

Learn C The Hard Way

A Learn Code The Hard Way Book

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Learn C The Hard Way

A Clear & Direct Introduction To Modern C Programming

Zed A. Shaw

July 2011

[Contents](#)

[Preface](#)

[Introduction: The Cartesian Dream Of C](#)

[What You Will Learn](#)

[How To Read This Book](#)

[The Core Competencies](#)

[License](#)

I [Basic Skills](#)

1 [Exercise 0: The Setup](#)

1.1 [Linux](#)

1.2 [Mac OSX](#)

1.3 [Windows](#)

1.4 [Text Editor](#)

2 [Exercise 1: Dust Off That Compiler](#)

2.1 [What You Should See](#)

2.2 [How To Break It](#)

2.3 [Extra Credit](#)

3 [Exercise 2: Make Is Your Python Now](#)

3.1 [Using Make](#)

3.2 [What You Should See](#)

3.3 [How To Break It](#)

3.4 [Extra Credit](#)

4 [Exercise 3: Formatted Printing](#)

4.1 [What You Should See](#)

4.2 [External Research](#)

4.3 [How To Break It](#)

4.4 [Extra Credit](#)

5 [Exercise 4: Introducing Valgrind](#)

5.1 [Installing Valgrind](#)

5.2 [Using Valgrind](#)

- 5.3 [What You Should See](#)
- 5.4 [Extra Credit](#)
- 6 [Exercise 5: The Structure Of A C Program](#)
 - 6.1 [What You Should See](#)
 - 6.2 [Breaking It Down](#)
 - 6.3 [Extra Credit](#)
- 7 [Exercise 6: Types Of Variables](#)
 - 7.1 [What You Should See](#)
 - 7.2 [How To Break It](#)
 - 7.3 [Extra Credit](#)
- 8 [Exercise 7: More Variables, Some Math](#)
 - 8.1 [What You Should See](#)
 - 8.2 [How To Break It](#)
 - 8.3 [Extra Credit](#)
- 9 [Exercise 8: Sizes And Arrays](#)
 - 9.1 [What You Should See](#)
 - 9.2 [How To Break It](#)
 - 9.3 [Extra Credit](#)
- 10 [Exercise 9: Arrays And Strings](#)
 - 10.1 [What You Should See](#)
 - 10.2 [How To Break It](#)
 - 10.3 [Extra Credit](#)
- 11 [Exercise 10: Arrays Of Strings, Looping](#)
 - 11.1 [What You Should See](#)
 - 11.2 [How To Break It](#)
 - 11.3 [Extra Credit](#)
- 12 [Exercise 11: While-Loop And Boolean Expressions](#)
 - 12.1 [What You Should See](#)
 - 12.2 [How To Break It](#)
 - 12.3 [Extra Credit](#)
- 13 [Exercise 12: If, Else-If, Else](#)
 - 13.1 [What You Should See](#)
 - 13.2 [How To Break It](#)
 - 13.3 [Extra Credit](#)
- 14 [Exercise 13: Switch Statement](#)
 - 14.1 [What You Should See](#)
 - 14.2 [How To Break It](#)
 - 14.3 [Extra Credit](#)
- 15 [Exercise 14: Writing And Using Functions](#)
 - 15.1 [What You Should See](#)
 - 15.2 [How To Break It](#)
 - 15.3 [Extra Credit](#)
- 16 [Exercise 15: Pointers Dreaded Pointers](#)
 - 16.1 [What You Should See](#)
 - 16.2 [Explaining Pointers](#)
 - 16.3 [Practical Pointer Usage](#)
 - 16.4 [The Pointer Lexicon](#)

- 16.5 [How To Break It](#)
- 16.6 [Extra Credit](#)
- 17 [Exercise 16: Structs And Pointers To Them](#)
 - 17.1 [What You Should See](#)
 - 17.2 [Explaining Structures](#)
 - 17.3 [How To Break It](#)
 - 17.4 [Extra Credit](#)
- 18 [Exercise 17: Heap And Stack Memory Allocation](#)
 - 18.1 [What You Should See](#)
 - 18.2 [Heap vs. Stack Allocation](#)
 - 18.3 [How To Break It](#)
 - 18.4 [Extra Credit](#)
- 19 [Exercise 18: Pointers To Functions](#)
 - 19.1 [What You Should See](#)
 - 19.2 [How To Break It](#)
 - 19.3 [Extra Credit](#)
- 20 [Exercise 19: A Simple Object System](#)
 - 20.1 [How The CPP Works](#)
 - 20.2 [The Prototype Object System](#)
 - 20.3 [The Game Implementation](#)
 - 20.4 [What You Should See](#)
 - 20.5 [Auditing The Game](#)
 - 20.6 [Extra Credit](#)
- 21 [Exercise 20: Zed's Awesome Debug Macros](#)
 - 21.1 [The C Error Handling Problem](#)
 - 21.2 [The Debug Macros](#)
 - 21.3 [Using dbg.h](#)
 - 21.4 [What You Should See](#)
 - 21.5 [How The CPP Expands Macros](#)
 - 21.6 [Extra Credit](#)
- 22 [Exercise 21: Advanced Data Types And Flow Control](#)
 - 22.1 [Available Data Types](#)
 - 22.2 [Available Operators](#)
 - 22.3 [Available Control Structures](#)
- 23 [Exercise 22: The Stack, Scope, And Globals](#)
 - 23.1 [What You Should See](#)
 - 23.2 [Scope, Stack, And Bugs](#)
 - 23.3 [How To Break It](#)
 - 23.4 [Extra Credit](#)
- 24 [Exercise 23: Meet Duff's Device](#)
 - 24.1 [What You Should See](#)
 - 24.2 [Solving The Puzzle](#)
 - 24.3 [Extra Credit](#)
- 25 [Exercise 24: Input, Output, Files](#)
 - 25.1 [What You Should See](#)
 - 25.2 [How To Break It](#)
 - 25.3 [The I/O Functions](#)

25.4 [Extra Credit](#)

26 [Exercise 25: Variable Argument Functions](#)

26.1 [What You Should See](#)

26.2 [How To Break It](#)

26.3 [Extra Credit](#)

27 [Exercise 26: Write A First Real Program](#)

27.1 [What Is devpkg?](#)

27.2 [Project Layout](#)

27.3 [The Makefile](#)

27.4 [The Source Files](#)

27.5 [The Mid-Term Exam](#)

II [Data Structures And Algorithms](#)

28 [Exercise 27: Creative And Defensive Programming](#)

28.1 [The Creative Programmer Mindset](#)

28.2 [The Defensive Programmer Mindset](#)

28.3 [The Eight Defensive Programmer Strategies](#)

28.4 [Applying The Eight Strategies](#)

28.5 [Order Is Not Important](#)

28.6 [Extra Credit](#)

29 [Exercise 28: Intermediate Makefiles](#)

29.1 [The Basic Project Structure](#)

29.2 [Makefile](#)

29.3 [What You Should See](#)

29.4 [Extra Credit](#)

30 [Exercise 29: Libraries And Linking](#)

30.1 [What You Should See](#)

30.2 [How To Break It](#)

30.3 [Extra Credit](#)

31 [Exercise 30: Automated Testing](#)

31.1 [Wiring Up The Test Framework](#)

31.2 [Extra Credit](#)

32 [Exercise 31: Debugging Code](#)

32.1 [Debug Printing Vs. GDB Vs. Valgrind](#)

32.2 [A Debugging Strategy](#)

32.3 [Using GDB](#)

32.4 [Process Attaching](#)

32.5 [GDB Tricks](#)

32.6 [Extra Credit](#)

33 [Exercise 32: Double Linked Lists](#)

33.1 [What Are Data Structures](#)

33.2 [Making The Library](#)

33.3 [Double Linked Lists](#)

33.4 [Tests](#)

33.5 [What You Should See](#)

33.6 [How To Improve It](#)

33.7 [Extra Credit](#)

34 [Exercise 33: Linked List Algorithms](#)

- 34.1 [What You Should See](#)
- 34.2 [How To Improve It](#)
- 34.3 [Extra Credit](#)
- 35 [Exercise 34: Dynamic Array](#)
- 35.1 [Advantages And Disadvantages](#)
- 35.2 [How To Improve It](#)
- 35.3 [Extra Credit](#)
- 36 [Exercise 35: Sorting And Searching](#)
- 36.1 [Radix Sort And Binary Search](#)
- 36.2 [How To Improve It](#)
- 36.3 [Extra Credit](#)
- 37 [Exercise 36: Safer Strings](#)
- 37.1 [Why C Strings Were A Horrible Idea](#)
- 37.2 [Using bstrlib](#)
- 37.3 [Learning The Library](#)
- 38 [Exercise 37: Hashmaps](#)
- 38.1 [How To Improve It](#)
- 38.2 [Extra Credit](#)
- 39 [Exercise 38: Hashmap Algorithms](#)
- 39.1 [What You Should See](#)
- 39.2 [How To Break It](#)
- 39.3 [Extra Credit](#)
- 40 [Exercise 39: String Algorithms](#)
- 40.1 [What You Should See](#)
- 40.2 [Analyzing The Results](#)
- 40.3 [Extra Credit](#)
- 41 [Exercise 40: Binary Search Trees](#)
- 41.1 [How To Improve It](#)
- 41.2 [Extra Credit](#)
- 42 [Exercise 41: Using Cachegrind And Callgrind For Performance Tuning](#)
- 42.1 [Running Callgrind](#)
- 42.2 [Callgrind Annotating Source](#)
- 42.3 [Analyzing Memory Access With Cachegrind](#)
- 42.4 [Judo Tuning](#)
- 42.5 [Using KCachegrind](#)
- 42.6 [Extra Credit](#)
- 43 [Exercise 42: Stacks and Queues](#)
- 43.1 [What You Should See](#)
- 43.2 [How To Improve It](#)
- 43.3 [Extra Credit](#)
- 44 [Exercise 43: A Simple Statistics Engine](#)
- 44.1 [Rolling Standard Deviation And Mean](#)
- 44.2 [Implementation](#)
- 44.3 [How To Use It](#)
- 44.4 [Extra Credit](#)
- 45 [Exercise 44: Ring Buffer](#)
- 45.1 [The Unit Test](#)

- 45.2 [What You Should See](#)
- 45.3 [How To Improve It](#)
- 45.4 [Extra Credit](#)
- 46 [Exercise 45: A Simple TCP/IP Client](#)
- 46.1 [Augment The Makefile](#)
- 46.2 [The netclient Code](#)
- 46.3 [What You Should See](#)
- 46.4 [How To Break It](#)
- 46.5 [Extra Credit](#)
- 47 [Exercise 46: Ternary Search Tree](#)
- 47.1 [What You Should See](#)
- 47.2 [How To Break It](#)
- 47.3 [Extra Credit](#)
- 48 [Exercise 47: Ternary Search Tree Algorithms](#)
- 48.1 [What You Should See](#)
- 48.2 [How To Break It](#)
- 48.3 [Extra Credit](#)
- 49 [Exercise 48: A Tiny Virtual Machine Part 1](#)
- 49.1 [What You Should See](#)
- 49.2 [How To Break It](#)
- 49.3 [Extra Credit](#)
- 50 [Exercise 48: A Tiny Virtual Machine Part 2](#)
- 50.1 [What You Should See](#)
- 50.2 [How To Break It](#)
- 50.3 [Extra Credit](#)
- 51 [Exercise 50: A Tiny Virtual Machine Part 3](#)
- 51.1 [What You Should See](#)
- 51.2 [How To Break It](#)
- 51.3 [Extra Credit](#)
- 52 [Exercise 51: A Tiny Virtual Machine Part 4](#)
- 52.1 [What You Should See](#)
- 52.2 [How To Break It](#)
- 52.3 [Extra Credit](#)
- 53 [Exercise 52: A Tiny Virtual Machine Part 5](#)
- 53.1 [What You Should See](#)
- 53.2 [How To Break It](#)
- 53.3 [Extra Credit](#)
- 54 [Next Steps](#)
- III [Reviewing And Critiquing Code](#)
- 55 [Deconstructing "K&R C"](#)
- 55.1 [An Overall Critique Of Correctness](#)
- 55.2 [Chapter 1 Examples](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

[Home](#)
[Python](#)
[Ruby](#)
[C](#)
[SQL](#)
[Regex](#)
[Blog](#)

Learn C The Hard Way (ALPHA)

Other Books

[Learn Python](#)

[Learn Ruby](#)

[Learn C](#)

[Learn Regex](#)

[Learn SQL](#)

[CLI Crash Course](#)

More Stuff

[About](#)

[Blog](#)

Introducing Learn C The Hard Way

Learn C The Hard Way will fill in the "beginning programmer" gap in the literature on the C Programming language. It will teach good **modern** C programming practices and avoid habits that lead to buffer overflows, security defects, and other problems that even experienced programmers cause. This book is designed for:

Students who have read [LPTHW](#) or [LRTHW](#).

Programmers with skill in one other "scripting" language.

Programmers who need to relearn C.

Objective-C programmers who want better C skills.

People who like a challenge.

[Read More](#)

Online Udemy Course Started

I've now started an online class with [Udemy.com](#) that will feature videos of me walking you through the different exercises. As I complete exercises I'll be putting up videos that cover them, usually about 5 at a time. The course is **\$29** and you'll get a copy of the PDF book when it's available to read.

[Signup For The Online Class](#)

Download The ALPHA Release

Right now there's an HTML version of the book up for people to read.

[Read More](#)

Helping Edit

If you are an experienced programmer and want to edit code that's in the book, or help with editing, then hit the [git repo](#) and start digging. You know what to do:

Fork my git repo.

Make your changes.

Send me a merge request.

By asking me to merge your changes in you agree that I retain copyright. Please don't make me make you sign something to prove that.

Useful Links

[The Blog](#)

[Contact](#)

Popular Articles

[Blog](#)

Current Status

[Learn Ruby The Hard Way now for sale in PDF and ePub format for \\$2.99](#)

Contact

Feel free to email and ask questions if you're having trouble with any book.

E-Mail: help@learncodethehardway.org

Copyright 2011 Zed A. Shaw. All Rights Reserved. Powered By Software He Wrote.

Learn C The Hard Way

A Learn Code The Hard Way Book

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Contents

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

I Basic Skills

1 [Exercise 0: The Setup](#)

1.1 [Linux](#)

1.2 [Mac OSX](#)

1.3 [Windows](#)

1.4 [Text Editor](#)

1.4.1 [WARNING: Do Not Use An IDE](#)

2 [Exercise 1: Dust Off That Compiler](#)

2.1 [What You Should See](#)

2.2 [How To Break It](#)

2.3 [Extra Credit](#)

3 [Exercise 2: Make Is Your Python Now](#)

3.1 [Using Make](#)

3.2 [What You Should See](#)

3.3 [How To Break It](#)

3.4 [Extra Credit](#)

4 [Exercise 3: Formatted Printing](#)

4.1 [What You Should See](#)

4.2 [External Research](#)

4.3 [How To Break It](#)

4.4 [Extra Credit](#)

5 [Exercise 4: Introducing Valgrind](#)

5.1 [Installing Valgrind](#)

5.2 [Using Valgrind](#)

5.3 [What You Should See](#)

5.4 [Extra Credit](#)

6 [Exercise 5: The Structure Of A C Program](#)

6.1 [What You Should See](#)

6.2 [Breaking It Down](#)

6.3 [Extra Credit](#)

7 [Exercise 6: Types Of Variables](#)

7.1 [What You Should See](#)

7.2 [How To Break It](#)

7.3 [Extra Credit](#)

8 [Exercise 7: More Variables, Some Math](#)

8.1 [What You Should See](#)

- 8.2 [How To Break It](#)
- 8.3 [Extra Credit](#)
- 9 [Exercise 8: Sizes And Arrays](#)
 - 9.1 [What You Should See](#)
 - 9.2 [How To Break It](#)
 - 9.3 [Extra Credit](#)
- 10 [Exercise 9: Arrays And Strings](#)
 - 10.1 [What You Should See](#)
 - 10.2 [How To Break It](#)
 - 10.3 [Extra Credit](#)
- 11 [Exercise 10: Arrays Of Strings, Looping](#)
 - 11.1 [What You Should See](#)
 - 11.1.1 [Understanding Arrays Of Strings](#)
 - 11.2 [How To Break It](#)
 - 11.3 [Extra Credit](#)
- 12 [Exercise 11: While-Loop And Boolean Expressions](#)
 - 12.1 [What You Should See](#)
 - 12.2 [How To Break It](#)
 - 12.3 [Extra Credit](#)
- 13 [Exercise 12: If, Else-If, Else](#)
 - 13.1 [What You Should See](#)
 - 13.2 [How To Break It](#)
 - 13.3 [Extra Credit](#)
- 14 [Exercise 13: Switch Statement](#)
 - 14.1 [What You Should See](#)
 - 14.2 [How To Break It](#)
 - 14.3 [Extra Credit](#)
- 15 [Exercise 14: Writing And Using Functions](#)
 - 15.1 [What You Should See](#)
 - 15.2 [How To Break It](#)
 - 15.3 [Extra Credit](#)
- 16 [Exercise 15: Pointers Dreaded Pointers](#)
 - 16.1 [What You Should See](#)
 - 16.2 [Explaining Pointers](#)
 - 16.3 [Practical Pointer Usage](#)
 - 16.4 [The Pointer Lexicon](#)
 - 16.5 [How To Break It](#)
 - 16.6 [Extra Credit](#)
- 17 [Exercise 16: Structs And Pointers To Them](#)
 - 17.1 [What You Should See](#)
 - 17.2 [Explaining Structures](#)
 - 17.3 [How To Break It](#)
 - 17.4 [Extra Credit](#)
- 18 [Exercise 17: Heap And Stack Memory Allocation](#)
 - 18.1 [What You Should See](#)
 - 18.2 [Heap vs. Stack Allocation](#)
 - 18.3 [How To Break It](#)

