon.

ex1.c:5 To start the body of any function, you write a { character that indicates the beginning of a *block*. In Python, you just did a `:` and indented. In other languages, you might have a `begin` or do word to start.

ex1.c:6 A variable declaration and assignment at the same time. This is how you create a variable, with the syntax `type name = value;`. In C, statements (except for logic) end in a `;` (semicolon) character.

ex1.c:8 Another kind of comment. It works like in Python or Ruby, where the comment starts at the // and goes until the end of the line.

ex1.c:9 A call to your old friend `printf`. Like in many languages, function calls work with the syntax `name(arg1, arg2);` and can have no arguments or any number of them. The `printf` function is actually kind of weird in that it can take multiple arguments. You'll see that later.

ex1.c:11 A return from the main function that gives the operating system (OS) your exit value. You may not be familiar with how UNIX software uses return codes, so we'll cover that as well.

ex1.c:12 Finally, we end the main function with a closing brace } character, and that's the end of the program.

There's a lot of information in this breakdown, so study it line by line and make sure you at least have a grasp of what's going on. You won't know everything, but you can probably guess before we continue.

What You Should See

You can put this into an `ex1.c` and then run the commands shown here in this sample shell output. If you're not sure how this works, watch the video that goes with this exercise to see me do it.

Exercise 1 Session

```
$ make ex1
cc -Wall -g    ex1.c   -o ex1
$ ./ex1
You are 100 miles away.
$
```

The first command `make` is a tool that knows how to build C programs (and many others). When you run it and give it ex1 you are telling `make` to look for the `ex1.c` file, run the compiler to build it, and leave the results in a file named `ex1`. This `ex1` file is an executable that you can run with `./ex1`, which outputs your results.

How to Break It

In this book, I'm going to have a small section for each program teaching you how to break the program if it's possible. I'll have you do odd things to the programs, run them in weird ways, or change code so that you can see crashes and compiler errors.

For this program, simply try removing things at random and still get it to compile. Just make a guess at what you can remove, recompile it, and then see what you get for an error.

Extra Credit

- Open the `ex1` file in your text editor and change or delete random parts. Try running it and see what happens.
- Print out five more lines of text or something more complex than "hello world."
- Run `man 3 printf` and read about this function and many others.
- For each line, write out the symbols you don't understand and see if you can guess what they mean. Write a little chart on paper with your guess so you can check it later to see if you got it right.

This page intentionally left blank

Using `Makefiles` to Build

We're going to use a program called `make` to simplify building your exercise code. The `make` program has been around for a very long time, and because of this it knows how to build quite a few types of software. In this exercise, I'll teach you just enough `Makefile` syntax to continue with the course, and then an exercise later will teach you more complete `Makefile` usage.

Using `Make`

How `make` works is you declare dependencies, and then describe how to build them or rely on the program's internal knowledge of how to build most common software. It has decades of knowledge about building a wide variety of files from other files. In the last exercise, you did this already using commands:

```
$ make ex1
# or this one too
$ CFLAGS="-Wall" make ex1
```

In the first command, you're telling `make`, "I want a file named ex1 to be created." The program then asks and does the following:

1. Does the file `ex1` exist already?
2. No. Okay, is there another file that starts with `ex1`?
3. Yes, it's called `ex1.c`. Do I know how to build `.c` files?
4. Yes, I run this command `cc ex1.c -o ex1` to build them.
5. I shall make you one `ex1` by using `cc` to build it from `ex1.c`.

The second command in the listing above is a way to pass *modifiers* to the `make` command. If you're not familiar with how the UNIX shell works, you can create these *environment variables* that will get picked up by programs you run. Sometimes you do this with a command like `export CFLAGS="-Wall"` depending on the shell you use. You can, however, also just put them before the command you want to run, and that environment variable will be set only while that command runs.

In this example, I did `CFLAGS="-Wall" make ex1` so that it would add the command line option `-Wall` to the `cc` command that `make` normally runs. That command line option tells the compiler `cc` to report all warnings (which, in a sick twist of fate, isn't actually all the warnings possible).

You can actually get pretty far with just using make in that way, but let's get into making a Makefile so you can understand make a little better. To start off, create a file with just the following in it.

ex2.1.mak

```
CFLAGS=-Wall -g

clean:
    rm -f ex1
```

Save this file as Makefile in your current directory. The program automatically assumes there's a file called Makefile and will just run it.

WARNING! Make sure you are only entering TAB characters, not mixtures of TAB and spaces.

This Makefile is showing you some new stuff with make. First, we set CFLAGS in the file so we never have to set it again, as well as adding the -g flag to get debugging. Then, we have a section named clean that tells make how to clean up our little project.