- 18.4 [Extra Credit](#)
- 19 [Exercise 18: Pointers To Functions](#)
 - 19.1 [What You Should See](#)
 - 19.2 [How To Break It](#)
 - 19.3 [Extra Credit](#)
- 20 [Exercise 19: A Simple Object System](#)
 - 20.1 [How The CPP Works](#)
 - 20.2 [The Prototype Object System](#)
 - 20.2.1 [The Object Header File](#)
 - 20.2.2 [The Object Source File](#)
 - 20.3 [The Game Implementation](#)
 - 20.4 [What You Should See](#)
 - 20.5 [Auditing The Game](#)
 - 20.6 [Extra Credit](#)
- 21 [Exercise 20: Zed's Awesome Debug Macros](#)
 - 21.1 [The C Error Handling Problem](#)
 - 21.2 [The Debug Macros](#)
 - 21.3 [Using dbg.h](#)
 - 21.4 [What You Should See](#)
 - 21.5 [How The CPP Expands Macros](#)
 - 21.6 [Extra Credit](#)
- 22 [Exercise 21: Advanced Data Types And Flow Control](#)
 - 22.1 [Available Data Types](#)
 - 22.1.1 [Type Modifiers](#)
 - 22.1.2 [Type Qualifiers](#)
 - 22.1.3 [Type Conversion](#)
 - 22.1.4 [Type Sizes](#)
 - 22.2 [Available Operators](#)
 - 22.2.1 [Math Operators](#)
 - 22.2.2 [Data Operators](#)
 - 22.2.3 [Logic Operators](#)
 - 22.2.4 [Bit Operators](#)
 - 22.2.5 [Boolean Operators](#)
 - 22.2.6 [Assignment Operators](#)
 - 22.3 [Available Control Structures](#)
 - 22.3.1 [Extra Credit](#)
- 23 [Exercise 22: The Stack, Scope, And Globals](#)
 - 23.0.2 [ex22.h and ex22.c](#)
 - 23.0.3 [ex22_main.c](#)
 - 23.1 [What You Should See](#)
 - 23.2 [Scope, Stack, And Bugs](#)
 - 23.3 [How To Break It](#)
 - 23.4 [Extra Credit](#)
- 24 [Exercise 23: Meet Duff's Device](#)
 - 24.1 [What You Should See](#)
 - 24.2 [Solving The Puzzle](#)
 - 24.2.1 [Why Bother?](#)

- 24.3 [Extra Credit](#)
- 25 [Exercise 24: Input, Output, Files](#)
 - 25.1 [What You Should See](#)
 - 25.2 [How To Break It](#)
 - 25.3 [The I/O Functions](#)
 - 25.4 [Extra Credit](#)
- 26 [Exercise 25: Variable Argument Functions](#)
 - 26.1 [What You Should See](#)
 - 26.2 [How To Break It](#)
 - 26.3 [Extra Credit](#)
- 27 [Exercise 26: Write A First Real Program](#)
 - 27.1 [What Is devpkg?](#)
 - 27.1.1 [What We Want To Make](#)
 - 27.1.2 [The Design](#)
 - 27.1.3 [The Apache Portable Runtime](#)
 - 27.2 [Project Layout](#)
 - 27.2.1 [Other Dependencies](#)
 - 27.3 [The Makefile](#)
 - 27.4 [The Source Files](#)
 - 27.4.1 [The DB Functions](#)
 - 27.4.2 [The Shell Functions](#)
 - 27.4.3 [The Command Functions](#)
 - 27.4.4 [The devpkg Main Function](#)
 - 27.5 [The Mid-Term Exam](#)
- II [Data Structures And Algorithms](#)
- 28 [Exercise 27: Creative And Defensive Programming](#)
 - 28.1 [The Creative Programmer Mindset](#)
 - 28.2 [The Defensive Programmer Mindset](#)
 - 28.3 [The Eight Defensive Programmer Strategies](#)
 - 28.4 [Applying The Eight Strategies](#)
 - 28.4.1 [Never Trust Input](#)
 - 28.4.2 [Prevent Errors](#)
 - 28.4.3 [Fail Early And Openly](#)
 - 28.4.4 [Document Assumptions](#)
 - 28.4.5 [Prevention Over Documentation](#)
 - 28.4.6 [Automate Everything](#)
 - 28.4.7 [Simplify And Clarify](#)
 - 28.4.8 [Question Authority](#)
 - 28.5 [Order Is Not Important](#)
 - 28.6 [Extra Credit](#)
- 29 [Exercise 28: Intermediate Makefiles](#)
 - 29.1 [The Basic Project Structure](#)
 - 29.2 [Makefile](#)
 - 29.2.1 [The Header](#)
 - 29.2.2 [The Target Build](#)
 - 29.2.3 [The Unit Tests](#)
 - 29.2.4 [The Cleaner](#)

- [29.2.5 The Install](#)
 - [29.2.6 The Checker](#)
- [29.3 What You Should See](#)
- [29.4 Extra Credit](#)
- [30 Exercise 29: Libraries And Linking](#)
 - [30.0.1 Dynamically Loading A Shared Library](#)
- [30.1 What You Should See](#)
- [30.2 How To Break It](#)
- [30.3 Extra Credit](#)
- [31 Exercise 30: Automated Testing](#)
 - [31.1 Wiring Up The Test Framework](#)
 - [31.2 Extra Credit](#)
- [32 Exercise 31: Debugging Code](#)
 - [32.1 Debug Printing Vs. GDB Vs. Valgrind](#)
 - [32.2 A Debugging Strategy](#)
 - [32.3 Using GDB](#)
 - [32.4 Process Attaching](#)
 - [32.5 GDB Tricks](#)
 - [32.6 Extra Credit](#)
- [33 Exercise 32: Double Linked Lists](#)
 - [33.1 What Are Data Structures](#)
 - [33.2 Making The Library](#)
 - [33.3 Double Linked Lists](#)
 - [33.3.1 Definition](#)
 - [33.3.2 Implementation](#)
 - [33.4 Tests](#)
 - [33.5 What You Should See](#)
 - [33.6 How To Improve It](#)
 - [33.7 Extra Credit](#)
- [34 Exercise 33: Linked List Algorithms](#)
 - [34.0.1 Bubble And Merge Sort](#)
 - [34.0.2 The Unit Test](#)
 - [34.0.3 The Implementation](#)
 - [34.1 What You Should See](#)
 - [34.2 How To Improve It](#)
 - [34.3 Extra Credit](#)
- [35 Exercise 34: Dynamic Array](#)
 - [35.1 Advantages And Disadvantages](#)
 - [35.2 How To Improve It](#)
 - [35.3 Extra Credit](#)
- [36 Exercise 35: Sorting And Searching](#)
 - [36.1 Radix Sort And Binary Search](#)
 - [36.1.1 C Unions](#)
 - [36.1.2 The Implementation](#)
 - [36.1.3 RadixMap_find And Binary Search](#)
 - [36.1.4 RadixMap_sort And radix_sort](#)
 - [36.2 How To Improve It](#)

- 36.3 [Extra Credit](#)
- 37 [Exercise 36: Safer Strings](#)
 - 37.1 [Why C Strings Were A Horrible Idea](#)
 - 37.2 [Using bstrlib](#)
 - 37.3 [Learning The Library](#)
- 38 [Exercise 37: Hashmaps](#)
 - 38.0.1 [The Unit Test](#)
 - 38.1 [How To Improve It](#)
 - 38.2 [Extra Credit](#)
- 39 [Exercise 38: Hashmap Algorithms](#)
 - 39.1 [What You Should See](#)
 - 39.2 [How To Break It](#)
 - 39.3 [Extra Credit](#)
- 40 [Exercise 39: String Algorithms](#)
 - 40.1 [What You Should See](#)
 - 40.2 [Analyzing The Results](#)
 - 40.3 [Extra Credit](#)
- 41 [Exercise 40: Binary Search Trees](#)
 - 41.1 [How To Improve It](#)
 - 41.2 [Extra Credit](#)
- 42 [Exercise 41: Using Cachegrind And Callgrind For Performance Tuning](#)
 - 42.1 [Running Callgrind](#)
 - 42.2 [Callgrind Annotating Source](#)
 - 42.3 [Analyzing Memory Access With Cachegrind](#)
 - 42.4 [Judo Tuning](#)
 - 42.5 [Using KCachegrind](#)
 - 42.6 [Extra Credit](#)
- 43 [Exercise 42: Stacks and Queues](#)
 - 43.1 [What You Should See](#)
 - 43.2 [How To Improve It](#)
 - 43.3 [Extra Credit](#)
- 44 [Exercise 43: A Simple Statistics Engine](#)
 - 44.1 [Rolling Standard Deviation And Mean](#)
 - 44.2 [Implementation](#)
 - 44.3 [How To Use It](#)
 - 44.4 [Extra Credit](#)
- 45 [Exercise 44: Ring Buffer](#)
 - 45.1 [The Unit Test](#)
 - 45.2 [What You Should See](#)
 - 45.3 [How To Improve It](#)
 - 45.4 [Extra Credit](#)
- 46 [Exercise 45: A Simple TCP/IP Client](#)
 - 46.1 [Augment The Makefile](#)
 - 46.2 [The netclient Code](#)
 - 46.3 [What You Should See](#)
 - 46.4 [How To Break It](#)
 - 46.5 [Extra Credit](#)

- 47 [Exercise 46: Ternary Search Tree](#)
 - 47.1 [What You Should See](#)
 - 47.2 [How To Break It](#)
 - 47.3 [Extra Credit](#)
- 48 [Exercise 47: Ternary Search Tree Algorithms](#)
 - 48.1 [What You Should See](#)
 - 48.2 [How To Break It](#)
 - 48.3 [Extra Credit](#)
- 49 [Exercise 48: A Tiny Virtual Machine Part 1](#)
 - 49.1 [What You Should See](#)
 - 49.2 [How To Break It](#)
 - 49.3 [Extra Credit](#)
- 50 [Exercise 48: A Tiny Virtual Machine Part 2](#)
 - 50.1 [What You Should See](#)
 - 50.2 [How To Break It](#)
 - 50.3 [Extra Credit](#)
- 51 [Exercise 50: A Tiny Virtual Machine Part 3](#)
 - 51.1 [What You Should See](#)
 - 51.2 [How To Break It](#)
 - 51.3 [Extra Credit](#)
- 52 [Exercise 51: A Tiny Virtual Machine Part 4](#)
 - 52.1 [What You Should See](#)
 - 52.2 [How To Break It](#)
 - 52.3 [Extra Credit](#)
- 53 [Exercise 52: A Tiny Virtual Machine Part 5](#)
 - 53.1 [What You Should See](#)
 - 53.2 [How To Break It](#)
 - 53.3 [Extra Credit](#)
- 54 [Next Steps](#)
- III [Reviewing And Critiquing Code](#)
- 55 [Deconstructing "K&R C"](#)
 - 55.1 [An Overall Critique Of Correctness](#)
 - 55.1.1 [A First Demonstration Defect](#)
 - 55.1.2 [Why copy\(\) Fails](#)
 - 55.1.3 [But, That's Not A C String](#)
 - 55.1.4 [Just Don't Do That](#)
 - 55.1.5 [Stylistic Issues](#)
 - 55.2 [Chapter 1 Examples](#)

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Preface

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

This is a rough in-progress dump of the book. The grammar will probably be bad, there will be sections missing, but you get to watch me write the book and see how I do things.

There is a mailing list for the book at lcthw@librelist.com which you can join. I'll be doing announcements as new material is up, and you can ask questions if you get stuck or have comments.

This list is a discussion list, not an announce-only list. It's for discussing the book and asking questions.

Finally, don't forget that I have [Learn Python The Hard Way, 2nd Edition](#) which you should read if you can't code yet. LCTHW will not be for beginners, but for people who have at least read LPTHW or know one other programming language.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

Learn C The Hard Way

A Learn Code The Hard Way Book

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Introduction: The Cartesian Dream Of C

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Whatever I have up till now accepted as most true and assured I have gotten either from the senses or through the senses. But from time to time I have found that the senses deceive, and it is prudent never to trust completely those who have deceived us even once.

(Rene Descartes, Meditations On First Philosophy)

If there ever were a quote that described programming with C, it would be this. To many programmers, this makes C scary and evil. It is the Devil, Satan, the trickster Loki come to destroy your productivity with his seductive talk of pointers and direct access to the machine. Then, once this computational Lucifer has you hooked, he destroys your world with the evil "segfault" and laughs as he reveals the trickery in your bargain with him.

But, C is not to blame for this state of affairs. No my friends, your computer and the Operating System controlling it are the real tricksters. They conspire to hide their true inner workings from you so that you can never really know what is going on. The C programming language's only failing is giving you access to what is really there, and telling you the cold hard raw truth. C gives you the red pill. C pulls the curtain back to show you the wizard. C is truth.

Why use C then if it's so dangerous? Because C gives you power over the false reality of abstraction and liberates you from stupidity.

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 in it, or modify someone else's code. At the end of the book we actually take code from a more famous book called "K&R C" and code review it using what you've learned. To get to this stage you'll have to learn a few things:

1. The basics of C syntax and idioms.
2. Compilation, make files, linkers.
3. Finding bugs and preventing them.
4. Defensive coding practices.
5. Breaking C code.
6. Writing basic Unix systems software.

By the final chapter 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 [Learn](#)

[Python The Hard Way](#) or to [Learn Ruby The Hard Way](#) if you haven't learned a programming language yet. Those two books are for total beginners and work very well. Once you've done those 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 I have you coding right away and then I explain what you just did. This works better because it's easier to explain something you've already experienced.

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

1. Type in all of the code. Do not copy-paste!
2. Type the code in exactly, even the comments.
3. Get it to run and make sure it prints the same output.
4. If there are bugs fix them.
5. Do the extra credit but it's alright to skip ones you can't figure out.
6. 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 the act of trying will make you a better programmer.

The Core Competencies

I'm going to guess that you come from a language for weaklings ¹. One of those "usable" languages that lets you get away with sloppy thinking and half-assed 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 that you walk around in looking for clues. Even worse, I'm betting you've been using an IDE, so your brain is riddled with memory holes and you can't even type out an entire function's name without hitting CTRL-SPACE every 3 characters you type.

No matter what your background, you are probably bad at four skills:

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 skim code they need to understand in detail and think they understand it when they really don't. Other languages provide tools that also let them avoid actually writing any code, so when faced with a language like C they break down. Simplest thing to do is just understand 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 easy 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 is 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 people from other languages have is their brain has 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 is wrong.

Planning And Debugging

I love other easier languages because I can just hang out. I 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 plan out what you'll create first. Sure, you can hack for a bit, but you have to get serious much earlier in C than other languages. I'll be teaching you ways to plan out key parts of your program before you start coding, and this will hopefully 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 and half-assed about what you write or nothing will work. The advantage of C is it's a simple language you can figure out on your own, which makes it a great language for learning about the machine and getting stronger in these core programmer skills.

C is harder than some other languages, but that's only because C's not hiding things from you that those other languages try and fail to obfuscate.

License

This book is free for you to read, but until I'm done you can't distribute it or modify it. I need to make sure that unfinished copies of it do not get out and mess up a student on accident.

¹If you can't tell, I'm just teasing you.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Part I

Basic Skills

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

- 1 [Exercise 0: The Setup](#)
 - 1.1 [Linux](#)
 - 1.2 [Mac OSX](#)
 - 1.3 [Windows](#)
 - 1.4 [Text Editor](#)
- 2 [Exercise 1: Dust Off That Compiler](#)
 - 2.1 [What You Should See](#)
 - 2.2 [How To Break It](#)
 - 2.3 [Extra Credit](#)
- 3 [Exercise 2: Make Is Your Python Now](#)
 - 3.1 [Using Make](#)
 - 3.2 [What You Should See](#)
 - 3.3 [How To Break It](#)
 - 3.4 [Extra Credit](#)
- 4 [Exercise 3: Formatted Printing](#)
 - 4.1 [What You Should See](#)
 - 4.2 [External Research](#)
 - 4.3 [How To Break It](#)
 - 4.4 [Extra Credit](#)
- 5 [Exercise 4: Introducing Valgrind](#)
 - 5.1 [Installing Valgrind](#)
 - 5.2 [Using Valgrind](#)
 - 5.3 [What You Should See](#)
 - 5.4 [Extra Credit](#)
- 6 [Exercise 5: The Structure Of A C Program](#)
 - 6.1 [What You Should See](#)
 - 6.2 [Breaking It Down](#)
 - 6.3 [Extra Credit](#)
- 7 [Exercise 6: Types Of Variables](#)
 - 7.1 [What You Should See](#)
 - 7.2 [How To Break It](#)
 - 7.3 [Extra Credit](#)
- 8 [Exercise 7: More Variables, Some Math](#)
 - 8.1 [What You Should See](#)

- 8.2 [How To Break It](#)
- 8.3 [Extra Credit](#)
- 9 [Exercise 8: Sizes And Arrays](#)
 - 9.1 [What You Should See](#)
 - 9.2 [How To Break It](#)
 - 9.3 [Extra Credit](#)
- 10 [Exercise 9: Arrays And Strings](#)
 - 10.1 [What You Should See](#)
 - 10.2 [How To Break It](#)
 - 10.3 [Extra Credit](#)
- 11 [Exercise 10: Arrays Of Strings, Looping](#)
 - 11.1 [What You Should See](#)
 - 11.2 [How To Break It](#)
 - 11.3 [Extra Credit](#)
- 12 [Exercise 11: While-Loop And Boolean Expressions](#)
 - 12.1 [What You Should See](#)
 - 12.2 [How To Break It](#)
 - 12.3 [Extra Credit](#)
- 13 [Exercise 12: If, Else-If, Else](#)
 - 13.1 [What You Should See](#)
 - 13.2 [How To Break It](#)
 - 13.3 [Extra Credit](#)
- 14 [Exercise 13: Switch Statement](#)
 - 14.1 [What You Should See](#)
 - 14.2 [How To Break It](#)
 - 14.3 [Extra Credit](#)
- 15 [Exercise 14: Writing And Using Functions](#)
 - 15.1 [What You Should See](#)
 - 15.2 [How To Break It](#)
 - 15.3 [Extra Credit](#)
- 16 [Exercise 15: Pointers Dreaded Pointers](#)
 - 16.1 [What You Should See](#)
 - 16.2 [Explaining Pointers](#)
 - 16.3 [Practical Pointer Usage](#)
 - 16.4 [The Pointer Lexicon](#)
 - 16.5 [How To Break It](#)
 - 16.6 [Extra Credit](#)
- 17 [Exercise 16: Structs And Pointers To Them](#)
 - 17.1 [What You Should See](#)
 - 17.2 [Explaining Structures](#)
 - 17.3 [How To Break It](#)
 - 17.4 [Extra Credit](#)
- 18 [Exercise 17: Heap And Stack Memory Allocation](#)
 - 18.1 [What You Should See](#)
 - 18.2 [Heap vs. Stack Allocation](#)
 - 18.3 [How To Break It](#)
 - 18.4 [Extra Credit](#)

[19 Exercise 18: Pointers To Functions](#)

[19.1 What You Should See](#)

[19.2 How To Break It](#)

[19.3 Extra Credit](#)

[20 Exercise 19: A Simple Object System](#)

[20.1 How The CPP Works](#)

[20.2 The Prototype Object System](#)

[20.3 The Game Implementation](#)

[20.4 What You Should See](#)

[20.5 Auditing The Game](#)

[20.6 Extra Credit](#)

[21 Exercise 20: Zed's Awesome Debug Macros](#)

[21.1 The C Error Handling Problem](#)

[21.2 The Debug Macros](#)

[21.3 Using dbg.h](#)

[21.4 What You Should See](#)

[21.5 How The CPP Expands Macros](#)

[21.6 Extra Credit](#)

[22 Exercise 21: Advanced Data Types And Flow Control](#)

[22.1 Available Data Types](#)

[22.2 Available Operators](#)

[22.3 Available Control Structures](#)

[23 Exercise 22: The Stack, Scope, And Globals](#)

[23.1 What You Should See](#)

[23.2 Scope, Stack, And Bugs](#)

[23.3 How To Break It](#)

[23.4 Extra Credit](#)

[24 Exercise 23: Meet Duff's Device](#)

[24.1 What You Should See](#)

[24.2 Solving The Puzzle](#)

[24.3 Extra Credit](#)

[25 Exercise 24: Input, Output, Files](#)

[25.1 What You Should See](#)

[25.2 How To Break It](#)

[25.3 The I/O Functions](#)

[25.4 Extra Credit](#)

[26 Exercise 25: Variable Argument Functions](#)

[26.1 What You Should See](#)

[26.2 How To Break It](#)

[26.3 Extra Credit](#)

[27 Exercise 26: Write A First Real Program](#)

[27.1 What Is devpkg?](#)

[27.2 Project Layout](#)

[27.3 The Makefile](#)

[27.4 The Source Files](#)

[27.5 The Mid-Term Exam](#)

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 1

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 0: The Setup

In this chapter you get your system setup to do C programming. The good news for anyone using Linux or Mac OSX is that you are on a system designed for programming in C. The authors of the C language were also instrumental in the creation of the Unix operating system, and both Linux and OSX are based on Unix. In fact, the install will be incredibly easy.

I have some bad news for users of Windows: learning C on Windows is painful. You can write C code for Windows, that's not a problem. The problem is all of the libraries, functions, and tools are just a little "off" from everyone else in the C world. C came from Unix and is much easier on a Unix platform. It's just a fact of life that you'll have to accept I'm afraid.

I wanted to get this bad news out right away so that you don't panic. I'm not saying to avoid Windows entirely. I am however saying that, if you want to have the easiest time learning C, then it's time to bust out some Unix and get dirty. This could also be really good for you, since knowing a little bit of Unix will also teach you some of the idioms of C programming and expand your skills.

This also means that for everyone you'll be using the command line. Yep, I said it. You've gotta get in there and type commands at the computer. Don't be afraid though because I'll be telling you what to type and what it should look like, so you'll actually be learning quite a few mind expanding skills at the same time.

1.1 Linux

On most Linux systems you just have to install a few packages. For Debian based systems, like Ubuntu you should just have to install a few things using these commands:

Source 1: Installing Requirements On Ubuntu

```
1 $ sudo aptitude install build-essential
```

The above is an example of a command line prompt, so to get to where you can run that, find your "Terminal" program and run it first. Then you'll get a shell prompt similar to the '\$' above and can type that command into it. Do not type the '\$', just the stuff after it.

Here's how you would install the same setup on an RPM based Linux like Fedora:

Source 2: Installing Requirements On Fedora

```
1 $ su -c "yum groupinstall development-tools"
```

Once you've run that, you should be able to do the first Exercise in this book and it'll work. If not then let me know.

1.2 Mac OSX

On Mac OSX the install is even easier. First, you'll need to either download the latest XCode from Apple, or find your install DVD and install it from there. The download will be massive and could take forever, so I recommend installing from the DVD. Also, search online for "installing xcode" for instructions on how to do it.

Once you're done installing XCode, and probably restarting your computer if it didn't make you do that, you can go find your Terminal program and get it put into your Dock. You'll be using Terminal a lot in the book, so it's good to put it in a handy location.

1.3 Windows

For Windows users I'll show you how to get a basic Ubuntu Linux system up and running in a virtual machine so that you can still do all of my exercises, but avoid all the painful Linux installation problems.

... have to figure this one out.

1.4 Text Editor

The choice of text editor for a programmer is a tough one. For beginners I tell them to just use [Gedit](#) since it's simple and works for code. However, it doesn't work in certain internationalized situations, and chances are you already have a favorite text editor if you've been programming for a while.

With this in mind, I want you to try out a few of the standard programmer text editors for your platform and then stick with the one that you like best. If you've been using GEdit and like it then stick with it. If you want to try something different, then try it out real quick and pick one.

The most important thing is do not get stuck picking the perfect editor. Text editors all just kind of suck in odd ways. Just pick one, stick with it, and if you find something else you like try it out. Don't spend days on end configuring it and making it perfect.

Some text editors to try out are:

1. [Gedit](#) on Linux and OSX.
2. [TextWrangler](#) on OSX.
3. [Nano](#) which runs in Terminal and works nearly everywhere.
4. [Emacs](#) and [Emacs for OSX](#). Be prepared to do some learning though.
5. [Vim](#) and [MacVim](#)

There is probably a different editor for every person out there, but these are just a few of the free ones that I know work. Try a few out, and maybe some commercial ones until you find one that you like.

1.4.1 WARNING: Do Not Use An IDE

An IDE, or "Integrated Development Environment" will turn you stupid. They are the worst tools if you want to be a good programmer because they hide what's going on from you, and your job is to know what's going on. They are useful if you're trying to get something done and the platform is designed around a particular IDE, but for learning to code C (and

many other languages) they are pointless.

Note 1: IDEs and Guitar Tablature

If you've played guitar then you know what tablature is, but for everyone else let me explain. In music there's an established notation called the "staff notation". It's a generic, very old, and universal way to write down what someone should play on an instrument. If you play piano this notation is fairly easy to use, since it was created mostly for piano and composers.

Guitar however is a weird instrument that doesn't really work with notation, so guitarists have an alternative notation called "tablature". What tablature does is, rather than tell you the note to play, it tells you the fret and string you should play at that time. You could learn whole songs without ever knowing about a single thing you're playing. Many people do it this way, but if you want to know what you're playing, then tablature is pointless.

It may be harder than tablature, but traditional notation tells you how to play the music rather than just how to play the guitar. With traditional notation I can walk over to a piano and play the same song. I can play it on a bass. I can put it into a computer and design whole scores around it. With tablature I can just play it on a guitar.

IDEs are like tablature. Sure, you can code pretty quickly, but you can only code in that one language on that one platform. This is why companies love selling them to you. They know you're lazy, and since it only works on their platform they've got you locked in because you are lazy.

The way you break the cycle is you suck it up and finally learn to code without an IDE. A plain editor, or a programmer's editor like Vim or Emacs, makes you work with the code. It's a little harder, but the end result is you can work with any code, on any computer, in any language, and you know what's going on.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 2

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 1: Dust Off That Compiler

Here is a simple first program you can make in C:

Source 3: ex1.c

```
1 int main()
2 {
3     puts("Hello world.");
4
5     return 0;
6 }
```

You can put this into a ex1.c then type:

Source 4: Building ex1

```
1 $ make ex1
2 cc ex1.c -o ex1
```

Your computer may use a slightly different command, but the end result should be a file named ex1 that you can run.

2.1 What You Should See

You can now run the program and see the output.

Source 5: Running ex1

```
1 $ ./ex1
2 Hello world.
```

If you don't then go back and fix it.

2.2 How To Break It

In this book I'm going to have a small section for each program on how to break the program. 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, rebuild it with all compiler warnings on:

Source 6: Building ex1 with -Wall

```

1  $ rm ex1
2  $ CFLAGS="-Wall" make ex1
3  cc -Wall  ex1.c  -o ex1
4  ex1.c: In function 'main':
5  ex1.c:3: warning: implicit declaration of function 'puts'
6  $ ./ex1
7  Hello world.
8  $

```

Now you are getting a warning that says the function "puts" is implicitly declared. The C compiler is smart enough to figure out what you want, but you should be getting rid of all compiler warnings when you can. How you do this is add the following line to the top of ex1.c and recompile:

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

Now do the make again like you just did and you'll see the warning go away.

2.3 Extra Credit

1. Open the ex1 file in your text editor and change or delete random parts. Try running it and see what happens.
2. Print out 5 more lines of text or something more complex than hello world.
3. Run man 3 puts and read about this function and many others.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 3

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 2: Make Is Your Python Now

In [Python](#) you ran programs by just typing python and the code you wanted to run. The Python interpreter would just run them, and import any other libraries and things you needed on the fly as it ran. C is a different beast completely where you have to compile your source files and manually stitch them together into a binary that can run on its own. Doing this manually is a pain, and in the last exercise you just ran make to do it.

In this exercise, you're going to get a crash course in GNU make, and you'll be learning to use it as you learn C. Make will for the rest of this book, be your Python. It will build your code, and run your tests, and set things up and do all the stuff for you that Python normally does.

The difference is, I'm going to show you smarter Makefile wizardry, where you don't have to specify every stupid little thing about your C program to get it to build. I won't do that in this exercise, but after you've been using "baby make" for a while, I'll show you "master make".

3.1 Using Make

The first stage of using make is to just use it to build programs it already knows how to build. Make has decades of knowledge on building a wide variety of files from other files. In the last exercise you did this already using commands like this:

Source 7: Building ex1 with -Wall

```
1 $ make ex1
2 # or this one too
3 $ CFLAGS="-Wall" make ex1
```

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

1. Does the file ex1 exist already?
2. No. Ok, 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" which 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 that way of using make, but let's get into making a Makefile so you can understand make a little better. To start off, create a file with just this in it:

Source 8: A simple Makefile

```
1 CFLAGS=-Wall -g
2
3 clean:
4     rm -f ex1
```

Save this file as `Makefile` in your current directory. Make automatically assumes there's a file called `Makefile` and will just run it. Also, 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` which 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:

Source 9: Running a simple Makefile

```
1 $ make clean
2 $ make ex1
```

3.2 What You Should See

If that worked then you should see this:

Source 10: Full build with Makefile

```
1 $ make clean
2 rm -f ex1
3 $ make ex1
4 cc -Wall -g ex1.c -o ex1
5 ex1.c: In function 'main':
6 ex1.c:3: warning: implicit declaration of function 'puts'
7 $
```

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 I indent and then I put 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.

Note 2: Did You Fix `ex1.c`?

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

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

3.3 How To Break It

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

Source 11: Bad make run

```
1 $ make clean
2 Makefile:4: *** missing separator. Stop.
```

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

3.4 Extra Credit

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

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 4

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 3: Formatted Printing

Keep that Makefile around since it'll help you spot errors and we'll be adding to it when we need to automate more things.

Many programming languages use the C way of formatting output, so let's try it:

Source 12: ex3.c

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int age = 10;
6      int height = 72;
7
8      printf("I am %d years old.\n", age);
9      printf("I am %d inches tall.\n", height);
10
11     return 0;
12 }
```

Once you have that, do the usual make ex3 to build it and run it. Make sure you fix all warnings.

This exercise has a whole lot going on in a small amount of code so let's break it down:

1. First you're including another "header file" called `stdio.h`. This tells the compiler that you're going to use the "standard Input/Output functions". One of those is `printf`.
2. Then you're using a variable named `age` and setting it to 10.
3. Next you're using a variable `height` and setting it to 72.
4. Then you use the `printf` function to print the age and height of the tallest 10 year old on the planet.
5. In the `printf` you'll notice you're passing in a string, and it's a format string like in many other languages.
6. After this format string, you put the variables that should be "replaced" into the format string by `printf`.

The result of doing this is you are handing `printf` some variables and it is constructing a new string then printing that new string to the terminal.

4.1 What You Should See

When you do the whole build you should see something like this:

Source 13: Building and running ex3.c

```
1  $ make ex3
```

```

2 cc -Wall -g ex3.c -o ex3
3 $ ./ex3
4 I am 10 years old.
5 I am 72 inches tall.
6 $

```

Pretty soon I'm going to stop telling you to run make and what the build looks like, so please make sure you're getting this right and that it's working.

4.2 External Research

In the Extra Credit section of each exercise I may have you go find information on your own and figure things out. This is an important part of being a self-sufficient programmer. If you constantly run to ask someone a question before trying to figure it out first then you never learn to solve problems independently. This leads to you never building confidence in your skills and always needing someone else around to do your work.

The way you break this habit is to force yourself to try to answer your own questions first, and to confirm that your answer is right. You do this by trying to break things, experimenting with your possible answer, and doing your own research.

For this exercise I want you to go online and find out all of the printf escape codes and format sequences. Escape codes are `\n` or `\t` that let you print a newline or tab (respectively). Format sequences are the `%s` or `%d` that let you print a string or a integer. Find all of the ones available, how you can modify them, and what kind of "precisions" and widths you can do.

From now on, these kinds of tasks will be in the Extra Credit and you should do them.

4.3 How To Break It

Try a few of these ways to break this program, which may or may not cause it to crash on your computer:

1. Take the age variable out of the first printf call then recompile. You should get a couple of warnings.
2. Run this new program and it will either crash, or print out a really crazy age.
3. Put the printf back the way it was, and then don't set age to an initial value by changing that line to `int age`; then rebuild and run again.

Source 14: Breaking ex3.c

```

1 # edit ex3.c to break printf
2 $ make ex3
3 cc -Wall -g ex3.c -o ex3
4 ex3.c: In function 'main':
5 ex3.c:8: warning: too few arguments for format
6 ex3.c:5: warning: unused variable 'age'
7 $ ./ex3
8 I am -919092456 years old.
9 I am 72 inches tall.
10 # edit ex3.c again to fix printf, but don't init age
11 $ make ex3
12 cc -Wall -g ex3.c -o ex3
13 ex3.c: In function 'main':
14 ex3.c:8: warning: 'age' is used uninitialized in this function
15 $ ./ex3
16 I am 0 years old.
17 I am 72 inches tall.
18 $

```

4.4 Extra Credit

1. Find as many other ways to break `ex3.c` as you can.
2. Run `man 3 printf` and read about the other `'%'` format characters you can use. These should look familiar if you used them in other languages (`printf` is where they come from).
3. Add `ex3` to your Makefile's all list. Use this to make clean all and build all your exercises so far.
4. Add `ex3` to your Makefile's clean list as well. Now use `make clean` will remove it when you need to.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 5

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 4: Introducing Valgrind

It's time to learn about another tool you will live and die by as you learn C called Valgrind. I'm introducing Valgrind to you now because you're going to use it from now on in the "How To Break It" sections of each exercise. Valgrind is a program that runs your programs, and then reports on all of the horrible mistakes you made. It's a wonderful free piece of software that I use constantly while I write C code.

Remember in the last exercise that I told you to break your code by removing one of the arguments to printf? It printed out some funky results, but I didn't tell you why it printed those results out. In this exercise we're going to use Valgrind to find out why.

Note 3: What's With All The Tools

These first few exercises are mixing some essential tools the rest of the book needs with learning a little bit of code. The reason is that most of the folks who read this book are not familiar with compiled languages, and definitely not with automation and helpful tools. By getting you to use make and Valgrind right now I can then use them to teach you C faster and help you find all your bugs early.

After this exercise we won't do many more tools, it'll be mostly code and syntax for a while. But, we'll also have a few tools we can use to really see what's going on and get a good understanding of common mistakes and problems.

5.1 Installing Valgrind

You could install Valgrind with the package manager for your OS, but I want you to learn to install things from source. This involves the following process:

1. Download a source archive file to get the source.
2. Unpack the archive to extract the files onto your computer.
3. Run ./configure to setup build configurations.
4. Run make to make it build, just like you've been doing.
5. Run sudo make install to install it onto your computer.

Here's a script of me doing this very process, which I want you to try to replicate:

Source 15: ex4.sh

```

1 # 1) Download it (use wget if you don't have curl)
2 curl -O http://valgrind.org/downloads/valgrind-3.6.1.tar.bz2
3
4 # use md5sum to make sure it matches the one on the site
5 md5sum valgrind-3.6.1.tar.bz2
6
7 # 2) Unpack it.
```



```

8  tar -xjvf valgrind-3.6.1.tar.bz2
9
10 # cd into the newly created directory
11 cd valgrind-3.6.1
12
13 # 3) configure it
14 ./configure
15
16 # 4) make it
17 make
18
19 # 5) install it (need root)
20 sudo make install

```

Follow this, but obviously update it for new Valgrind versions. If it doesn't build then try digging into why as well.

5.2 Using Valgrind

Using Valgrind is easy, you just run valgrind theprogram and it runs your program, then prints out all the errors your program made while it was running. In this exercise we'll break down one of the error outputs and you can get an instant crash course in "Valgrind hell". Then we'll fix the program.

First, here's a purposefully broken version of the ex3.c code for you to build, now called ex4.c. For practice, type it in again:

Source 16: ex4.c

```

1  #include <stdio.h>
2
3  /* Warning: This program is wrong on purpose. */
4
5  int main()
6  {
7      int age = 10;
8      int height;
9
10     printf("I am %d years old.\n");
11     printf("I am %d inches tall.\n", height);
12
13     return 0;
14 }

```

You'll see it's the same except I've made two classic mistakes:

1. I've failed to initialize the height variable.
2. I've forgot to give the first printf the age variable.

5.3 What You Should See

Now we will build this just like normal, but instead of running it directly, we'll run it with Valgrind (see Source: "Building and running ex4.c with Valgrind"):

Source 17: Building and running ex4.c with Valgrind

```

1  $ make ex4
2  cc -Wall -g  ex4.c  -o ex4
3  ex4.c: In function 'main':
4  ex4.c:10: warning: too few arguments for format
5  ex4.c:7: warning: unused variable 'age'
6  ex4.c:11: warning: 'height' is used uninitialized in this function
7  $ valgrind ./ex4
8  ==3082== Memcheck, a memory error detector
9  ==3082== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
10 ==3082== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info

```

```

11 ==3082== Command: ./ex4
12 ==3082==
13 I am -16775432 years old.
14 ==3082== Use of uninitialised value of size 8
15 ==3082== at 0x4E730EB: _itoa_word (_itoa.c:195)
16 ==3082== by 0x4E743D8: fprintf (fprintf.c:1613)
17 ==3082== by 0x4E7E6F9: printf (printf.c:35)
18 ==3082== by 0x40052B: main (ex4.c:11)
19 ==3082==
20 ==3082== Conditional jump or move depends on uninitialised value(s)
21 ==3082== at 0x4E730F5: _itoa_word (_itoa.c:195)
22 ==3082== by 0x4E743D8: fprintf (fprintf.c:1613)
23 ==3082== by 0x4E7E6F9: printf (printf.c:35)
24 ==3082== by 0x40052B: main (ex4.c:11)
25 ==3082==
26 ==3082== Conditional jump or move depends on uninitialised value(s)
27 ==3082== at 0x4E7633B: fprintf (fprintf.c:1613)
28 ==3082== by 0x4E7E6F9: printf (printf.c:35)
29 ==3082== by 0x40052B: main (ex4.c:11)
30 ==3082==
31 ==3082== Conditional jump or move depends on uninitialised value(s)
32 ==3082== at 0x4E744C6: fprintf (fprintf.c:1613)
33 ==3082== by 0x4E7E6F9: printf (printf.c:35)
34 ==3082== by 0x40052B: main (ex4.c:11)
35 ==3082==
36 I am 0 inches tall.
37 ==3082==
38 ==3082== HEAP SUMMARY:
39 ==3082== in use at exit: 0 bytes in 0 blocks
40 ==3082== total heap usage: 0 allocs, 0 frees, 0 bytes allocated
41 ==3082==
42 ==3082== All heap blocks were freed -- no leaks are possible
43 ==3082==
44 ==3082== For counts of detected and suppressed errors, rerun with: -v
45 ==3082== Use --track-origins=yes to see where uninitialised values come from
46 ==3082== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 4 from 4)
47 $

```

This one is huge because Valgrind is telling you exactly where every problem in your program is. Starting at the top here's what you're reading, line by line (line numbers are on the left so you can follow):

1

You do the usual make ex4 and that builds it. Make sure the cc command you see is the same and has the -g option or your Valgrind output won't have line numbers.

2-6

Notice that the compiler is also yelling at you about this source file and it warns you that you have "too few arguments for format". That's where you forgot to include the age variable.

7

Then you run your program using valgrind ./ex4.

8

Then Valgrind goes crazy and yells at you for:

14-18

On line main (ex4.c:11) (read as "in the main function in file ex4.c at line 11) you have "Use of uninitialised value of size 8". You find this by looking at the error, then you see what's called a "stack trace" right under that. The line to look at first (ex4.c:11) is the bottom one, and if you don't see what's going wrong then you go up, so you'd try printf.c:35. Typically it's the bottom most line that matters (in this case, on line 18).

20-24

Next error is yet another one on line ex4.c:11 in the main function. Valgrind hates this line. This error says that some kind of if-statement or while-loop happened that was based on an uninitialized variable, in this case height.

25-35

The remaining errors are more of the same because the variable keeps getting used.

37-46

Finally the program exits and Valgrind tells you a summary of how bad your program is.

That is quite a lot of information to take in, but here's how you deal with it:

1. Whenever you run your C code and get it working, rerun it under Valgrind to check it.
2. For each error that you get, go to the source:line indicated and fix it. You may have to search online for the error message to figure out what it means.
3. Once your program is "Valgrind pure" then it should be good, and you have probably learned something about how you write code.

In this exercise I'm not expecting you to fully grasp Valgrind right away, but instead get it installed and learn how to use it real quick so we can apply it to all the later exercises.

5.4 Extra Credit

1. Fix this program using Valgrind and the compiler as your guide.
2. Read up on Valgrind on the internet.
3. Download other software and build it by hand. Try something you already use but never built for yourself.
4. Look at how the Valgrind source files are laid out in the source directory and read its Makefile. Don't worry, none of that makes sense to me either.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 6

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 5: The Structure Of A C Program

You know how to use printf and have a couple basic tools at your disposal, so let's break down a simple C program line-by-line so you know how one is structured. In this program you're going to type in a few more things that you're unfamiliar with, and I'm going to lightly break them down. Then in the next few exercises we're going to work with these concepts.

Source 18: ex5.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 }
```

Type this code in, make it run, and make sure you get no Valgrind errors. You probably won't but get in the habit of checking it.