Make sure it's in the same directory as your ex1.c file, and then run these commands:

```
$ make clean
$ make ex1
```

What You Should See

If that worked, then you should see this:

Exercise 2 Session

```
$ make clean
rm -f ex1
$ make ex1
cc -Wall -g    ex1.c   -o ex1
ex1.c: In function 'main':
ex1.c:3: warning: implicit declaration of function 'puts'
$
```

Here you can see that I'm running make clean, which tells make to run our clean target. Go look at the Makefile again and you'll see that under this command, I indent and then put in the shell commands I want make to run for me. You could put as many commands as you wanted in there, so it's a great automation tool.

WARNING! If you fixed `ex1.c` to have `#include <stdio.h>`, then your output won't have the warning (which should really be an error) about puts. I have the error here because I didn't fix it.

Notice that even though we don't mention `ex1` in the `Makefile`, `make` still knows how to build it *and* use our special settings.

How to Break It

That should be enough to get you started, but first let's break this `Makefile` in a particular way so you can see what happens. Take the line `rm -f ex1` and remove the indent (move it all the way left) so you can see what happens. Rerun `make clean`, and you should get something like this:

```
$ make clean
Makefile:4: *** missing separator.  Stop.
```

Always remember to indent, and if you get weird errors like this, double check that you're consistently using tab characters because some `make` variants are very picky.

Extra Credit

- Create an `all: ex1` target that will build `ex1` with just the command `make`.
- Read `man make` to find out more information on how to run it.
- Read `man cc` to find out more information on what the flags `-Wall` and `-g` do.
- Research `Makefile`s online and see if you can improve this one.
- Find a `Makefile` in another C project and try to understand what it's doing.

This page intentionally left blank

Index

Operators
– (minus sign)
 negative number (unary), 108
 subtract (binary), 108
 subtraction operator, 21
–– (minus signs)
 decrement, then read (prefix), 108
 read, then decrement (postfix), 108
( ) (parentheses)
 C operator, 23
 function call operator, 108
! (exclamation point)
 Boolean NOT operator, 109
 logical NOT operator, 22
? : (question mark, colon)
 Boolean ternary operator, 109
 logical ternary operator, 22
. (period)
 structure reference operator, 23
 structure value access, 108
[ ] (square brackets)
 array index, 108
 array subscript operator, 23
{} (curly braces)
 C operator, 23
 enclosing functions, 7
* (asterisk)
 multiplication operator, 21
 multiply operator (binary), 108
 value of (unary), 108
 value-of operator, 23
& (ampersand)
 address of (unary), 108
 address-of operator, 23
 bitwise AND operator, 22
 Boolean AND operator, 109
^ (caret)
 assign XOR-equal, 109
 bitwise XOR operator, 22
^= (caret, equal)
 assign XOR-equal operator, 23
 bitwise XOR and assign operator, 110
+ (plus sign)
 add operator (binary), 108
 add operator (unary), 108
 addition operator, 21
++ (plus signs)
 increment, then read (prefix), 108
 read, then increment (postfix), 108
|= (vertical bar, equal)
 assign or-equal, 23
 bitwise OR and assign operator, 110
, (comma), C operator, 23
–– (minus signs), decrement operator, 21
/ (slash), divide operator, 21, 108
// (slashes), comment indicator, 7, 24
; (semicolon), statement terminator, 7
: (colon), C operator, 23
!= (exclamation point, equal), not equal operator, 22, 109
?= (question mark, equal), devpkg syntax, 278
*/ (asterisk slash), multi-line comment end, 6, 24
/* (slash asterisk), multi-line comment start, 6, 24
*= (asterisk, equal), assign multiply-equal, 23, 110
&& (ampersands), logical AND operator, 22
&= (ampersand, equal), assign and-equal, 23, 110
++ (plus signs), increment operator, 21
+= (plus sign, equal), assign plus-equal, 23, 110

< (left angle bracket), less than operator, 22, 109
<< (left angle brackets), bitwise shift left operator, 22, 109
<<= (left angle brackets, equal), assign shift-left-equal, 23, 110
<= (left angle bracket, equal), less than or equal operator, 22, 109
= (equal sign), assign equal, 23, 110
−= (minus, equal), assign minus-equal, 23, 110
/= (slash, equal), assign divide-equal, 23, 110
== (equal signs), equals operator, 22, 109
| (vertical bar), bitwise OR operator, 22, 109
| | (vertical bars), Boolean OR operator, 109
~ (tilde), complement operator, 109
% (percent sign)
  modulus operator, 21
  printing as a literal, 33
%= (percent sign, equal), assign modulus-equal, 23, 110

Symbols
-> (dash, right angle bracket)
  structure dereference operator, 23
  structure pointer access, 108
> (right angle bracket), greater than operator, 22, 109
>= (right angle bracket, equal), greater than or equal operator, 22, 109
>> (right angle brackets), bitwise shift right operator, 22, 109
>>= (right angle brackets, equal), assign shift-left-equal, 23, 110
`'0'` (nul) byte, array terminator, 46–49

A
Adler, Mark, 240
`Adler-32` function, 240–247
Alphabetical characters, identifying, 60
Ampersand (&)
  address of (unary), 108
  address-of operator, 23
  bitwise AND operator, 22
  Boolean AND operator, 109
Ampersand, equal (&=), assign and-equal, 23, 110
Ampersands (&&), logical AND operator, 22
APR (Apache Portable Runtime), 274, 275–276
Arguments, passing to a program
  GDB, 18
  LLDB, 19
Arguments, printing, 54
Arithmetic operators, 21
Arrays
  '0' (nul) byte, array terminator, 46–49
  description, 46–49
  dynamic, 198–206
  indexing into, 65–66
  multiple dimensions, 57
  vs. pointers, 67
  sizing, 50–53
  of strings, 54–57
Assignment operators, 23, 109–110
Asterisk (*)
  multiplication operator, 21
  multiply operator (binary), 108
  value of (unary), 108
  value-of operator, 23
Asterisk, equal (*=), assign multiply-equal, 23, 110
Asterisk slash (*/), multi-line comment end, 6, 24
`attach pid` command
  GDB, 19
  LLDB, 19
Attaching to/detaching from a process
  GDB, 19
  LLDB, 19
`auto` operator, 26
Automate everything, 148
Automated testing
  description, 166
  sample code, 166
  wiring the test framework, 167–171

B

Backtrace, dumping
  GDB, 18–19
  LLDB, 19
backtrace command, GDB, 18
bassign function, 227
bassignblk function, 227
bassigncstr function, 227
bchar function, 227
bconcat function, 227
bdata function, 227
bdestroy function, 227
Bernstein, Dan J., 240
Better String Library, 225–227
bfindreplace function, 227
bformat function, 227
bfromcstr function, 227
Binary search, 211–220
Binary search trees, 260–273. *See also* Hashmaps; Ternary search trees.
binstr function, 227
biseq function, 227
Bit operators, 109
Bitwise operators, 22, 109
Blanks, detecting, 60
blength function, 227
blk2bstr function, 227
BMH (Boyer-Moore-Horspool) algorithm, 248–257
Boolean expressions
  switch statements, 42–44
  while-loop statements, 40–41
Boolean operators, 109. *See also* True/false branching.
break command
  flow control, 110
  GDB, 18
break operator, 26
breakpoint set command, LLDB, 19
Breakpoints, GDB
  clearing, 19
  setting, 18
  showing information about, 19
Breakpoints, LLDB
  clearing, 19
  setting, 19
  showing information about, 19
bsplit function, 227
bstrcpy function, 227
BSTree, 260–273. *See also* TSTree.
bstricmp function, 227
bstrlib library, 225–227
bstrlib.o file, 278
Bubble sort, 190–197
Building code. *See also* make command; Makefile.
  in GDB, 18
  in LLDB, 19

C

C language, compilers
  checking version, 2
  confirming, 6–8
  error handling, 90–91
  sample code, 6–8
  support under Windows, 3
C language, operators. *See also specific operators*.
  arithmetic, 21
  assignment, 23, 109–110
  bitwise, 22, 109
  Boolean, 109
  data, 23, 108
  logical, 22, 109
  math, 108
  memorizing, 20–21
  relational, 22
C language, syntax
  keywords, 26–27
  lexemes, 26–29
  lexical analysis, 26–29
  memorizing, 26–29
  syntax structures, 27–30
C preprocessor (CPP)
  conditionally compiling code, 98
  expanding macros, 96–98

C UB (common undefined behavior)
definition, xv
description, 172–173
top 20 undefined behaviors, 174–177
C unions, 212–213
Caret (^)
assign XOR-equal, 109
bitwise XOR operator, 22
Caret, equal (^=)
assign XOR-equal operator, 23
bitwise XOR and assign operator, 110
`case` operator, 26
`cc -Wall -g -DNDEBUG -ldl ex29.c -o ex29` command, 164
`cd` command
GDB, 18
LLDB, 19
Changing directory
GDB, 18
LLDB, 19
`char` operator, 26, 104
Character data types, 104
Character type data, defining, 26
Characters, detecting, 60
Clang's Getting Started instructions, 2
`clear` command, 19
Clearing breakpoints, 19
Code, building. *See* `make` command; `Makefile`.
Colon (`:`), C operator, 23
Comma (`,`), C operator, 23
Command line arguments, printing, 54
`commands.c file`, 288–291
`commands.h file`, 287–288
Comparing strings, 227
Compilers, C language. *See* C language, compilers.
Concatenating strings, 227
`const` operator, 26
`const` qualifier, 105
`continue` command
flow control, 110
GDB, 18
LLDB, 19
`continue` operator, 27
Continue running the program
GDB, 18
LLDB, 19
Control structures, 110
Copying strings, 227
Counting statistical samples, 300, 340–341
CPP (C preprocessor)
conditionally compiling code, 98
expanding macros, 96–98
Creating
double linked list libraries, 178–179
strings, 227
variables, 32–34
Creating, data types
from multiple variables, 27, 32–34
for new types, 29
as structs, 30
Creative programmer mind-set, 140–141
Curly braces (`{}`)
C operator, 23
enclosing functions, 7
Cygwin system, 3