6.1 What You Should See

This has pretty boring output, but the point of this exercise is to analyze the code:

Source 19: ex5 output

```
1  $ make ex5
2  cc -Wall -g  ex5.c  -o ex5
3  $ ./ex5
4  You are 100 miles away.
5  $
```

6.2 Breaking It Down

There's a few features of the C language in this code that you might have only slightly figured out while you were typing code. Let's break this down line-by-line quickly, and then we can do exercises to understand each part better:

ex5.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 then contain lists of functions you want to use in your program.

ex5.c:3

This is a multi-line comment and you could put as many lines of text between the `/*` and closing `*/` characters as you want.

ex5.c:4

A more complex version of the main function you've been using blindly so far. How C programs work is the operating system loads your program, and then 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? Do not worry, we'll cover this soon.

ex5.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.

ex5.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.

ex5.c:8

Another kind of comment, and it works like Python or Ruby comments where it starts at the `//` and goes until the end of the line.

ex5.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. The `printf` function is actually kind of weird and can take multiple arguments. We'll see that later.

ex5.c:11

A return from the main function, which gives the OS your exit value. You may not be familiar with how Unix software uses return codes, so we'll cover that as well.

ex5.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 break-down, so study it line-by-line and make sure you at least have a little grasp of what's going on. You won't know everything, but you can probably guess before we continue.

6.3 Extra Credit

1. 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 that you can use to check later and see if you get it right.
2. Go back to the source code from the previous exercises and do a similar break-down to see if you're getting it. Write down what you don't know and can't explain to yourself.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 7

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 6: Types Of Variables

You should be getting a grasp of how a simple C program is structured, so let's do the next simplest thing which is making some variables of different types:

Source 20: ex6.c

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int distance = 100;
6      float power = 2.345f;
7      double super_power = 56789.4532;
8      char initial = 'A';
9      char first_name[] = "Zed";
10     char last_name[] = "Shaw";
11
12     printf("You are %d miles away.\n", distance);
13     printf("You have %f levels of power.\n", power);
14     printf("You have %f awesome super powers.\n", super_power);
15     printf("I have an initial %c.\n", initial);
16     printf("I have a first name %s.\n", first_name);
17     printf("I have a last name %s.\n", last_name);
18     printf("My whole name is %s %c. %s.\n",
19           first_name, initial, last_name);
20
21     return 0;
22 }
```

In this program we're declaring variables of different types and then printing them with different printf format strings.

7.1 What You Should See

Your output should look like mine, and you can start to see how the format strings for C are similar to Python and other languages. They've been around for a long time.

Source 21: ex6 output

```

1  $ make ex6
2  cc -Wall -g  ex6.c  -o ex6
3  $ ./ex6
4  You are 100 miles away.
5  You have 2.345000 levels of power.
6  You have 56789.453200 awesome super powers.
7  I have an initial A.
8  I have a first name Zed.
9  I have a last name Shaw.
10 My whole name is Zed A. Shaw.
11 $
```


What you can see is we have a set of "types", which are ways of telling the C compiler what each variable should represent, and then format strings to match different types. Here's the breakdown of how they match up:

Integers

You declare Integers with the `int` keyword, and print them with `%d`.

Floating Point

Declared with `float` or `double` depending on how big they need to be (`double` is bigger), and printed with `%f`.

Character

Declared with `char`, written with a ' (single-quote) character around the char, and then printed with `%c`.

String (Array of Characters)

Declared with `char name[]`, written with " characters, and printed with `%s`.

You'll notice that C makes a distinction between single-quote for char and double-quote for `char[]` or strings.

Note 4: C Type Short-Hand For English
 When talking about C types, I will typically write in English `char[]` instead of the whole `char SOMENAME[]`. This is not valid C code, just a simpler way to talk about types when writing English.

7.2 How To Break It

You can easily break this program by passing the wrong thing to the `printf` statements. For example, if you take the line that prints my name, but put the initial variable before the `first_name` in the arguments, you'll get a bug. Make that change and the compiler will yell at you, then when you run it you might get a "Segmentation fault" like I did:

Source 22: ex6 explosion

```

1  $ make ex6
2  cc -Wall -g  ex6.c  -o ex6
3  ex6.c: In function 'main':
4  ex6.c:19: warning: format '%s' expects type 'char *', but argument 2 has type 'int'
5  ex6.c:19: warning: format '%c' expects type 'int', but argument 3 has type 'char *'
6  $ ./ex6
7  You are 100 miles away.
8  You have 2.345000 levels of power.
9  You have 56789.453125 awesome super powers.
10 I have an initial A.
11 I have a first name Zed.
12 I have a last name Shaw.
13 Segmentation fault
14 $
    
```

Run this change under Valgrind too to see what it tells you about the error "Invalid read of size 1".

7.3 Extra Credit

1. Come up with other ways to break this C code by changing the `printf`, then fix them.
2. Go search for "printf formats" and try using a few of the more exotic ones.
3. Research how many different ways you can write a number. Try octal, hexadecimal, and others you can find.
4. Try printing an empty string that's just "".

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 8

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 7: More Variables, Some Math

Let's get familiar with more things you can do with variables by declaring various ints, floats, chars, and doubles. We'll then use these in various math expressions so you get introduced to C's basic math.

Source 23: ex7.c

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int bugs = 100;
6      double bug_rate = 1.2;
7
8      printf("You have %d bugs at the imaginary rate of %f.\n",
9             bugs, bug_rate);
10
11     long universe_of_defects = 1L * 1024L * 1024L * 1024L;
12     printf("The entire universe has %ld bugs.\n",
13            universe_of_defects);
14
15     double expected_bugs = bugs * bug_rate;
16     printf("You are expected to have %f bugs.\n",
17            expected_bugs);
18
19     double part_of_universe = expected_bugs / universe_of_defects;
20     printf("That is only a %e portion of the universe.\n",
21            part_of_universe);
22
23     // this makes no sense, just a demo of something weird
24     char nul_byte = '\0';
25     int care_percentage = bugs * nul_byte;
26     printf("Which means you should care %d%.\n",
27            care_percentage);
28
29     return 0;
30 }
```

Here's what's going on in this little bit of nonsense:

ex7.c:1-4

The usual start of a C program.

ex7.c:5-6

Declare an int and double for some fake bug data.

ex7.c:8-9

Print out those two, so nothing new here.

ex7.c:11

Declare a huge number using a new type long for storing big numbers.

ex7.c:12-13

Print out that number using `%ld` which adds a modifier to the usual `%d`. Adding 'l' (the letter ell) means "print this as a long decimal".

ex7.c:15–17

Just more math and printing.

ex7.c:19–21

Craft up a depiction of your bug rate compared to the bugs in the universe, which is a completely inaccurate calculation. It's so small though that we have to use `%e` to print it in scientific notation.

ex7.c:24

Make a character, with a special syntax `'\0'` which creates a 'nul byte' character. This is effectively the number 0.

ex7.c:25

Multiply bugs by this character, which produces 0 for how much you should care. This demonstrates an ugly hack you find sometimes.

ex7.c:26–27

Print that out, and notice I've got a `%%` (two percent chars) so I can print a `'%` (percent) character.

ex7.c:28–30

The end of the main function.

This bit of source is entirely just an exercise, and demonstrates how some math works. At the end, it also demonstrates something you see in C, but not in many other languages. To C, a "character" is just an integer. It's a really small integer, but that's all it is. This means you can do math on them, and a lot of software does just that, for good or bad.

This last bit is your first glance at how C gives you direct access to the machine. We'll be exploring that more in later exercises.

8.1 What You Should See

As usual, here's what you should see for the output:

Source 24: ex7 output

```
1  $ make ex7
2  cc -Wall -g  ex7.c  -o ex7
3  $ ./ex7
4  You have 100 bugs at the imaginary rate of 1.200000.
5  The entire universe has 1073741824 bugs.
6  You are expected to have 120.000000 bugs.
7  That is only a 1.117587e-07 portion of the universe.
8  Which means you should care 0%.
9  $
```

8.2 How To Break It

Again, go through this and try breaking the `printf` by passing in the wrong arguments. See what happens when you try to print out that `nul_byte` variable too with `%s` vs. `%c`. When you break it, run it under Valgrind to see what it says about your breaking attempts.

8.3 Extra Credit

1. Make the number you assign to `universe_of_defects` various sizes until you get a warning from the compiler.
2. What do these really huge numbers actually print out?
3. Change `long` to `unsigned long` and try to find the number that makes that one too big.

4. Go search online to find out what unsigned does.
5. Try to explain to yourself (before I do in the next exercise) why you can multiply a char and an int.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 9

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 8: Sizes And Arrays

In the last exercise you did math, but with a '\0' (nul) character. This may be odd coming from other languages, since they try to treat "strings" and "byte arrays" as different beasts. C however treats strings as just arrays of bytes, and it's only the different printing functions that know there's a difference.

Before I can really explain the significance of this, I have to introduce a few more concepts: sizeof and arrays. Here's the code we'll be talking about:

Source 25: ex8.c

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int areas[] = {10, 12, 13, 14, 20};
6      char name[] = "Zed";
7      char full_name[] = {
8          'Z', 'e', 'd',
9          ' ', 'A', ' ',
10         'S', 'h', 'a', 'w', '\0'
11     };
12
13     // WARNING: On some systems you may have to change the
14     // %ld in this code to a %u since it will use unsigned ints
15     printf("The size of an int: %ld\n", sizeof(int));
16     printf("The size of areas (int[]): %ld\n",
17         sizeof(areas));
18     printf("The number of ints in areas: %ld\n",
19         sizeof(areas) / sizeof(int));
20     printf("The first area is %d, the 2nd %d.\n",
21         areas[0], areas[1]);
22
23     printf("The size of a char: %ld\n", sizeof(char));
24     printf("The size of name (char[]): %ld\n",
25         sizeof(name));
26     printf("The number of chars: %ld\n",
27         sizeof(name) / sizeof(char));
28
29     printf("The size of full_name (char[]): %ld\n",
30         sizeof(full_name));
31     printf("The number of chars: %ld\n",
32         sizeof(full_name) / sizeof(char));
33
34     printf("name=\"%s\" and full_name=\"%s\"\n",
35         name, full_name);
36
37     return 0;
38 }
```

In this code we create a few arrays with different data types in them. Because arrays of data are so central to how C works, there's a huge number of ways to create them. For now, just use the syntax `type name[] = {initializer};` and we'll explore more. What this syntax means is, "I want an array of type that is initialized to ..." When C sees this it does the

following:

1. Look at the type, in this first case it's int.
2. Look at the [] and see that there's no length given.
3. Look at the initializer, {10, 12, 13, 14, 20} and figure out that you want those 5 ints in your array.
4. Create a piece of memory in the computer, that can hold 5 integers one after another.
5. Take the name you want, areas and assign it this location.

In the case of areas it's creating an array of 5 ints that contain those numbers. When it gets to char name[] = "Zed"; it's doing the same thing, except it's creating an array of 3 chars and assigning that to name. The final array we make is full_name, but we use the annoying syntax of spelling it out, one character at a time. To C, name and full_name are identical methods of creating a char array.

The rest of the file, we're using a keyword called sizeof to ask C how big things are in bytes. C is all about the size and location of pieces of memory and what you do with them. To help you keep that straight, it gives you sizeof so you can ask how big something is before you work with it.

This is where stuff gets tricky, so first let's run this and then explain further.

9.1 What You Should See

Source 26: ex8 output

```

1  $ make ex8
2  cc -Wall -g  ex8.c  -o ex8
3  $ ./ex8
4  The size of an int: 4
5  The size of areas (int[]): 20
6  The number of ints in areas: 5
7  The first area is 10, the 2nd 12.
8  The size of a char: 1
9  The size of name (char[]): 4
10 The number of chars: 4
11 The size of full_name (char[]): 12
12 The number of chars: 12
13 name="Zed" and full_name="Zed A. Shaw"
14 $

```

Now you see the output of these different printf calls and start to get a glimpse of what C is doing. Your output could actually be totally different from mine, since your computer might have different size integers. I'll go through my output:

5

My computer thinks an int is 4 bytes in size. Your computer might use a different size if it's a 32-bit vs. 64-bit.

6

The areas array has 5 integers in it, so it makes sense that my computer requires 20 bytes to store it.

7

If we divide the size of areas by size of an int then we get 5 elements. Looking at the code, this matches what we put in the initializer.

8

We then did an array access to get areas[0] and areas[1] which means C is "zero indexed" like Python and Ruby.

9-11

We repeat this for the name array, but notice something odd about the size of the array? It says it's 4 bytes long, but we only typed "Zed" for 3 characters. Where's the 4th one coming from?

12-13

We do the same thing with full_name and notice it gets this correct.

13

Finally we just print out the name and full_name to prove that they actually are "strings" according to printf.

Make sure you can go through and see how these output lines match what was created. We'll be building on this and exploring more about arrays and storage next.

9.2 How To Break It

Breaking this program is fairly easy. Try some of these:

1. Get rid of the '\0' at the end of full_name and re-run it. Run it under Valgrind too. Now, move the definition of full_name to the top of main before areas. Try running it under Valgrind a few times and see if you get some new errors. In some cases, you might still get lucky and not catch any errors.
2. Change it so that instead of areas[0] you try to print areas[10] and see what Valgrind thinks of that.
3. Try other versions of these, doing it to name and full_name too.

9.3 Extra Credit

1. Try assigning to elements in the areas array with areas[0] = 100; and similar.
2. Try assigning to elements of name and full_name.
3. Try setting one element of areas to a character from name.
4. Go search online for the different sizes used for integers on different CPUs.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 10

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 9: Arrays And Strings

In the last exercise you went through an introduction to creating basic arrays and how they map to strings. In this exercise we'll more completely show the similarity between arrays and strings, and get into more about memory layouts.

This exercise shows you that C stores its strings simply as an array of bytes, terminated with the '\0' (nul) byte. You probably clued into this in the last exercise since we did it manually. Here's how we do it in another way to make it even more clear by comparing it to an array of numbers:

Source 27: ex9.c

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int numbers[4] = {0};
6      char name[4] = {'a'};
7
8      // first, print them out raw
9      printf("numbers: %d %d %d %d\n",
10         numbers[0], numbers[1],
11         numbers[2], numbers[3]);
12
13     printf("name each: %c %c %c %c\n",
14         name[0], name[1],
15         name[2], name[3]);
16
17     printf("name: %s\n", name);
18
19     // setup the numbers
20     numbers[0] = 1;
21     numbers[1] = 2;
22     numbers[2] = 3;
23     numbers[3] = 4;
24
25     // setup the name
26     name[0] = 'Z';
27     name[1] = 'e';
28     name[2] = 'd';
29     name[3] = '\0';
30
31     // then print them out initialized
32     printf("numbers: %d %d %d %d\n",
33         numbers[0], numbers[1],
34         numbers[2], numbers[3]);
35
36     printf("name each: %c %c %c %c\n",
37         name[0], name[1],
38         name[2], name[3]);
39
40     // print the name like a string
41     printf("name: %s\n", name);
42
43     // another way to use name
44     char *another = "Zed";
45

```

```

46  printf("another: %s\n", another);
47
48  printf("another each: %c %c %c %c\n",
49         another[0], another[1],
50         another[2], another[3]);
51
52  return 0;
53 }

```

In this code, we setup some arrays the tedious way, by assigning a value to each element. In numbers we are setting up numbers, but in name we're actually building a string manually.

10.1 What You Should See

When you run this code you should see first the arrays printed with their contents initialized to zero, then in its initialized form:

Source 28: ex9 output

```

1  $ make ex9
2  cc -Wall -g  ex9.c -o ex9
3  $ ./ex9
4  numbers: 0 0 0 0
5  name each: a
6  name: a
7  numbers: 1 2 3 4
8  name each: Z e d
9  name: Zed
10 another: Zed
11 another: Z e d
12 $

```

You'll notice some interesting things about this program:

1. I didn't have to give all 4 elements of the arrays to initialize them. This is a short-cut that C has where, if you set just one element, it'll fill the rest in with 0.
2. When each element of numbers is printed they all come out as 0.
3. When each element of name is printed, only the first element 'a' shows up because the '\0' character is special and won't display.
4. Then the first time we print name it only prints "a" because, since the array will be filled with 0 after the first 'a' in the initializer, then the string is correctly terminated by a '\0' character.
5. We then setup the arrays with a tedious manual assignment to each thing and print them out again. Look at how they changed. Now the numbers are set, but see how the name string prints my name correctly?
6. There's also two syntaxes for doing a string: `char name[4] = {'a'}` on line 6 vs. `char *another = name` on line 44. The first one is less common and the second is what you should use for string literals like this.