D

`DArray` program, 198–206
Dash, right angle bracket (`->`)
structure dereference operator, 23
structure pointer access, 108
Data operators, 23, 108
Data size, determining, 27
Data structures
definition, 178
fuzzing, 272
testing, 272
Data types. *See also specific data types*.
character, 26, 104
combining into a single record. *See* Structs.
conversion, 105
declare empty, 27
double floating point, 27, 104

enumerated types, 104
floating point, 27, 104
modifiers, 104–107
qualifiers, 105
sizes, 106–107
void, 104
Data types, and flow control
assignment operators, 109–110
bit operators, 109
Boolean operators, 109
control structures, 110
data operators, 108
logic operators, 109
math operators, 108
operators, 107–110
type conversion, 105
type modifiers, 104–107
type qualifiers, 105
type sizes, 106–107
types, 104–107
Data types, creating
from multiple variables, 27, 32–34
for new types, 29
as structs, 30
Data types, integer
declaring, 27
integer constants, 29, 104
short integer, 27
signed modifier for, 27
unsigned modifier for, 27
DB functions, devpkg program, 279–287
db.c file, 280–282
dbg.h macro, 91–95
db.h file, 280
db.o file, 278
Debug macros, 91–95
Debug printing vs. GDB, 100–102
Debugging. *See also* GDB (GNU Debugger); LLDB Debugger.
advanced techniques, 100–102
avoiding stack bugs, 118–119
strategy, 101–102
with vararg functions, 132–136
default operator, 27
Defensive programmer mind-set, 141
Defensive strategies
automate everything, 148
document assumptions, 147
fail early and openly, 146
importance of order, 149
never trust input, 142–145
overview, 141–142
prevent errors, 145–146
prevention over documentation, 147–148
question authority, 149
simplify and clarify, 148–149
Destroying strings, 227
detach command, 19
Detaching from/attaching to a process
GDB, 19
LLDB, 19
devpkg file, 279
devpkg program
?= (question mark, equal), 278
apr_off_t error, 278
apr_thread_proc.h functions, 283
checking for installed URLs, 279–287
command functions, 287–288
commands, 274
commands.c file, 288–291
commands.h file, 287–288
DB functions, 279–287
dependencies, 277
description, 274
devpkg.c file, 292–294
external tools, 283
Main function, 292–294
Makefile, 277–278
off64_t error, 278
README file, 294
recording and listing installed URLs, 279–287
shell functions, 283–287
shell.c file, 284–286
Shell_exec file, 287

devpkg program (*continued*)
  shell.h file, 283–284
  Shell.run function, 286
  source files, 278–279
devpkg.c file, 292–294
Dictionaries. *See* Hashmaps.
DJB Hash function, 240–247
do operator, 27
Document assumptions, 147
Double floating point data types
  declaring, 27
  description, 104
Double linked lists
  creating a library, 178–179
  data structures, definition, 178
  description, 179–181
  implementing, 181–185
  testing, 185–187
double operator, 27, 104
do-while loop
  example, 29
  flow control, 110
  starting, 27
Duff, Tom, 120
Duff's device, 120–125
Dumping a backtrace
  GDB, 18–19
  LLDB, 19
Dynamic arrays, 198–206
Dynamic libraries, 160

E

else operator, 27
else-if statement, 36–38
else-statement, 36–38
Emacs text editor, 4
enum operator, 27, 29, 104
Enumerated data types, 104
Equal sign (=), assign equal, 23, 110
Equal signs (==), equals operator, 22, 109
Equality testing. *See* Logic operators.
Error codes, 90–91
Error handling. *See* C language, error handling.
ex22.c file, 112–114
ex22.h file, 112–114
ex22_main.c file, 114–118
Exclamation point (!)
  Boolean NOT operator, 109
  logical NOT operator, 22
Exclamation point, equal (!=), not equal operator, 22, 109
Exit out of a compound statement, 26
extern operator, 27

F

Fail early and openly, 146
fclose function, 129
fcloseall function, 130
fdopen function, 129
fgetpos function, 130
fgets function, 126–130
Find and replace strings, 227
float operator, 27, 104
Floating point data types
  declaring, 27
  description, 104
FNV-1a function, 240–247
fopen function, 129
for operator, 27
for-loops
  arrays of strings, 54–57
  example, 29
  starting, 27
Formatted printing, 14–16
Formatting strings, 227
Fowler, Glenn, 240
fprintf function, 130
fread function, 130
freopen function, 129
fscanf function, 126–130
fseek function, 130
ftell function, 130

Function calls, stepping into
  GDB, 18
  LLDB, 19
Function calls, stepping over
  GDB, 18
  LLDB, 19
Functions. *See also specific functions*.
  bad, checking for, 158–159
  defining, 29
  I/O handling, 126–130
  returning from, 27
  using, 58–60
  vararg, 132–136
  with variable arguments, 132–136
  writing, 58–60
Functions, pointers to
  description, 84–85
  format, 84
  sample code, 84–85
Fuzzing data structures, 272
`fwrite` function, 130

G

GDB (GNU Debugger). *See also* LLDB Debugger.
  attaching to/detaching from a running process, 19
  build code, 18
  change directory, 18
  continue running the program, 18
  vs. debug printing, 100–102
  dumping a backtrace, 18–19
  help, 18
  list ten source lines, 19
  passing arguments to the program, 18
  quit, 18
  start a program, 18
  start a shell, 19
  step into function calls, 18
  step over function calls, 18
  watchpoints, showing information about, 19
GDB (GNU Debugger), breakpoints
  clearing, 19
  setting, 18
  showing information about, 19
GDB (GNU Debugger), commands
  `attach pid`, 19
  `backtrace`, 18
  `break`, 18
  `cd`, 18
  `clear`, 19
  `continue`, 18
  `detach`, 19
  `help`, 18
  `info break`, 19
  `info watch`, 19
  `list`, 19
  `make`, 18
  `next`, 18
  `print expr`, 18
  `pwd`, 18
  quick reference, 18–19
  `quit`, 18
  `run`, 18
  `shell`, 19
  `step`, 18
GDB (GNU Debugger), printing
  value of an expression, 18
  working directory, 18
gedit text editor, 3
Getting started. *See* Setting up your computer.
GNU Debugger (GDB). *See* GDB (GNU Debugger).
goto operator, 27, 29, 110

H

Hashmaps
  `Adler-32` function, 240–247
  algorithms, 240–247
  vs. binary or ternary search trees, 323
  definition, 228
  `DJB Hash` function, 240–247
  example, 228–235
  finding prefixes, 327

Hashmaps (*continued*)
  FNV-1a function, 240–247
  scanning, 235
  unit testing, 235–237
Hashmap_traverse function, 235
Headers, Makefile example, 154–155
Heap sort, 208–210
Heaps
  potential problems, 81
  sample code, 74–79
  vs. stack memory allocation, 74–79
Help
  GDB, 18
  LLDB, 19
help command
  GDB, 18
  LLDB, 19

I
IDE (Integrated Development Environment), 4
Identifiers, declaring as external, 27
if operator, 27
if-statement
  else branch, 27
  example, 28
  starting, 27
  true/false branching, 36–38
Indexing into arrays, 65–66
Inequality testing. *See* Logic operators.
info break command, 19
info watch command, 19
Input/output. *See* I/O.
Installing
  a Makefile, example, 158
  software. *See* devpkg program.
Int constants, defining a set of, 27
int operator, 27, 104
int8_t type size, 106
int16_t type size, 106
int32_t type size, 106
int64_t type size, 106
Integer constants, data types for, 104
Integer data type
  declaring, 27
  signed modifier for, 27
  unsigned modifier for, 27
Integrated Development Environment (IDE), 4
INT_FAST (N)_MAX type size, 107
int_fast (N)_t type size, 107
INT_FAST(N)_MAX type size, 107
int_fast(N)_t type size, 107
INT_LEAST (N)_MAX type size, 107
INT_LEAST (N)_MIN type size, 107
int_least (N)_t type size, 107
INT_LEAST(N)_MAX type size, 107
INT_LEAST(N)_MIN type size, 107
int_least(N)_t type size, 107
INTMAX_MAX type size, 107
INTMAX_MIN type size, 107
intmax_t type size, 107
INTPTR_MAX type size, 107
INTPTR_MIN type size, 107
intptr_t type size, 107
I/O, reading from files, 126–130
isalpha function, 60
isblank function, 60

J
Jump tables, 42–44
Jumping to a label, 27, 29

K
Keywords, C, 26–27

L
Left angle bracket, equal (<=), less than or equal operator, 22, 109
Left angle bracket (<), less than operator, 22, 109
Left angle brackets, equal (<<=), assign shift-left-equal, 23, 110
Left angle brackets (<<), bitwise shift left operator, 22, 109
Length of strings, getting, 227
Letters, identifying, 60