Notice that I'm using the same syntax and style of code to interact with both an array of integers and an array of characters, but that `printf` thinks that the name is just a string. Again, this is because to the C language there's no difference between a string and an array of characters.

Finally, when you make string literals you should usually use the `char *another = "Literal"` syntax. This works out to be the same thing, but it's more idiomatic and easier to write.

10.2 How To Break It

The source of almost all bugs in C come from forgetting to have enough space, or forgetting to put a '\0' at the end of a string. In fact it's so common and hard to get right that the majority of good C code just doesn't use C style strings. In

later exercises we'll actually learn how to avoid C strings completely.

In this program the key to breaking it is to forget to put the `'\0'` character at the end of the strings. There's a few ways to do this:

1. Get rid of the initializers that setup name.
2. Accidentally set `name[3] = 'A'`; so that there's no terminator.
3. Set the initializer to `{'a','a','a','a'}` so there's too many 'a' characters and no space for the `'\0'` terminator.

Try to come up with some other ways to break this, and as usual run all of these under Valgrind so you can see exactly what is going on and what the errors are called. Sometimes you'll make these mistakes and even Valgrind can't find them, but try moving where you declare the variables to see if you get the error. This is part of the voodoo of C, that sometimes just where the variable is located changes the bug.

10.3 Extra Credit

1. Assign the characters into numbers and then use `printf` to print them a character at a time. What kind of compiler warnings did you get?
2. Do the inverse for name, trying to treat it like an array of int and print it out one int at a time. What does Valgrind think of that?
3. How many other ways can you print this out?
4. If an array of characters is 4 bytes long, and an integer is 4 bytes long, then can you treat the whole name array like it's just an integer? How might you accomplish this crazy hack?
5. Take out a piece of paper and draw out each of these arrays as a row of boxes. Then do the operations you just did on paper to see if you get them right.
6. Convert name to be in the style of another and see if the code keeps working.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)

2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 11

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 10: Arrays Of Strings, Looping

You can make an array of various types, and have the idea down that a "string" and an "array of bytes" are the same thing. The next thing is to take this one step further and do an array that has strings in it. We'll also introduce your first looping construct, the for-loop to help print out this new data structure.

The fun part of this is that there's been an array of strings hiding in your programs for a while now, the `char *argv[]` in the main function arguments. Here's code that will print out any command line arguments you pass it:

Source 29: ex10.c

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int i = 0;
6
7      // go through each string in argv
8      // why am I skipping argv[0]?
9      for(i = 1; i < argc; i++) {
10         printf("arg %d: %s\n", i, argv[i]);
11     }
12
13     // let's make our own array of strings
14     char *states[] = {
15         "California", "Oregon",
16         "Washington", "Texas"
17     };
18     int num_states = 4;
19
20     for(i = 0; i < num_states; i++) {
21         printf("state %d: %s\n", i, states[i]);
22     }
23
24     return 0;
25 }
```

The format of a for-loop is this:

```

for(INITIALIZER; TEST; INCREMENTER) {
    CODE;
}
```

Here's how the for-loop works:

1. The INITIALIZER is code that is run to setup the loop, in this case `i = 0`.
2. Next the TEST boolean expression is checked, and if it's false (0) then CODE is skipped, doing nothing.
3. The CODE runs, does whatever it does.
4. After the CODE runs, the INCREMENTER part is run, usually incrementing something, like in `i++`.
5. And it continues again with Step 2 until the TEST is false (0).

This for-loop is going through the command line arguments using argc and argv like this:

1. The OS passes each command line argument as a string in the argv array. The program's name (./ex10) is at 0, with the rest coming after it.
2. The OS also sets argc to the number of arguments in the argv array so you can process them without going past the end. Remember that if you give one argument, the program's name is the first, so argc is 2.
3. The for-loop sets up with $i = 1$ in the initializer.
4. It then tests that i is less than argc with the test $i < \text{argc}$. Since initially $1 < 2$ it will pass.
5. It then runs the code which just prints out the i and uses i to index into argv.
6. The incrementer is then run using the $i++$ syntax, which is a handy way of writing $i = i + 1$.
7. This then repeats until $i < \text{argc}$ is finally false (0) when the loop exits and the program continues on.

11.1 What You Should See

To play with this program you have to run it two ways. The first way is to pass in some command line arguments so that argc and argv get set. The second is to run it with no arguments so you can see that the first for-loop doesn't run since $i < \text{argc}$ will be false.

Source 30: ex10 output

```

1  $ make ex10
2  cc -Wall -g  ex10.c -o ex10
3  $ ./ex10 i am a bunch of arguments
4  arg 1: i
5  arg 2: am
6  arg 3: a
7  arg 4: bunch
8  arg 5: of
9  arg 6: arguments
10 state 0: California
11 state 1: Oregon
12 state 2: Washington
13 state 3: Texas
14 $
15 $ ./ex10
16 state 0: California
17 state 1: Oregon
18 state 2: Washington
19 state 3: Texas
20 $

```

11.1.1 Understanding Arrays Of Strings

From this you should be able to figure out that in C you make an "array of strings" by combining the `char *str = "blah"` syntax with the `char str[] = {'b','l','a','h'}` syntax to construct a 2-dimensional array. The syntax `char *states[] = {...}` on line 14 is this 2-dimension combination, with each string being one element, and each character in the string being another.

Confusing? The concept of multiple dimensions is something most people never think about so what you should do is build this array of strings on paper:

1. Make a grid with the index of each string on the left.
2. Then put the index of each character on the top.
3. Then, fill in the squares in the middle with what single character goes in that cell.
4. Once you have the grid, trace through the code manually using this grid of paper.

Another way to figure this is out is to build the same structure in a programming language you are more familiar with like Python or Ruby.

11.2 How To Break It

1. Take your favorite other language, and use it to run this program, but with as many command line arguments as possible. See if you can bust it by giving it way too many arguments.
2. Initialize `i` to 0 and see what that does. Do you have to adjust `argc` as well or does it just work? Why does 0-based indexing work here?
3. Set `num_states` wrong so that it's a higher value and see what it does.

11.3 Extra Credit

1. Figure out what kind of code you can put into the parts of a for-loop.
2. Look up how to use the `,` (comma) character to separate multiple statements in the parts of the for-loop.
3. Read what a NULL is and try to use it in one of the elements of the `states` array to see what it'll print.
4. See if you can assign an element from the `states` array to the `argv` array before printing both. Try the inverse.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 12

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 11: While-Loop And Boolean Expressions

You've had your first taste of how C does loops, but the boolean expression `i < argc` might have not been clear to you. Let me explain something about it before we see how a while-loop works.

In C, there's not really a "boolean" type, and instead any integer that's 0 is "false" and otherwise it's "true". In the last exercise the expression `i < argc` actually resulted in 1 or 0, not an explicit True or False like in Python. This is another example of C being closer to how a computer works, because to a computer truth values are just integers.

Now you'll take and implement the same program from the last exercise but use a while-loop instead. This will let you compare the two so you can see how one is related to another.

Source 31: ex11.c

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      // go through each string in argv
6
7      int i = 0;
8      while(i < argc) {
9          printf("arg %d: %s\n", i, argv[i]);
10         i++;
11     }
12
13     // let's make our own array of strings
14     char *states[] = {
15         "California", "Oregon",
16         "Washington", "Texas"
17     };
18
19     int num_states = 4;
20     i = 0; // watch for this
21     while(i < num_states) {
22         printf("state %d: %s\n", i, states[i]);
23         i++;
24     }
25
26     return 0;
27 }
```

You can see from this that a while-loop is simpler:

```

while(TEST) {
    CODE;
}
```

It simply runs the CODE as long as TEST is true (1). This means that to replicate how the for-loop works we need to do our own initializing and incrementing of i.

12.1 What You Should See

The output is basically the same, so I just did it a little different so you can see another way it runs.

Source 32: ex11 output

```

1  $ make ex11
2  cc -Wall -g  ex11.c  -o ex11
3  $ ./ex11
4  arg 0: ./ex11
5  state 0: California
6  state 1: Oregon
7  state 2: Washington
8  state 3: Texas
9  $
10 $ ./ex11 test it
11 arg 0: ./ex11
12 arg 1: test
13 arg 2: it
14 state 0: California
15 state 1: Oregon
16 state 2: Washington
17 state 3: Texas
18 $

```

12.2 How To Break It

In your own code you should favor for-loop constructs over while-loop because a for-loop is harder to break. Here's a few common ways:

1. Forget to initialize the first int i; so have it loop wrong.
2. Forget to initialize the second loop's i so that it retains the value from the end of the first loop. Now your second loop might or might not run.
3. Forget to do a i++ increment at the end of the loop and you get a "forever loop", one of the dreaded problems of the first decade or two of programming.

12.3 Extra Credit

1. Make these loops count backward by using i-- to start at argc and count down to 0. You may have to do some math to make the array indexes work right.
2. Use a while loop to copy the values from argv into states.
3. Make this copy loop never fail such that if there's too many argv elements it won't put them all into states.
4. Research if you've really copied these strings. The answer may surprise and confuse you though.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 13

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 12: If, Else-If, Else

Something common in every language is the if-statement, and C has one. Here's code that uses an if-statement to make sure you enter only 1 or 2 arguments:

Source 33: ex12.c

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int i = 0;
6
7      if(argc == 1) {
8          printf("You only have one argument. You suck.\n");
9      } else if(argc > 1 && argc < 4) {
10         printf("Here's your arguments:\n");
11
12         for(i = 0; i < argc; i++) {
13             printf("%s ", argv[i]);
14         }
15         printf("\n");
16     } else {
17         printf("You have too many arguments. You suck.\n");
18     }
19
20     return 0;
21 }
```

The format for the if-statement is this:

```

if(TEST) {
    CODE;
} else if(TEST) {
    CODE;
} else {
    CODE;
}
```

This is like most other languages except for some specific C differences:

1. As mentioned before, the TEST parts are false if they evaluate to 0, and true otherwise.
2. You have to put parenthesis around the TEST elements, while some other languages let you skip that.
3. You don't need the {} braces to enclose the code, but it is very bad form to not use them. The braces make it clear where one branch of code begins and ends. If you don't include it then obnoxious errors come up.

Other than that, they work like others do. You don't need to have either else if or else parts.

13.1 What You Should See

This one is pretty simple to run and try out:

Source 34: ex12 output

```
1 $ make ex12
2 cc -Wall -g ex12.c -o ex12
3 $ ./ex12
4 You only have one argument. You suck.
5 $ ./ex12 one
6 Here's your arguments:
7 ./ex12 one
8 $ ./ex12 one two
9 Here's your arguments:
10 ./ex12 one two
11 $ ./ex12 one two three
12 You have too many arguments. You suck.
13 $
```

13.2 How To Break It

This one isn't easy to break because it's so simple, but try messing up the tests in the if-statement.

1. Remove the else at the end and it won't catch the edge case.
2. Change the && to a || so you get an "or" instead of "and" test and see how that works.

13.3 Extra Credit

1. You were briefly introduced to &&, which does an "and" comparison, so go research online the different "boolean operators".
2. Write a few more test cases for this program to see what you can come up with.
3. Go back to Exercises 10 and 11, and use if-statements to make the loops exit early. You'll need the break statement to do that. Go read about it.
4. Is the first test really saying the right thing? To you the "first argument" isn't the same first argument a user entered. Fix it.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 14

Exercise 13: Switch Statement

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

In other languages like Ruby you have a switch-statement that can take any expression. Some languages like Python just don't have a switch-statement since an if-statement with boolean expressions is about the same thing. For these languages, switch-statements are more alternatives to if-statements and work the same internally.

The switch-statement is actually entirely different and is really a "jump table". Instead of random boolean expressions, you can only put expressions that result in integers, and these integers are used to calculate jumps from the top of the switch to the part that matches that value. Here's some code that we'll break down to understand this concept of "jump tables":

Source 35: ex13.c

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      if(argc != 2) {
6          printf("ERROR: You need one argument.\n");
7          // this is how you abort a program
8          return 1;
9      }
10
11     int i = 0;
12     for(i = 0; argv[1][i] != '\0'; i++) {
13         char letter = argv[1][i];
14
15         switch(letter) {
16             case 'a':
17             case 'A':
18                 printf("%d: 'A'\n", i);
19                 break;
20
21             case 'e':
22             case 'E':
23                 printf("%d: 'E'\n", i);
24                 break;
25
26             case 'i':
27             case 'I':
28                 printf("%d: 'I'\n", i);
29                 break;
30
31             case 'o':
32             case 'O':
33                 printf("%d: 'O'\n", i);
34                 break;
35
36             case 'u':
37             case 'U':
38                 printf("%d: 'U'\n", i);
39                 break;
40
41             case 'y':
42             case 'Y':

```



```

43     if(i > 2) {
44         // it's only sometimes Y
45         printf("%d: 'Y'\n", i);
46     }
47     break;
48
49     default:
50         printf("%d: %c is not a vowel\n", i, letter);
51 }
52 }
53
54 return 0;
55 }

```

In this program we take a single command line argument and print out all of the vowels in an incredibly tedious way to demonstrate a switch-statement. Here's how the switch-statement works:

1. The compiler marks the place in the program where the switch-statement starts, let's call this location Y.
2. It then evaluates the expression in switch(letter) to come up with a number. In this case the number will be the raw ASCII code of the letter in argv[1].
3. The compiler has also translated each of the case blocks like case 'A': into a location in the program that is that far away. So the code under case 'A' is at Y+'A' in the program.
4. It then does the math to figure out where Y+letter is located in the switch-statement, and if it's too far then it adjusts it to Y+default.
5. Once it knows the location, the program "jumps" to that spot in the code, and then continues running. This is why you have break on some of the case blocks, but not others.
6. If 'a' is entered, then it jumps to case 'a', there's no break so it "falls through" to the one right under it case 'A' which has code and a break.
7. Finally it runs this code, hits the break then exits out of the switch-statement entirely.

This is a deep dive into how the switch-statement works, but in practice you just have to remember a few simple rules:

1. Always include a default: branch so that you catch any missing inputs.
2. Don't allow "fall through" unless you really want it, and it's a good idea to add a comment `//fallthrough` so people know it's on purpose.
3. Always write the case and the break before you write the code that goes in it.
4. Try to just use if-statements instead if you can.

14.1 What You Should See

Here's an example of me playing with this, and also demonstrating various ways to pass the argument in:

Source 36: ex13 output

```

1  $ make ex13
2  cc -Wall -g  ex13.c -o ex13
3  $ ./ex13
4  ERROR: You need one argument.
5  $
6  $ ./ex13 Zed
7  0: Z is not a vowel
8  1: 'E'
9  2: d is not a vowel
10 $
11 $ ./ex13 Zed Shaw
12 ERROR: You need one argument.
13 $
14 $ ./ex13 "Zed Shaw"
15 0: Z is not a vowel
16 1: 'E'

```

```

17 2: d is not a vowel
18 3:  is not a vowel
19 4: S is not a vowel
20 5: h is not a vowel
21 6: 'A'
22 7: w is not a vowel
23 $

```

Remember that there's that if-statement at the top that exits with a return 1; when you don't give enough arguments. Doing a return that's not 0 is how you indicate to the OS that the program had an error. Any value that's greater than 0 can be tested for in scripts and other programs to figure out what happened.

14.2 How To Break It

It is incredibly easy to break a switch-statement. Here's just a few of the ways you can mess one of these up:

1. Forget a break and it'll run two or more blocks of code you don't want it to run.
2. Forget a default and have it silently ignore values you forgot.
3. Accidentally put in variable into the switch that evaluates to something unexpected, like an int that becomes weird values.
4. Use uninitialized values in the switch.

You can also break this program in a few other ways. See if you can bust it yourself.

14.3 Extra Credit

1. Write another program that uses math on the letter to convert it to lowercase, and then remove all the extraneous uppercase letters in the switch.
2. Use the ',' (comma) to initialize letter in the for-loop.
3. Make it handle all of the arguments you pass it with yet another for-loop.
4. Convert this switch-statement to an if-statement. Which do you like better?
5. In the case for 'Y' I have the break outside the if-statement. What's the impact of this and what happens if you move it inside the if-statement. Prove to yourself that you're right.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 15

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 14: Writing And Using Functions

Until now you've just used functions that are part of the `stdio.h` header file. In this exercise you will write some functions and use some other functions.

Source 37: `ex14.c`

```

1  #include <stdio.h>
2  #include <ctype.h>
3
4  // forward declarations
5  int can_print_it(char ch);
6  void print_letters(char arg[]);
7
8  void print_arguments(int argc, char *argv[])
9  {
10     int i = 0;
11
12     for(i = 0; i < argc; i++) {
13         print_letters(argv[i]);
14     }
15 }
16
17 void print_letters(char arg[])
18 {
19     int i = 0;
20
21     for(i = 0; arg[i] != '\0'; i++) {
22         char ch = arg[i];
23
24         if(can_print_it(ch)) {
25             printf("%c" == "%d ", ch, ch);
26         }
27     }
28     printf("\n");
29 }
30
31
32 int can_print_it(char ch)
33 {
34     return isalpha(ch) || isblank(ch);
35 }
36
37
38 int main(int argc, char *argv[])
39 {
40     print_arguments(argc, argv);
41     return 0;
42 }
```

In this example you're creating functions to print out the characters and ASCII codes for any that are "alpha" or "blanks". Here's the breakdown:

`ex14.c:2`

Include a new header file so we can gain access to `isalpha` and `isblank`.

ex14.c:5–6

Tell C that you will be using some functions later in your program, without having to actually define them. This is a "forward declaration" and it solves the chicken-and-egg problem of needing to use a function before you've defined it.

ex14.c:8–15

Define the `print_arguments` which knows how to print the same array of strings that `main` typically gets.

ex14.c:17–30

Define the next function `print_letters` that is called by `print_arguments` and knows how to print each of the characters and their codes.

ex14.c:32–35

Define `can_print_it` which simply returns the truth value (0 or 1) of `isalpha(ch) || isblank(ch)` back to its caller `print_letters`.

ex14.c:38–42

Finally `main` simply calls `print_arguments` to make the whole chain of function calls go.

I shouldn't have to describe what's in each function because it's all things you've ran into before. What you should be able to see though is that I've simply defined functions the same way you've been defining `main`. The only difference is you have to help C out by telling it ahead of time if you're going to use functions it hasn't encountered yet in the file. That's what the "forward declarations" at the top do.