Lexemes, C syntax, 26–29
Lexical analysis, C syntax, 26–29
Libraries
  Better String Library, 225–227
  `bstrlib`, 225–227
  double linked lists, 178–179
  dynamic, 160
  linking to, 160–164
  shared, dynamic loading, 161–164
  static, 160
Linked list algorithms, sorting with, 190–197
Linking to libraries, 160–164
Linux
  compiler version, checking, 2
  running under Windows, 3
  setting up your computer, 2
`list` command, 19
List next ten source lines in GDB and LLDB, 19
`list_algos.c` file, 193–195
`list_algos.h` file, 193
`list_algos_tests.f` file, 191–193
LLDB Debugger. *See also* GDB (GNU Debugger).
  attaching to/detaching from a process, 19
  building code, 19
  change directory, 19
  continue running the program, 19
  dumping a backtrace, 19
  help, 19
  listing next ten lines, 19
  quitting, 19
  starting a shell, 19
  starting the program, 19
  stepping into function calls, 19
  stepping over function calls, 19
  watchpoints, showing information about, 19
LLDB Debugger, breakpoints
  clearing, 19
  setting, 19
  showing information about, 19
LLDB Debugger, commands
  `attach pid`, 19
  `breakpoint set`, 19
  `cd`, 19
  `clear`, 19
  `continue`, 19
  `detach`, 19
  `help`, 19
  `info break`, 19
  `info watch`, 19
  `list`, 19
  `make`, 19
  `next`, 19
  `print expr`, 19
  `pwd`, 19
  quick reference, 19
  `quit`, 19
  `run` command, 19
  `shell`, 19
  `step`, 19
  `thread backtrace`, 19
LLDB Debugger, printing
  expressions, 19
  working directory, 19
`load` command, 344
Local variables, giving a local lifetime, 26
Log files, finding, 138
Logfind project, 138
Logic operators, 22, 109
`long` modifier, 104
Loops. *See also specific loops*.
  breaking to exit, 28
  continuing to the top of, 27
  infinite, 144

M

Mac OS X
  compiler version, checking, 2
  setting up your computer, 2–3
Macros
  `dbg.h`, 91–95
  for debugging, 91–95
  expanding, 96–98
MacVim text editor, 4
`Main` function, devpkg program, 292–294

`make clean` command, 10–12
`make` command
  building code, 10–12
  GDB, 18
  LLDB, 19
`Makefile`
  as automation tool, 11
  building code, 10–12
  devpkg program, 277–278
`Makefile`, examples
  basic structure, 152–154
  checking for bad functions, 158–159
  cleanup, 157–158
  header, 154–155
  installing, 158
  target build, 155–156
  unit tests, 156–157
Math operators, 108
Max/min samples, identifying, 300
Mean, calculating, 300, 340–341
Memorizing
  C operators, 20–21
  C syntax, 26–29
Memory
  format conversion, 212–213
  leaks, shown by the debugger, 80
  stack allocation, 80–81
Merge sort, 190–197, 208–210
Middle number, calculating, 300, 340–341
Mind-set for programming. *See* Programmer mind-set.
MinGw system, 3
Min/max samples, identifying, 300, 340–341
Minus sign (-)
  negative number (unary), 108
  subtract (binary), 108
  subtraction operator, 21
Minus sign, equal (−=), assign minus-equal, 23, 110
Minus signs (−−)
  decrement, then read (prefix), 108
  read, then decrement (postfix), 108
Minus signs (−−), decrement operator, 21

N
Nano text editor, 4
`netclient.c` file, 316–320
Network server program, 338
Never trust input, 142–145
`next` command
  GDB, 18
  LLDB, 19
Noll, Leonard Curt, 240
Nul byte, array terminator, 46–49

O
`off64_t` error, 278
Operators. *See* C language, operators.
Output. *See* I/O.

P
Percent sign (%)
  modulus operator, 21
  printing as a literal, 33
Percent sign, equal (%=), assign modulus-equal, 23, 110
Period (`.`)
  structure reference operator, 23
  structure value access, 108
Plus sign (+)
  add operator (binary), 108
  add operator (unary), 108
  addition operator, 21
Plus sign, equal (+=), assign plus-equal, 23, 110
Plus signs (++)
  increment, then read (prefix), 108
  read, then increment (postfix), 108
Plus signs (++), increment operator, 21
Pointers
  vs. arrays, 67
  definition, 65
  description, 65
  indexing into arrays, 65–66
  lexicon of, 66–67

  sample code, 62–64
  to structures, 68–71
  uses for, 66
Pointers, to functions
  description, 84–85
  format, 84
  sample code, 84–85
Prevent errors, 145–146
Prevention over documentation, 147–148
print expr command
  GDB, 18
  LLDB, 19
printf function, 14–16
Printing
  % (percent signs), as literals, 33
  command line arguments, 54
  expression values, 18, 19
  formatting, 14–16
  from GDB, 18
  from LLDB, 19
  in scientific notation, 33
  working directory, 18, 19
Programmer mind-set
  creative, 140–141
  defensive, 141
Programmer mind-set, defensive strategies
  automate everything, 148
  document assumptions, 147
  fail early and openly, 146
  importance of order, 149
  never trust input, 142–145
  overview, 141–142
  prevent errors, 145–146
  prevention over documentation, 147–148
  question authority, 149
  simplify and clarify, 148–149
PTRDIFF_MAX type size, 107
PTRDIFF_MIN type size, 107
pwd command
  GDB, 18
  LLDB, 19

Q

Question authority, 149
Question mark, colon (?:)
  Boolean ternary operator, 109
  logical ternary operator, 22
Question mark, equal (?=), devpkg syntax, 278
Queues, 296–299
Quick sort, 208–210
quit command
  GDB, 18
  LLDB, 19
Quitting
  GDB, 18
  LLDB, 19

R

Radix sort, 211–221
RadixMap_add function, 219
RadixMap_create function, 219
RadixMap_delete function, 219
RadixMap_destroy function, 219
RadixMap_find function, 219–220
RadixMap_sort function, 219, 220–221
radix_sort function, 219, 220–221
Reading from files, 126–130
README file, devpkg program, 294
register operator, 27
register qualifier, 105
Relational operators, 22
Return from a function, 27
return operator, 27
rewind function, 130
Right angle bracket, equal (>=), greater than or equal operator, 22, 109
Right angle bracket (>), greater than operator, 22, 109
Right angle brackets, equal (>>=), assign shift-left-equal, 23, 110
Right angle brackets (>>), bitwise shift right operator, 22, 109
Ring buffers, 310–314

run command
  GDB, 18
  LLDB, 19

S

Scanning hashmaps, 235
Scope
  ex22.c file, 112–114
  ex22.h file, 112–114
  ex22_main.c file, 114–118
  and stacks, 118–119
Search algorithms, 248–257
Searching
  binary search, 211–220
  binary search trees, 260–273
  ternary search trees, 322–330
Semicolon (;), statement terminator, 7
Server improvements, 346
Server login times, summary statistics, 307
Setting
  breakpoints, GDB, 18
  breakpoints, LLDB, 19
  strings, 227
Setting up your computer
  Clang's Getting Started instructions, 2
  Linux, 2
  Mac OS X, 2–3
  text editors, 3–4. *See also specific text editors.*
  Windows, 3
Shared libraries, dynamic loading, 161–164
Shaw, Zed A., contact information, xv
shell command, 19
shell.c file, 284–286
Shell_exec file, 287
shell.h file, 283–284
shell.o file, 278
Shell.run function, 286
Shells, starting, 19
Short integer data type, declaring, 27
short modifier, 104
short operator, 27
Showing information about breakpoints, 19
signed modifier
  description, 104
  for integer data types, 27
Simplify and clarify, 148–149
SIZE_MAX type size, 107
sizeof operator
  data access, 108
  description, 23, 27
  sizing arrays, 50–53
Slash, equal (/=), assign divide-equal, 23, 110
Slash (/), divide operator, 21, 108
Slash asterisk (/*), multi-line comment start, 6, 24
Slashes (//), comment indicator, 7, 24
Sorting
  bubble sort, 190–197
  heap sort, 208–210
  with linked list algorithms, 190–197
  merge sort, 190–197, 208–210
  quick sort, 208–210
  radix sort, 211–221
  statistics, 344
Splitting strings, 227
Square brackets ([ ])
  array index, 108
  array subscript operator, 23
Squares of numbers, calculating, 300
Stacks
  avoiding bugs, 118–119
  definition, 81
  description, 296–299
  memory allocation, 80–81
  potential problems, 81
Standard deviation, calculating, 300–304
Starting a program with arguments
  GDB, 18
  LLDB, 19
Starting a shell, 19
Static libraries, 160
static operator, 27
Statistics. *See* Summary statistics.
Statistics server, 340–341
stats.c file, 302–304