15.1 What You Should See

To play with this program you just feed it different command line arguments, which get passed through your functions. Here's me playing with it to demonstrate:

Source 38: ex14 output

```

1  $ make ex14
2  cc -Wall -g  ex14.c  -o ex14
3
4  $ ./ex14
5  'e' == 101 'x' == 120
6
7  $ ./ex14 hi this is cool
8  'e' == 101 'x' == 120
9  'h' == 104 'i' == 105
10 't' == 116 'h' == 104 'i' == 105 's' == 115
11 'i' == 105 's' == 115
12 'c' == 99 'o' == 111 'o' == 111 'l' == 108
13
14 $ ./ex14 "I go 3 spaces"
15 'e' == 101 'x' == 120
16 'I' == 73 ' ' == 32 'g' == 103 'o' == 111 ' ' == 32 ' ' == 32 's' == 115 'p' == 112 'a' == 97 'c' == 99 'e' == 101 's' == 115
17 $
```

The `isalpha` and `isblank` do all the work of figuring out if the given character is a letter or a blank. When I do the last run it prints everything but the '3' character, since that is a digit.

15.2 How To Break It

There's two different kinds of "breaking" in this program:

1. Confuse the compiler by removing the forward declarations so it complains about `can_print_it` and `print_letters`.
2. When you call `print_arguments` inside `main` try adding 1 to `argc` so that it goes past the end of the `argv` array.

15.3 Extra Credit

1. Rework these functions so that you have fewer functions. For example, do you really need `can_print_it`?
2. Have `print_arguments` figure how long each argument string is using the `strlen` function, and then pass that length to `print_letters`. Then, rewrite `print_letters` so it only processes this fixed length and doesn't rely on the `'\0'` terminator.
3. Use `man` to lookup information on `isalpha` and `isblank`. Use the other similar functions to print out only digits or other characters.
4. Go read about how different people like to format their functions. Never use the "K&R syntax" as it's antiquated and confusing, but understand what it's doing in case you run into someone who likes it.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 16

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 15: Pointers Dreaded Pointers

Pointers are famous mystical creatures in C that I will attempt to demystify by teaching you the vocabulary used to deal with them. They actually aren't that complex, it's just they are frequently abused in weird ways that make them hard to use. If you avoid the stupid ways to use pointers then they're fairly easy.

To demonstrate pointers in a way we can talk about them, I've written a frivolous program that prints a group of people's ages in three different ways: ¹

Source 39: ex15.c

```

1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      // create two arrays we care about
6      int ages[] = {23, 43, 12, 89, 2};
7      char *names[] = {
8          "Alan", "Frank",
9          "Mary", "John", "Lisa"
10     };
11     // safely get the size of ages
12     int count = sizeof(ages) / sizeof(int);
13     int i = 0;
14
15     // first way using indexing
16     for(i = 0; i < count; i++) {
17         printf("%s has %d years alive.\n",
18             names[i], ages[i]);
19     }
20
21     printf("---\n");
22
23     // setup the pointers to the start of the arrays
24     int *cur_age = ages;
25     char **cur_name = names;
26
27     // second way using pointers
28     for(i = 0; i < count; i++) {
29         printf("%s is %d years old.\n",
30             *(cur_name+i), *(cur_age+i));
31     }
32
33     printf("---\n");
34
35     // third way, pointers are just arrays
36     for(i = 0; i < count; i++) {
37         printf("%s is %d years old again.\n",
38             cur_name[i], cur_age[i]);
39     }
40
41     printf("---\n");
42
43     // fourth way with pointers in a stupid complex way
44     for(cur_name = names, cur_age = ages;
45         (cur_age - ages) < count;
```

```

46     cur_name++, cur_age++)
47     {
48         printf("%s lived %d years so far.\n",
49             *cur_name, *cur_age);
50     }
51
52     return 0;
53 }

```

Before explaining how pointers work, let's break this program down line-by-line so you get an idea of what's going on. As you go through this detailed description, try to answer the questions for yourself on a piece of paper, then see if what you guessed was going on matches my description of pointers later.

ex15.c:6-10

Create two arrays, ages storing some int data, and names storing an array of strings.

ex15.c:12-13

Some variables for our for-loops later.

ex15.c:16-19

You know this is just looping through the two arrays and printing how old each person is. This is using i to index into the array.

ex15.c:24

Create a pointer that points at ages. Notice the use of int * to create a "pointer to integer" type of pointer. That's similar to char *, which is a "pointer to char", and a string is an array of chars. Seeing the similarity yet?

ex15.c:25

Create a pointer that points at names. A char * is already a "pointer to char", so that's just a string. You however need 2 levels, since names is 2-dimensional, that means you need char ** for a "pointer to (a pointer to char)" type. Study that too, explain it to yourself.

ex15.c:28-31

Loop through ages and names but instead use the pointers plus an offset of i. Writing *(cur_name+i) is the same as writing name[i], and you read it as "the value of (pointer cur_name plus i)".

ex15.c:35-39

This shows the "magic" of how C treats pointers and arrays as the same thing. You just use the array syntax for accessing the array, but use the pointer's name. C figures it all out for you.

ex15.c:44-50

Another admittedly insane loop that does the same thing as the other two, but instead it uses various pointer arithmetic methods:

ex15.c:44

Initialize our for-loop by setting cur_name and cur_age to the beginning of the names and ages arrays.

ex15.c:45

The test portion of the for-loop then compares the distance of the pointer cur_age from the start of ages. Why does that work?

ex15.c:46

The increment part of the for-loop then increments both cur_name and cur_age so that they point at the next element of the name and age arrays.

ex15.c:48-49

The pointers cur_name and cur_age are now pointing at one element of the arrays they work on, and we can print them out using just *cur_name and *cur_age, which means "the value of wherever cur_name is pointing".

This seemingly simple program has a large amount of information, and the goal is to get you to attempt figuring pointers out for yourself before I explain them. Don't continue until you've written down what you think a pointer does.

16.1 What You Should See

After you run this program try to trace back each line printed out to the line in the code that produced it. If you have to, alter the printf calls to make sure you got the right line number.

Source 40: ex15 output

```

1  $ make ex15
2  cc -Wall -g  ex15.c -o ex15
3  $ ./ex15
4  Alan has 23 years alive.
5  Frank has 43 years alive.
6  Mary has 12 years alive.
7  John has 89 years alive.
8  Lisa has 2 years alive.
9  ---
10 Alan is 23 years old.
11 Frank is 43 years old.
12 Mary is 12 years old.
13 John is 89 years old.
14 Lisa is 2 years old.
15 ---
16 Alan is 23 years old again.
17 Frank is 43 years old again.
18 Mary is 12 years old again.
19 John is 89 years old again.
20 Lisa is 2 years old again.
21 ---
22 Alan lived 23 years so far.
23 Frank lived 43 years so far.
24 Mary lived 12 years so far.
25 John lived 89 years so far.
26 Lisa lived 2 years so far.
27 $

```

16.2 Explaining Pointers

When you type something like `ages[i]` you are "indexing" into the array `ages`, and you're using the number that's held in `i` to do it. If `i` is set to 0 then it's the same as typing `ages[0]`. We've been calling this number `i` an "index" since it's a location inside `ages` that we want. It could also be called an "address", that's a way of saying "I want the integer in `ages` that is at address `i`".

If `i` is an index, then what's `ages`? To C `ages` is a location in the computer's memory where all of these integers start. It is also an address, and the C compiler will replace anywhere you type `ages` with the address of the very first integer in `ages`. Another way to think of `ages` is it's the "address of the first integer in `ages`". But, the trick is `ages` is an address inside the entire computer. It's not like `i` which was just an address inside `ages`. The `ages` array name is actually an address in the computer.

That leads to a certain realization: C thinks your whole computer is one massive array of bytes. Obviously this isn't very useful, but then C layers on top of this massive array of bytes the concept of types and sizes of those types. You already saw how this worked in previous exercises, but now you can start to get an idea that C is somehow doing the following with your arrays:

1. Creating a block of memory inside your computer.
2. "Pointing" the name `ages` at the beginning of that block.
3. "Indexing" into the block by taking the base address of `ages` and getting the element that's `i` away from there.
4. Converting that address at `ages+i` into a valid int of the right size, such that the index works to return what you want: the int at index `i`.

If you can take a base address, like `ages`, and then "add" to it with another address like `i` to produce a new address, then can you just make something that points right at this location all the time? Yes, and that thing is called a "pointer". This is what the pointers `cur_age` and `cur_name` are doing. They are variables pointing at the location where `ages` and `names` live in your computer's memory. The example program is then moving them around or doing math on them to get values out of the memory. In one instance, they just add `i` to `cur_age`, which is the same as what it does with `array[i]`. In the last for-loop though these two pointers are being moved on their own, without `i` to help out. In that loop, the pointers are treated like a combination of array and integer offset rolled into one.

A pointer is simply an address pointing somewhere inside the computer's memory, with a type specifier so you get the right size of data with it. It is kind of like a combined `ages` and `i` rolled into one data type. C knows where pointers are pointing, knows the data type they point at, the size of those types, and how to get the data for you. Just like `i` you can increment them, decrement them, subtract or add to them. But, just like `ages` you can also get values out with them, put new values in, and all the array operations.

The purpose of a pointer is to let you manually index into blocks or memory when an array won't do it right. In almost all other cases you actually want to use an array. But, there are times when you have to work with a raw block of memory and that's where a pointer comes in. A pointer gives you raw, direct access to a block of memory so you can work with it.

The final thing to grasp at this stage is that you can use either syntax for most array or pointer operations. You can take a pointer to something, but use the array syntax for accessing it. You can take an array and do pointer arithmetic with it. In the above example program I demonstrate this and it's a fundamental aspect of C: pointers and arrays are the same thing (mostly).

16.3 Practical Pointer Usage

There are four primary useful things you do with pointers in C code:

1. Ask the OS for a chunk of memory use a pointer to work with it. This includes strings and something you haven't seen yet, structs.
2. Passing large blocks of memory (like strings and arrays) to functions with a pointer so you don't have to pass the whole thing to them.
3. Taking the address of a function so you can use it as a dynamic callback.
4. Complex scanning of chunks of memory such as converting bytes off a network socket into data structures or parsing files.

For nearly everything else you see people use pointers, they should be using arrays. In the early days of C programming people used pointers to speed up their programs because the compilers were really bad at optimizing array usage. These days the syntax to access an array vs. a pointer are translated into the same machine code and optimized the same, so it's not as necessary. Instead, you go with arrays every time you can, and then only use pointers as a performance optimization if you absolutely have to.

16.4 The Pointer Lexicon

I'm now going to give you a little lexicon to use for reading and writing pointers. Whenever you run into a complex pointer statement, just refer to this and break it down bit by bit (or just don't use that code since it's probably not good code):

```
type *ptr
    "a pointer of type named ptr"
```

```
*ptr
    "the value of whatever ptr is pointed at"
*(ptr + i)
    "the value of (whatever ptr is pointed at plus i)"
&thing
    "the address of thing"
type *ptr = &thing
    "a pointer of type named ptr set to the address of thing"
ptr++
    "increment where ptr points"
```

We'll be using this simple lexicon to break down all of the pointers we use from now on in the book.

16.5 How To Break It

You can break this program by simply pointing the pointers at the wrong things:

1. Try to make `cur_age` point at names. You'll need to use a C cast to force it, so go look that up and try to figure it out.
2. In the final for-loop try getting the math wrong in weird ways.
3. Try rewriting the loops so they start at the end of the arrays and go to the beginning. This is harder than it looks.

16.6 Extra Credit

1. Rewrite all the array usage in this program so that it's pointers.
2. Rewrite all the pointer usage so they're arrays.
3. Go back to some of the other programs that use arrays and try to use pointers instead.
4. Process command line arguments using just pointers similar to how you did names in this one.
5. Play with combinations of getting the value of and the address of things.
6. Add another for-loop at the end that prints out the addresses these pointers are using. You'll need the `%p` format for `printf`.
7. Rewrite this program to use a function for each of the ways you're printing out things. Try to pass pointers to these functions so they work on the data. Remember you can declare a function to accept a pointer, but just use it like an array.
8. Change the for-loops to while-loops and see what works better for which kind of pointer usage.

¹Remember, in this book you type in programs you might not understand, and then try figure them out before I explain what's going on.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 17

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 16: Structs And Pointers To Them

In this exercise you'll learn how to make a struct, point a pointer at them, and use them to make sense of internal memory structures. I'll also apply the knowledge of pointers from the last exercise and get you constructing these structures from raw memory using malloc.

As usual, here's the program we'll talk about, so type it in and make it work:

Source 41: ex16.c

```

1  #include <stdio.h>
2  #include <assert.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  struct Person {
7      char *name;
8      int age;
9      int height;
10     int weight;
11 };
12
13 struct Person *Person_create(char *name, int age, int height, int weight)
14 {
15     struct Person *who = malloc(sizeof(struct Person));
16     assert(who != NULL);
17
18     who->name = strdup(name);
19     who->age = age;
20     who->height = height;
21     who->weight = weight;
22
23     return who;
24 }
25
26 void Person_destroy(struct Person *who)
27 {
28     assert(who != NULL);
29
30     free(who->name);
31     free(who);
32 }
33
34 void Person_print(struct Person *who)
35 {
36     printf("Name: %s\n", who->name);
37     printf("\tAge: %d\n", who->age);
38     printf("\tHeight: %d\n", who->height);
39     printf("\tWeight: %d\n", who->weight);
40 }
41
42 int main(int argc, char *argv[])
43 {
44     // make two people structures
45     struct Person *joe = Person_create(
46         "Joe Alex", 32, 64, 140);
47

```

```

48 struct Person *frank = Person_create(
49     "Frank Blank", 20, 72, 180);
50
51 // print them out and where they are in memory
52 printf("Joe is at memory location %p:\n", joe);
53 Person_print(joe);
54
55 printf("Frank is at memory location %p:\n", frank);
56 Person_print(frank);
57
58 // make everyone age 20 years and print them again
59 joe->age += 20;
60 joe->height -= 2;
61 joe->weight += 40;
62 Person_print(joe);
63
64 frank->age += 20;
65 frank->weight += 20;
66 Person_print(frank);
67
68 // destroy them both so we clean up
69 Person_destroy(joe);
70 Person_destroy(frank);
71
72 return 0;
73 }

```

To describe this program, I'm going to use a different approach than before. I'm not going to give you a line-by-line breakdown of the program, but I'm going to make you write it. I'm going to give you a guide through the program based on the parts it contains, and your job is to write out what each line does.

includes

I include some new header files here to gain access to some new functions. What does each give you?

struct Person

This is where I'm creating a structure that has 4 elements to describe a person. The final result is a new compound type that lets me reference these elements all as one, or each piece by name. It's similar to a row of a database table or a class in an OOP language.

function Person_create

I need a way to create these structures so I've made a function to do that. Here's the important things this function is doing:

1. I use malloc for "memory allocate" to ask the OS to give me a piece of raw memory.
2. I pass to malloc the sizeof(struct Person) which calculates the total size of the struct, given all the fields inside it.
3. I use assert to make sure that I have a valid piece of memory back from malloc. There's a special constant called NULL that you use to mean "unset or invalid pointer". This assert is basically checking that malloc didn't return a NULL invalid pointer.
4. I initialize each field of struct Person using the x->y syntax, to say what part of the struct I want to set.
5. I use the strdup function to duplicate the string for the name, just to make sure that this structure actually owns it. The strdup actually is like malloc and it also copies the original string into the memory it creates.

function Person_destroy

If I have a create, then I always need a destroy function, and this is what destroys Person structs. I again use assert to make sure I'm not getting bad input. Then I use the function free to return the memory I got with malloc and strdup. If you don't do this you get a "memory leak".

function Person_print

I then need a way to print out people, which is all this function does. It uses the same x->y syntax to get the field from the struct to print it.

function main

In the main function I use all the previous functions and the struct Person to do the following:

1. Create two people, joe and frank.
2. Print them out, but notice I'm using the %p format so you can see where the program has actually put your struct in memory.
3. Age both of them by 20 years, with changes to their body too.
4. Print each one after aging them.
5. Finally destroy the structures so we can clean up correctly.

Go through this description carefully, and do the following:

1. Look up every function and header file you don't know about. Remember that you can usually do man 2 function or man 3 function and it'll tell you about it. You can also search online for the information.
2. Write a comment above each and every single line saying what the line does in English.
3. Trace through each function call and variable so you know where it comes from in the program.
4. Look up any symbols you don't know as well.

17.1 What You Should See

After you augment the program with your description comments, make sure it really runs and produces this output:

Source 42: ex16 output

```

1  $ make ex16
2  cc -Wall -g  ex16.c  -o ex16
3
4  $ ./ex16
5  Joe is at memory location 0xeba010:
6  Name: Joe Alex
7     Age: 32
8     Height: 64
9     Weight: 140
10 Frank is at memory location 0xeba050:
11 Name: Frank Blank
12 Age: 20
13 Height: 72
14 Weight: 180
15 Name: Joe Alex
16 Age: 52
17 Height: 62
18 Weight: 180
19 Name: Frank Blank
20 Age: 40
21 Height: 72
22 Weight: 200

```

17.2 Explaining Structures

If you've done the work I asked you then structures should be making sense, but let me explain them explicitly just to make sure you've understood it.

A structure in C is a collection of other data types (variables) that are stored in one block of memory but let you access each variable independently by name. They are similar to a record in a database table, or a very simplistic class in an object oriented language. We can break one down this way:

1. In the above code, you make a struct that has the fields you'd expect for a person: name, age, weight, height.
2. Each of those fields has a type, like int.
3. C then packs those together so they can all be contained in one single struct.
4. The struct Person is now a compound data type, which means you can now refer to struct Person in the same kinds of

expressions you would other data types.

5. This lets you pass the whole cohesive grouping to other functions, as you did with `Person_print`.
6. You can then access the individual parts of a struct by their names using `x->y` if you're dealing with a pointer.
7. There's also a way to make a struct that doesn't need a pointer, and you use the `x.y` (period) syntax to work with it. You'll do this in the Extra Credit.

If you didn't have struct you'd need to figure out the size, packing, and location of pieces of memory with contents like this. In fact, in most early assembler code (and even some now) this is what you do. With C you can let C handle the memory structuring of these compound data types and then focus on what you do with them.

17.3 How To Break It

With this program the ways to break it involve how you use the pointers and the malloc system:

1. Try passing `NULL` to `Person_destroy` to see what it does. If it doesn't abort then you must not have the `-g` option in your Makefile's `CFLAGS`.
2. Forget to call `Person_destroy` at the end, then run it under Valgrind to see it report that you forgot to free the memory. Figure out the options you need to pass to Valgrind to get it to print how you leaked this memory.
3. Forget to free `who->name` in `Person_destroy` and compare the output. Again, use the right options to see how Valgrind tells you exactly where you messed up.
4. This time, pass `NULL` to `Person_print` and see what Valgrind thinks of that.
5. You should be figuring out that `NULL` is a quick way to crash your program.