Stats_create function, 304
Stats_dump function, 304
statserve program, 338, 340–341
stats.h API, 340–341
Stats_mean function, 304
Stats_recreate function, 304
Stats_sample function, 304
Stats_stddev function, 304
stats_tests.c file, 304–306
step command
  GDB, 18
  LLDB, 19
Stepping into function calls
  GDB, 18
  LLDB, 19
Stepping over function calls
  GDB, 18
  LLDB, 19
store command, 344
String_base_search function, 252–255
String_find function, 249–255, 257
Strings
  arrays of, 54–57
  Better String Library, 225–227
  BMH (Boyer-Moore-Horspool) algorithm, 248–257
  checking for validity, 224–225
  comparing, 227
  concatenating, 227
  copying, 227
  creating, 227
  destroying, 227
  disadvantages of, 224–225
  find and replace, 227
  formatting, 227
  functions for, 227
  getting characters from, 227
  getting data from, 227
  getting length of, 227
  search algorithms, 248–257
  setting, 227
  splitting, 227
  storing as arrays, 46–49
  testing for equality, 227
StringScanner_scan function, 249–255, 257
String_setup_skip_chars function, 252–255
struct operator, 27, 30
Structs, 68–71
Sum, calculating, 300, 340–341
Sum of squares, calculating, 340–341
Summary statistics
  counting samples, 300, 340–341
  load command, 344
  loading from a hard drive, 344
  mean, calculating, 300, 340–341
  middle number, calculating, 300, 340–341
  min/max samples, 300, 340–341
  routing, 342
  for server login times, 307
  sorting, 344
  standard deviation, calculating, 300–304
  statistics on statistics, 306–307
  statistics server, 340–341
  stats.c file, 302–304
  Stats_create function, 304
  Stats_dump function, 304
  stats.h API, 340–341
  Stats_mean function, 304
  Stats_recreate function, 304
  Stats_sample function, 304
  Stats_stddev function, 304
  stats_tests.c file, 304–306
  store command, 344
  storing to a hard drive, 344
  sum, calculating, 300, 340–341
  sum of squares, calculating, 300, 340–341
  unit test, 304
switch operator, 27
switch-statements
  branching in a, 26
  default branch, 27
  description, 42–44

switch-statements (*continued*)
  example, 28
  starting, 27
Syntax structures, C syntax, 27–30

T

TCP/IP client, 316–321
Ternary search trees, 322–330. *See also* Binary search trees; TSTree.
Testing
  automated. *See* Automated testing.
  data structures, 272
  double linked lists, 185–187
  strings for equality, 227
Text editors, 3–4. *See also specific text editors*.
TextWrangler text editor, 3
thread backtrace command, LLDB, 19
Tilde (~), complement operator, 109
True/false branching, 36–38
TSTree. *See also* BSTree.
  fast URL routing, 332–336
  searching with, 322–330
typedef operator, 27, 30

U

UB (undefined behavior). *See* C UB (common undefined behavior).
uint8_t type size, 106
uint16_t type size, 106
uint32_t type size, 106
uint64_t type size, 106
UINT_FAST (N)_MAX type size, 107
uint_fast (N)_t type size, 107
UINT_FAST(N)_MAX type size, 107
uint_fast(N)_t type size, 107
UINT_LEAST (N)_MAX type size, 107
uint_least (N)_t type size, 107
UINT_LEAST(N)_MAX type size, 107
uint_least(N)_t type size, 107
UINTMAX_MAX type size, 107
uintmax_t type size, 107
UINTPTR_MIN type size, 107
uintptr_t type size, 107
union operator, 27, 30
Unions, 212–213
union-statement, starting, 27, 30
unsigned operator, 27, 104
URL routing, 332–336
Urlor tool, 332–336

V

Validity checking strings, 224–225
Variables
  combining into a single record, 27. *See also* Structs.
  creating, 32–34
  declaring as modifiable, 27
  declaring to be stored in a CPU register, 27
  make unmodifiable, 26
  preserving value after scope exits, 27
Vertical bar, equal (|=)
  assign or-equal, 23
  bitwise OR and assign operator, 110
Vertical bar (|), bitwise OR operator, 22, 109
Vertical bars (||), Boolean OR operator, 109
Vim text editor, 4
VirtualBox, 3
Vo, Phong, 240
Void data types, 104
void operator, 27, 104
volatile operator, 27
volatile type qualifier, 105

W

Watchpoints, showing information about GDB and LLDB, 19
while operator, 27
while-loop
  Boolean expressions, 40–41
  example, 28
  starting, 27
Windows
  C support, 3
  running Linux under, 3
  setting up your computer, 3

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON **InformIT** is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the leading brands, authors, and contributors from the tech community.

Addison-Wesley Cisco Press IBM Press Microsoft Press

PEARSON IT CERTIFICATION PRENTICE HALL QUE SAMS vmware PRESS

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials. Looking for expert opinions, advice, and tips? **InformIT has a solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of monthly newsletters. Visit **informit.com/newsletters**.
- FREE Podcasts from experts at **informit.com/podcasts**.
- Read the latest author articles and sample chapters at **informit.com/articles**.
- Access thousands of books and videos in the Safari Books Online digital library. **safari.informit.com**.
- Get Advice and tips from expert blogs at **informit.com/blogs**.

Visit **informit.com** to find out all the ways you can access the hottest technology content.

Are you part of the IT crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube and more! Visit **informit.com/socialconnect**.

ALWAYS LEARNING PEARSON