17.4 Extra Credit

In this exercise I want you to attempt something difficult for the extra credit: Convert this program to not use pointers and malloc. This will be hard, so you'll want to research the following:

1. How to create a struct on the stack, which means just like you've been making any other variable.
2. How to initialize it using the `x.y` (period) character instead of the `x->y` syntax.
3. How to pass a structure to other functions without using a pointer.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 18

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 17: Heap And Stack Memory Allocation

In this exercise you're going to make a big leap in difficulty and create an entire small program to manage a database. This database isn't very efficient and doesn't store very much, but it does demonstrate most of what you've learned so far. It also introduces memory allocation more formally and gets you started working with files. We use some file I/O functions, but I won't be explaining them too well so you can try to figure them out first.

As usual, type this whole program in and get it working, then we'll discuss:

Source 43: ex17.c

```

1  #include <stdio.h>
2  #include <assert.h>
3  #include <stdlib.h>
4  #include <errno.h>
5  #include <string.h>
6
7  #define MAX_DATA 512
8  #define MAX_ROWS 100
9
10 struct Address {
11     int id;
12     int set;
13     char name[MAX_DATA];
14     char email[MAX_DATA];
15 };
16
17 struct Database {
18     struct Address rows[MAX_ROWS];
19 };
20
21 struct Connection {
22     FILE *file;
23     struct Database *db;
24 };
25
26 void die(const char *message)
27 {
28     if(errno) {
29         perror(message);
30     } else {
31         printf("ERROR: %s\n", message);
32     }
33
34     exit(1);
35 }
36
37 void Address_print(struct Address *addr)
38 {
39     printf("%d %s %s\n",
40         addr->id, addr->name, addr->email);
41 }
42
43 void Database_load(struct Connection *conn)
44 {
45     int rc = fread(conn->db, sizeof(struct Database), 1, conn->file);

```

```

46  if(rc != 1) die("Failed to load database.");
47  }
48
49  struct Connection* Database_open(const char *filename, char mode)
50  {
51      struct Connection *conn = malloc(sizeof(struct Connection));
52      if(!conn) die("Memory error");
53
54      conn->db = malloc(sizeof(struct Database));
55      if(!conn->db) die("Memory error");
56
57      if(mode == 'c') {
58          conn->file = fopen(filename, "w");
59      } else {
60          conn->file = fopen(filename, "r+");
61
62          if(conn->file) {
63              Database_load(conn);
64          }
65      }
66
67      if(!conn->file) die("Failed to open the file");
68
69      return conn;
70  }
71
72  void Database_close(struct Connection *conn)
73  {
74      if(conn) {
75          if(conn->file) fclose(conn->file);
76          if(conn->db) free(conn->db);
77          free(conn);
78      }
79  }
80
81  void Database_write(struct Connection *conn)
82  {
83      rewind(conn->file);
84
85      int rc = fwrite(conn->db, sizeof(struct Database), 1, conn->file);
86      if(rc != 1) die("Failed to write database.");
87
88      rc = fflush(conn->file);
89      if(rc == -1) die("Cannot flush database.");
90  }
91
92  void Database_create(struct Connection *conn)
93  {
94      int i = 0;
95
96      for(i = 0; i < MAX_ROWS; i++) {
97          // make a prototype to initialize it
98          struct Address addr = {.id = i, .set = 0};
99          // then just assign it
100         conn->db->rows[i] = addr;
101     }
102 }
103
104 void Database_set(struct Connection *conn, int id, const char *name, const char *email)
105 {
106     struct Address *addr = &conn->db->rows[id];
107     if(addr->set) die("Already set, delete it first");
108
109     addr->set = 1;
110     // WARNING: bug, read the "How To Break It" and fix this
111     char *res = strncpy(addr->name, name, MAX_DATA);
112     // demonstrate the strncpy bug
113     if(!res) die("Name copy failed");
114
115     res = strncpy(addr->email, email, MAX_DATA);
116     if(!res) die("Email copy failed");
117 }
118
119 void Database_get(struct Connection *conn, int id)
120 {
121     struct Address *addr = &conn->db->rows[id];
122
123     if(addr->set) {
124         Address_print(addr);
125     } else {
126         die("ID is not set");
127     }

```

```

128 }
129
130 void Database_delete(struct Connection *conn, int id)
131 {
132     struct Address addr = {.id = id, .set = 0};
133     conn->db->rows[id] = addr;
134 }
135
136 void Database_list(struct Connection *conn)
137 {
138     int i = 0;
139     struct Database *db = conn->db;
140
141     for(i = 0; i < MAX_ROWS; i++) {
142         struct Address *cur = &db->rows[i];
143
144         if(cur->set) {
145             Address_print(cur);
146         }
147     }
148 }
149
150 int main(int argc, char *argv[])
151 {
152     if(argc < 3) die("USAGE: ex17 <dbfile> <action> [action params]");
153
154     char *filename = argv[1];
155     char action = argv[2][0];
156     struct Connection *conn = Database_open(filename, action);
157     int id = 0;
158
159     if(argc > 3) id = atoi(argv[3]);
160     if(id >= MAX_ROWS) die("There's not that many records.");
161
162     switch(action) {
163         case 'c':
164             Database_create(conn);
165             Database_write(conn);
166             break;
167
168         case 'g':
169             if(argc != 4) die("Need an id to get");
170
171             Database_get(conn, id);
172             break;
173
174         case 's':
175             if(argc != 6) die("Need id, name, email to set");
176
177             Database_set(conn, id, argv[4], argv[5]);
178             Database_write(conn);
179             break;
180
181         case 'd':
182             if(argc != 4) die("Need id to delete");
183
184             Database_delete(conn, id);
185             Database_write(conn);
186             break;
187
188         case 'l':
189             Database_list(conn);
190             break;
191         default:
192             die("Invalid action, only: c=create, g=get, s=set, d=del, l=list");
193     }
194
195     Database_close(conn);
196
197     return 0;
198 }

```

In this program I am using a set of structures to create a simple database for an address book. In it I'm using some things you've never seen, so you should go through it line-by-line, explain what each line does, and look up any functions you do not recognize. There are few key things I'm doing that you should pay attention to as well:

#define for constants

I use another part of the "C Pre-Processor" to create constant settings of MAX_DATA and MAX_ROWS. I'll cover more of what the CPP does, but this is a way to create a constant that will work reliably. There's other ways but they don't apply in certain situations.

Fixed Sized Structs

The Address struct then uses these constants to create a piece of data that is fixed in size making it less efficient, but easier to store and read. The Database struct is then also fixed size because it is a fixed length array of Address structs. That lets you write the whole thing to disk in one move later on.

die function to abort with an error

In a small program like this you can make a single function that kills the program with an error if there's anything wrong. I call this die, and it's used after any failed function calls or bad inputs to exit with an error using exit.

errno and perror() for error reporting

When you have an error return from a function, it will usually set an "external" variable called errno to say exactly what error happened. These are just numbers, so you can use perror to "print the error message".

FILE functions

I'm using all new functions like fopen, fread, fclose, and rewind to work with files. Each of these functions works on a FILE struct that's just like your structs, but it's defined by the C standard library.

nested struct pointers

There's use of nested structures and getting the address of array elements that you should study. Specifically code like `&conn->db->rows[i]` which reads "get the i element of rows, which is in db, which is in conn, then get the address of (&) it".

copying struct prototypes

best shown in Database_delete, you can see I'm using a temporary local Address, initializing its id and set fields, and then simply copying it into the rows array by assigning it to the element I want. This trick makes sure that all fields but set and id are initialized to 0s and is actually easier to write. Incidentally, you shouldn't be using memcpy to do these kinds of struct copying operations. Modern C allows you to simply assign one struct to another and it'll handle the copying for you.

processing complex arguments

I'm doing some more complex argument parsing, but this isn't really the best way to do it. We'll get into better option parsing later in the book.

converting strings to ints

I use the atoi function to take the string for the id on the command line and convert it to the int id variable. Read up on this function and similar ones.

allocating large data on the "heap"

The whole point of this program is that I'm using malloc to ask the OS for a large amount of memory to work with when I create the Database. I cover this in more detail below.

NULL is 0 so boolean works

In many of the checks I'm testing that a pointer is not NULL by simply doing `if(!ptr) die("fail!")` this is valid because NULL is the same as 0 so it will evaluate to false. You could be explicit and say `if(ptr == NULL) die("fail!")` as well.

18.1 What You Should See

You should spend as much time as you can testing that it works, and running it with Valgrind to confirm you've got all the memory usage right. Here's a session of me testing it normally and then using Valgrind to check the operations:

Source 44: ex17 output

```
1 $ make ex17
2 cc -Wall -g ex17.c -o ex17
3 $ ./ex17 db.dat c
```

```

4 $ ./ex17 db.dat s 1 zed zed@zedshaw.com
5 $ ./ex17 db.dat s 2 frank frank@zedshaw.com
6 $ ./ex17 db.dat s 3 joe joe@zedshaw.com
7 $
8 $ ./ex17 db.dat l
9 1 zed zed@zedshaw.com
10 2 frank frank@zedshaw.com
11 3 joe joe@zedshaw.com
12 $ ./ex17 db.dat d 3
13 $ ./ex17 db.dat l
14 1 zed zed@zedshaw.com
15 2 frank frank@zedshaw.com
16 $ ./ex17 db.dat g 2
17 2 frank frank@zedshaw.com
18 $
19 $ valgrind --leak-check=yes ./ex17 db.dat g 2
20 # cut valgrind output...
21 $

```

The actual output of Valgrind is taken out since you should be able to detect it.

Note 5: OSX Valgrind "Leaks"

Valgrind will report that you're leaking small blocks of memory, but sometimes it's just over-reporting from OSX's internal APIs. If you see it showing leaks that aren't inside your code then just ignore them.

18.2 Heap vs. Stack Allocation

You kids these days have it great. You play with your Ruby or Python and just make objects and variables without any care for where they live. You don't care if it's on the "stack", and the heap? Fuggedaboutit. You don't even know, and you know what, chances are your language of choice doesn't even put the variables on stack at all. It's all heap, and you don't even know if it is.

C is different because it's using the real CPU's actual machinery to do its work, and that involves a chunk of ram called the stack and another called the heap. What's the difference? It all depends on where you get the storage.

The heap is easier to explain as it's just all the remaining memory in your computer, and you access it with the function malloc to get more. Each time you call malloc, the OS uses internal functions to register that piece of memory to you, and then returns a pointer to it. When you're done with it, you use free to return it to the OS so that it can be used by other programs. Failing to do this will cause your program to "leak" memory, but Valgrind will help you track these leaks down.

The stack is a special region of memory that stores temporary variables each function creates as locals to that function. How it works is each argument to a function is "pushed" onto the stack, and then used inside the function. It is really a stack data structure, so the last thing in is the first thing out. This also happens with all local variables like char action and int id in main. The advantage of using a stack for this is simply that, when the function exits, the C compiler "pops" these variables off the stack to clean up. This is simple and prevents memory leaks if the variable is on the stack.

The easiest way to keep this straight is with this mantra: If you didn't get it from malloc or a function that got it from malloc, then it's on the stack.

There's three primary problems with stacks and heaps to watch for:

1. If you get a block of memory from malloc, and have that pointer on the stack, then when the function exits, the pointer will get popped off and lost.
2. If you put too much data on the stack (like large structs and arrays) then you can cause a "stack overflow" and the program will abort. In this case, use the heap with malloc.

3. If you take a pointer to something on the stack, and then pass that or return it from your function, then the function receiving it will "segmentation fault" (segfault) because the actual data will get popped off and disappear. You'll be pointing at dead space.

This is why in the program I've created a `Database_open` that allocates memory or dies, and then a `Database_close` that frees everything. If you create a "create" function, that makes the whole thing or nothing, and then a "destroy" function that cleans up everything safely, then it's easier to keep it all straight.

Finally, when a program exits the OS will clean up all the resources for you, but sometimes not immediately. A common idiom (and one I use in this exercise) is to just abort and let the OS clean up on error.

18.3 How To Break It

This program has a lot of places you can break it, so try some of these but also come up with your own:

1. The classic way is to remove some of the safety checks such that you can pass in arbitrary data. For example, if you remove the check on line 159 that prevents you from passing in any record number.
2. You can also try corrupting the data file. Open it in any editor and change random bytes then close it.
3. You could also find ways to pass bad arguments to the program when it's run, such as getting the file and action backwards will make it create a file named after the action, then do an action based on the first character.
4. There is a bug in this program because of `strncpy` being poorly designed. Go read about `strncpy` then try to find out what happens when the name or address you give is greater than 512 bytes. Fix this by simply forcing the last character to `'\0'` so that it's always set no matter what (which is what `strncpy` should do).
5. In the extra credit I have you augment the program to create arbitrary size databases. Try to see what the biggest database is before you cause the program to die for lack of memory from `malloc`.

18.4 Extra Credit

1. The `die` function needs to be augmented to let you pass the `conn` variable so it can close it and clean up.
2. Change the code to accept parameters for `MAX_DATA` and `MAX_ROWS`, store them in the `Database` struct, and write that to the file, thus creating a database that can be arbitrarily sized.
3. Add more operations you can do on the database, like `find`.
4. Read about how C does its struct packing, and then try to see why your file is the size it is. See if you can calculate a new size after adding more fields.
5. Add some more fields to the `Address` and make them searchable.
6. Write a shell script that will do your testing automatically for you by running commands in the right order. Hint: Use `set -e` at the top of a bash to make it abort the whole script if any command has an error.
7. Try reworking the program to use a single global for the database connection. How does this new version of the program compare to the other one?
8. Go research "stack data structure" and write one in your favorite language, then try to do it in C.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 19

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 18: Pointers To Functions

Functions in C are actually just pointers to a spot in the program where some code exists. Just like you've been creating pointers to structs, strings, and arrays, you can point a pointer at a function too. The main use for this is to pass "callbacks" to other functions, or to simulate classes and objects. In this exercise we'll do some callbacks, and in the next one we'll make a simple object system.

The format of a function pointer goes like this:

```
int (*POINTER_NAME)(int a, int b)
```

A way to remember how to write one is to do this:

1. Write a normal function declaration: `int callme(int a, int b)`
2. Wrap function name with pointer syntax: `int (*callme)(int a, int b)`
3. Change the name to the pointer name: `int (*compare_cb)(int a, int b)`

The key thing to remember is, when you're done with this, the variable name for the pointer is called `compare_cb` and then you use it just like it's a function. This is similar to how pointers to arrays can be used just like the arrays they point to. Pointers to functions can be used like the functions they point to but with a different name.

Source 45: Using A Raw Function Pointer

```
1 int (*tester)(int a, int b) = sorted_order;
2 printf("TEST: %d is same as %d\n", tester(2, 3), sorted_order(2, 3));
```

This will work even if the function pointer returns a pointer to something:

1. Write it: `char *make_coolness(int awesome_levels)`
2. Wrap it: `char *(*make_coolness)(int awesome_levels)`
3. Rename it: `char *(*coolness_cb)(int awesome_levels)`

The next problem to solve with using function pointers is that it's hard to give them as paramters to a function, like when you want to pass the function callback to another function. The solution to this is to use `typedef` which is a C keyword for making new names for other more complex types. The only thing you need to do is put `typedef` before the same function pointer syntax, and then after that you can use the name like it's a type. I demonstrate this in the following exercise code:

Source 46: ex18.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```

3  #include <assert.h>
4  #include <errno.h>
5  #include <string.h>
6
7  /** Our old friend die from ex17. */
8  void die(const char *message)
9  {
10     if(errno) {
11         perror(message);
12     } else {
13         printf("ERROR: %s\n", message);
14     }
15
16     exit(1);
17 }
18
19 // a typedef creates a fake type, in this
20 // case for a function pointer
21 typedef int (*compare_cb)(int a, int b);
22
23 /**
24  * A classic bubble sort function that uses the
25  * compare_cb to do the sorting.
26  */
27 int *bubble_sort(int *numbers, int count, compare_cb cmp)
28 {
29     int temp = 0;
30     int i = 0;
31     int j = 0;
32     int *target = malloc(count * sizeof(int));
33
34     if(!target) die("Memory error.");
35
36     memcpy(target, numbers, count * sizeof(int));
37
38     for(i = 0; i < count; i++) {
39         for(j = 0; j < count - 1; j++) {
40             if(cmp(target[j], target[j+1]) > 0) {
41                 temp = target[j+1];
42                 target[j+1] = target[j];
43                 target[j] = temp;
44             }
45         }
46     }
47
48     return target;
49 }
50
51 int sorted_order(int a, int b)
52 {
53     return a - b;
54 }
55
56 int reverse_order(int a, int b)
57 {
58     return b - a;
59 }
60
61 int strange_order(int a, int b)
62 {
63     if(a == 0 || b == 0) {
64         return 0;
65     } else {
66         return a % b;
67     }
68 }
69
70 /**
71  * Used to test that we are sorting things correctly
72  * by doing the sort and printing it out.
73  */
74 void test_sorting(int *numbers, int count, compare_cb cmp)
75 {
76     int i = 0;
77     int *sorted = bubble_sort(numbers, count, cmp);
78
79     if(!sorted) die("Failed to sort as requested.");
80
81     for(i = 0; i < count; i++) {
82         printf("%d ", sorted[i]);
83     }
84     printf("\n");

```

```

85
86     free(sorted);
87 }
88
89
90 int main(int argc, char *argv[])
91 {
92     if(argc < 2) die("USAGE: ex18 4 3 1 5 6");
93
94     int count = argc - 1;
95     int i = 0;
96     char **inputs = argv + 1;
97
98     int *numbers = malloc(count * sizeof(int));
99     if(!numbers) die("Memory error.");
100
101     for(i = 0; i < count; i++) {
102         numbers[i] = atoi(inputs[i]);
103     }
104
105     test_sorting(numbers, count, sorted_order);
106     test_sorting(numbers, count, reverse_order);
107     test_sorting(numbers, count, strange_order);
108
109     free(numbers);
110
111     return 0;
112 }

```

In this program you're creating a dynamic sorting algorithm that can sort an array of integers using a comparison callback. Here's the breakdown of this program so you can clearly understand it:

ex18.c:1-6

The usual includes needed for all the functions we call.

ex18.c:7-17

This is the die function from the previous exercise which I'll use to do error checking.

ex18.c:21

This is where the typedef is used, and later I use compare_cb like it's a type similar to int or char in bubble_sort and test_sorting.

ex18.c:27-49

A bubble sort implementation, which is a very inefficient way to sort some integers. This function contains:

ex18.c:27

Here's where I use the typedef for compare_cb as the last parameter cmp. This is now a function that will return a comparison between two integers for sorting.

ex18.c:29-34

The usual creation of variables on the stack, followed by a new array of integers on the heap using malloc. Make sure you understand what count * sizeof(int) is doing.

ex18.c:38

The outer-loop of the bubble sort.

ex18.c:39

The inner-loop of the bubble sort

ex18.c:40

Now I call the cmp callback just like it's a normal function, but instead of being the name of something we defined, it's just a pointer to it. This lets the caller pass in anything they want as long as it matches the "signature" of the compare_cb typedef.

ex18.c:41-43

The actual swapping operation a bubble sort needs to do what it does.

ex18.c:48

Finally return the newly created and sorted result array target.

ex18.c:51–68

Three different versions of the `compare_cb` function type, which needs to have the same definition as the typedef we created. The C compiler will complain to you if you get this wrong and say the types don't match.

ex18.c:74–87

This is a tester for the `bubble_sort` function. You can see now how I'm also using `compare_cb` to then pass to `bubble_sort` demonstrating how these can be passed around like any other pointers.

ex18.c:90–103

A simple main function that sets up an array based on integers you pass on the command line, then calls the `test_sorting` function.

ex18.c:105–107

Finally, you get to see how the `compare_cb` function pointer typedef is used. I simply call `test_sorting` but give it the name of `sorted_order`, `reverse_order`, and `strange_order` as the function to use. The C compiler then finds the address of those functions, and makes it a pointer for `test_sorting` to use. If you look at `test_sorting` you'll see it then passes each of these to `bubble_sort` but it actually has no idea what they do, only that they match the `compare_cb` prototype and should work.

ex18.c:109

Last thing we do is free up the array of numbers we made.

19.1 What You Should See

Running this program is simple, but try different combinations of numbers, and try even non-numbers to see what it does.

Source 47: ex18 output

```
1  $ make ex18
2  cc -Wall -g ex18.c -o ex18
3  $ ./ex18 4 1 7 3 2 0 8
4  0 1 2 3 4 7 8
5  8 7 4 3 2 1 0
6  3 4 2 7 1 0 8
7  $
```

19.2 How To Break It

I'm going to have you do something kind of weird to break this. These function pointers are pointers like every other pointer, so they point at blocks of memory. C has this ability to take one pointer and convert it to another so you can process the data in different ways. It's usually not necessary, but to show you how to hack your computer, I want you to add this at the end of `test_sorting`:

Source 48: Function Pointer Evil

```
1  unsigned char *data = (unsigned char *)cmp;
2
3  for(i = 0; i < 25; i++) {
4      printf("%0x:", data[i]);
5  }
6  printf("\n");
```

This loop is sort of like converting your function to a string and then printing out it's contents. This won't break your program unless the CPU and OS you're on has a problem with you doing this. What you'll see is a string of hexadecimal numbers after it prints the sorted array:

```
55:48:89:e5:89:7d:fc:89:75:f8:8b:55:fc:8b:45:f8:29:d0:c9:c3:55:48:89:e5:89:
```

That should be the raw assembler byte code of the function itself, and you should see they start the same, but then have different endings. It's also possible that this loop isn't getting all of the function or is getting too much and stomping on another piece of the program. Without more analysis you wouldn't know.

19.3 Extra Credit

1. Get a hex editor and open up ex18, then find this sequence of hex digits that start a function to see if you can find the function in the raw program.
2. Find other random things in your hex editor and change them. Rerun your program and see what happens. Changing strings you find are the easiest things to change.
3. Pass in the wrong function for the compare_cb and see what the C compiler complains about.
4. Pass in NULL and watch your program seriously bite it. Then run Valgrind and see what that reports.
5. Write another sorting algorithm, then change test_sorting so that it takes both an arbitrary sort function and the sort function's callback comparison. Use it to test both of your algorithms.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 20

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 19: A Simple Object System

I learned C before I learned Object Oriented Programming, so it helped me to build an OOP system in C to understand the basics of what OOP meant. You are probably the kind of person who learned an OOP language before you learned C, so this kind of bridge might help you as well. In this exercise, you will build a simple object system, but also learn more about the C Pre-Processor or CPP.

This exercise will build a simple game where you kill a Minotaur in a small little castle. Nothing fancy, just four rooms and a bad guy. This project will also be a multi-file project, and look more like a real C software project than your previous ones. This is why I'm introducing the CPP here because you need it to start using multiple files in your own software.

20.1 How The CPP Works

The C Pre-Processor is a template processing system. It's a highly targeted one that helps make C easier to work with, but it does this by having a syntax aware templating mechanism. Traditionally people just used the CPP to store constants and make "macros" to simplify repetitive coding. In modern C you'll actually use the CPP as a code generator to create templated pieces of code.

How the CPP works is you give it one file, usually a .c file, and it processes various bits of text starting with the # (octothorpe¹) character. When it encounters one of these it performs a specific replacement on the text of the input file. It's main advantage though is it can include other files, and then augment its list of macros based on that file's contents.

A quick way to see what the CPP does is take the last exercise and run this:

```
cpp ex18.c | less
```

It will be a huge amount of output, but scroll through it and you'll see the contents of the other files you included with #include. Scroll down to the original code and you can see how the cpp is altering the source based on various #define macros in the header files.

The C compiler is so tightly integrated with cpp that it just runs this for you and understands how it works intimately. In modern C, the cpp system is so integral to C's function that you might as well just consider it to be part of the language.

In the remaining sections, we'll be using more of the CPP syntax and explaining it as we go.

20.2 The Prototype Object System

The OOP system we'll create is a simple "prototype" style object system more like JavaScript. Instead of classes, you start with prototypes that have fields set, and then use those as the basis of creating other object instances. This "classless" design is much easier to implement and work with than a traditional class based one.

20.2.1 The Object Header File

I want to put the data types and function declarations into a separate header file named `object.h`. This is standard C practice and it lets you ship binary libraries but still let the programmer compile against it. In this file I have several advanced CPP techniques I'm going to quickly describe and then have you see in action later:

Source 49: `object.h`

```

1  #ifndef _object_h
2  #define _object_h
3
4  typedef enum {
5      NORTH, SOUTH, EAST, WEST
6  } Direction;
7
8  typedef struct {
9      char *description;
10     int (*init)(void *self);
11     void (*describe)(void *self);
12     void (*destroy)(void *self);
13     void (*move)(void *self, Direction direction);
14     int (*attack)(void *self, int damage);
15 } Object;
16
17 int Object_init(void *self);
18 void Object_destroy(void *self);
19 void Object_describe(void *self);
20 void *Object_move(void *self, Direction direction);
21 int Object_attack(void *self, int damage);
22 void *Object_new(size_t size, Object proto, char *description);
23
24 #define NEW(T, N) Object_new(sizeof(T), T##Proto, N)
25 #define _(N) proto.N
26
27 #endif

```

Taking a look at this file, you can see we have a few new pieces of syntax you haven't encountered before:

`#ifndef`

You've seen a `#define` for making simple constants, but the CPP can also do logic and remove sections of code. This `#ifndef` is "if not defined" and checks if there's already a `#define _object_h` and if there is it skips all of this code. I do this so that we can include this file any time we want and not worry about it defining things multiple times.

`#define`

With the above `#ifndef` shielding this file from we then add the `_object_h` define so that any attempts to include it later cause the above to skip.

`#define NEW(T,N)`

This makes a macro, and it works like a template function that spits out the code on the right, whenever you write use the macro on the left. This one is simply making a short version of the normal way we'll call `Object_new` and avoids potential errors with calling it wrong. The way the macro works is the `T` and `N` parameters to `NEW` are "injected" into the line of code on the right. The syntax `T##Proto` says to "concat `Proto` at the end of `T`", so if you had `NEW(Room, "Hello.")` then it'd make `RoomProto` there.

`#define _(N)`

This macro is a bit of "syntactic sugar" for the object system and basically helps you write `obj->proto.blah` as simply `obj->_(blah)`. It's not necessary, but it's a fun little trick that I'll use later.

20.2.2 The Object Source File

The object.h file is declaring functions and data types that are defined (created) in the object.c, so that's next:

Source 50: object.c

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include "object.h"
5  #include <assert.h>
6
7  void Object_destroy(void *self)
8  {
9      Object *obj = self;
10
11     if(obj) {
12         if(obj->description) free(obj->description);
13         free(obj);
14     }
15 }
16
17 void Object_describe(void *self)
18 {
19     Object *obj = self;
20     printf("%s.\n", obj->description);
21 }
22
23 int Object_init(void *self)
24 {
25     // do nothing really
26     return 1;
27 }
28
29 void *Object_move(void *self, Direction direction)
30 {
31     printf("You can't go that direction.\n");
32     return NULL;
33 }
34
35 int Object_attack(void *self, int damage)
36 {
37     printf("You can't attack that.\n");
38     return 0;
39 }
40
41 void *Object_new(size_t size, Object proto, char *description)
42 {
43     // setup the default functions in case they aren't set
44     if(!proto.init) proto.init = Object_init;
45     if(!proto.describe) proto.describe = Object_describe;
46     if(!proto.destroy) proto.destroy = Object_destroy;
47     if(!proto.attack) proto.attack = Object_attack;
48     if(!proto.move) proto.move = Object_move;
49
50     // this seems weird, but we can make a struct of one size,
51     // then point a different pointer at it to "cast" it
52     Object *el = calloc(1, size);
53     *el = proto;
54
55     // copy the description over
56     el->description = strdup(description);
57
58     // initialize it with whatever init we were given
59     if(!el->init(el)) {
60         // looks like it didn't initialize properly
61         el->destroy(el);
62         return NULL;
63     } else {
64         // all done, we made an object of any type
65         return el;
66     }
67 }

```

There's really nothing new in this file, except one tiny little trick. The function Object_new uses an aspect of how structs

work by putting the base prototype at the beginning of the struct. When you look at the ex19.h header later, you'll see how I make the first field in the struct an Object. Since C puts the fields in a struct in order, and since a pointer just points at a chunk of memory, I can "cast" a pointer to anything I want. In this case, even though I'm taking a potentially larger block of memory from calloc, I'm using a Object pointer to work with it.

I explain this a bit better when we write the ex19.h file since it's easier to understand when you see it being used.

That creates your base object system, but you'll need a way to compile it and link it into your ex19.c file to create a complete program. The object.c file on its own doesn't have a main so it isn't enough to make a full program. Here's a Makefile that will do this based on the one you've been using:

Source 51: The Makefile

```
1 CFLAGS=-Wall -g
2
3 all: ex19
4
5 ex19: object.o
6
7 clean:
8     rm -f ex19
```

This Makefile is doing nothing more than saying that ex19 depends on object.o. Remember how make knows how to build different kinds of files by their extensions? Doing this tells make the following:

1. When I say run make the default all should just build ex19.
2. When you build ex19, you need to also build object.o and include it in the build.
3. make can't see anything in the file for object.o, but it does see an object.c file, and it knows how to turn a .c into a .o, so it does that.
4. Once it has object.o built it then runs the correct compile command to build ex19 from ex19.c and object.o.

20.3 The Game Implementation

Once you have those files you just need to implement the actual game using the object system, and first step is putting all the data types and function declarations in a ex19.h file:

Source 52: ex19.h

```
1 #ifndef _ex19_h
2 #define _ex19_h
3
4 #include "object.h"
5
6 struct Monster {
7     Object proto;
8     int hit_points;
9 };
10
11 typedef struct Monster Monster;
12
13 int Monster_attack(void *self, int damage);
14 int Monster_init(void *self);
15
16 struct Room {
17     Object proto;
18
19     Monster *bad_guy;
20
21     struct Room *north;
22     struct Room *south;
23     struct Room *east;
24     struct Room *west;
25 };
```

```

26
27 typedef struct Room Room;
28
29 void *Room_move(void *self, Direction direction);
30 int Room_attack(void *self, int damage);
31 int Room_init(void *self);
32
33
34 struct Map {
35     Object proto;
36     Room *start;
37     Room *location;
38 };
39
40 typedef struct Map Map;
41
42 void *Map_move(void *self, Direction direction);
43 int Map_attack(void *self, int damage);
44 int Map_init(void *self);
45
46 #endif

```

That sets up three new Objects you'll be using: Monster, Room, and Map.

Taking a look at object.c:52 you can see where I use a pointer `Object *el = calloc(1, size)`. Go back and look at the `NEW` macro in object.h and you can see that it is getting the `sizeof` of another struct, say `Room`, and I allocate that much. However, because I've pointed a `Object` pointer at this block of memory, and because I put an `Object proto` field at the from of `Room`, I'm able to treat a `Room` like it's an `Object`.

The way to break this down is like so:

1. I call `NEW(Room, "Hello.")` which the CPP expands as a macro into `Object_new(sizeof(Room), RoomProto, "Hello.")`.
2. This runs, and inside `Object_new` I allocate a piece of memory that's `Room` in size, but point a `Object *el` pointer at it.
3. Since C puts the `Room.proto` field first, that means the `el` pointer is really only pointing at enough of the block of memory to see a full `Object` struct. It has no idea that it's even called `proto`.
4. It then uses this `Object *el` pointer to set the contents of the piece of memory correctly with `*el = proto;`. Remember that you can copy structs, and that `*el` means "the value of whatever `el` points at", so this means "assign the `proto` struct to whatever `el` points at".
5. Now that this mystery struct is filled in with the right data from `proto`, the function can then call `init` or `destroy` on the `Object`, but the cool part is whoever called this function can change these out for whatever ones they want.

And with that, we have a way to get this one function to construct new types, and give them new functions to change their behavior. This may seem like "hackery" but it's stock C and totally valid. In fact there's quite a few standard system functions that work this same way, and we'll be using some of them for converting addresses in network code.

With the function definitions and data structures written out I can now actually implement the game with four rooms and a minotaur to beat up:

Source 53: ex19.c

```

1  #include <stdio.h>
2  #include <errno.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <time.h>
6  #include "ex19.h"
7
8
9  int Monster_attack(void *self, int damage)
10 {
11     Monster *monster = self;
12
13     printf("You attack %s!\n", monster->_(description));
14

```

```

15  monster->hit_points -= damage;
16
17  if(monster->hit_points > 0) {
18      printf("It is still alive.\n");
19      return 0;
20  } else {
21      printf("It is dead!\n");
22      return 1;
23  }
24 }
25
26 int Monster_init(void *self)
27 {
28     Monster *monster = self;
29     monster->hit_points = 10;
30     return 1;
31 }
32
33 Object MonsterProto = {
34     .init = Monster_init,
35     .attack = Monster_attack
36 };
37
38
39 void *Room_move(void *self, Direction direction)
40 {
41     Room *room = self;
42     Room *next = NULL;
43
44     if(direction == NORTH && room->north) {
45         printf("You go north, into: \n");
46         next = room->north;
47     } else if(direction == SOUTH && room->south) {
48         printf("You go south, into: \n");
49         next = room->south;
50     } else if(direction == EAST && room->east) {
51         printf("You go east, into: \n");
52         next = room->east;
53     } else if(direction == WEST && room->west) {
54         printf("You go west, into: \n");
55         next = room->west;
56     } else {
57         printf("You can't go that direction.");
58         next = NULL;
59     }
60
61     if(next) {
62         next->_(describe)(next);
63     }
64
65     return next;
66 }
67
68
69 int Room_attack(void *self, int damage)
70 {
71     Room *room = self;
72     Monster *monster = room->bad_guy;
73
74     if(monster) {
75         monster->_(attack)(monster, damage);
76         return 1;
77     } else {
78         printf("You flail in the air at nothing. Idiot.\n");
79         return 0;
80     }
81 }
82
83
84 Object RoomProto = {
85     .move = Room_move,
86     .attack = Room_attack
87 };
88
89
90 void *Map_move(void *self, Direction direction)
91 {
92     Map *map = self;
93     Room *location = map->location;
94     Room *next = NULL;
95
96     next = location->_(move)(location, direction);

```

```

97
98     if(next) {
99         map->location = next;
100     }
101
102     return next;
103 }
104
105 int Map_attack(void *self, int damage)
106 {
107     Map* map = self;
108     Room *location = map->location;
109
110     return location->_(attack)(location, damage);
111 }
112
113
114 int Map_init(void *self)
115 {
116     Map *map = self;
117
118     // make some rooms for a small map
119     Room *hall = NEW(Room, "The great Hall");
120     Room *throne = NEW(Room, "The throne room");
121     Room *arena = NEW(Room, "The arena, with the minotaur");
122     Room *kitchen = NEW(Room, "Kitchen, you have the knife now");
123
124     // put the bad guy in the arena
125     arena->bad_guy = NEW(Monster, "The evil minotaur");
126
127     // setup the map rooms
128     hall->north = throne;
129
130     throne->west = arena;
131     throne->east = kitchen;
132     throne->south = hall;
133
134     arena->east = throne;
135     kitchen->west = throne;
136
137     // start the map and the character off in the hall
138     map->start = hall;
139     map->location = hall;
140
141     return 1;
142 }
143
144 Object MapProto = {
145     .init = Map_init,
146     .move = Map_move,
147     .attack = Map_attack
148 };
149
150 int process_input(Map *game)
151 {
152     printf("\n> ");
153
154     char ch = getchar();
155     getchar(); // eat ENTER
156
157     int damage = rand() % 4;
158
159     switch(ch) {
160         case -1:
161             printf("Giving up? You suck.\n");
162             return 0;
163             break;
164
165         case 'n':
166             game->_(move)(game, NORTH);
167             break;
168
169         case 's':
170             game->_(move)(game, SOUTH);
171             break;
172
173         case 'e':
174             game->_(move)(game, EAST);
175             break;
176
177         case 'w':
178             game->_(move)(game, WEST);

```

```

179     break;
180
181     case 'a':
182
183         game->_(attack)(game, damage);
184         break;
185     case 'l':
186         printf("You can go:\n");
187         if(game->location->north) printf("NORTH\n");
188         if(game->location->south) printf("SOUTH\n");
189         if(game->location->east) printf("EAST\n");
190         if(game->location->west) printf("WEST\n");
191         break;
192
193     default:
194         printf("What?: %d\n", ch);
195 }
196
197 return 1;
198 }
199
200 int main(int argc, char *argv[])
201 {
202     // simple way to setup the randomness
203     srand(time(NULL));
204
205     // make our map to work with
206     Map *game = NEW(Map, "The Hall of the Minotaur.");
207
208     printf("You enter the ");
209     game->location->_(describe)(game->location);
210
211     while(process_input(game)) {
212     }
213
214     return 0;
215 }

```

Honestly there isn't much in this that you haven't seen, and only you might need to understand how I'm using the macros I made from the headers files. Here's the important key things to study and understand:

1. Implementing a prototype involves creating its version of the functions, and then creating a single struct ending in "Proto". Look at MonsterProto, RoomProto and MapProto.
2. Because of how Object_new is implemented, if you don't set a function in your prototype, then it will get the default implementation created in object.c.
3. In Map_init I create the little world, but more importantly I use the NEW macro from object.h to build all of the objects. To get this concept in your head, try replacing the NEW usage with direct Object_new calls to see how it's being translated.
4. Working with these objects involves calling functions on them, and the _(N) macro does this for me. If you look at the code monster->_(attack)(monster, damage) you see that I'm using the macro, which gets replaced with monster->proto.attack(monster, damage). Study this transformation again by rewriting these calls back to their original. Also, if you get stuck then run cpp manually to see what it's going to do.
5. I'm using two new functions srand and rand, which setup a simple random number generator good enough for the game. I also use time to initialize the random number generator. Research those.
6. I use a new function getchar that gets a single character from the stdin. Research it.

20.4 What You Should See

Here's me playing my own game:

Source 54: ex19 output

```

1  $ make ex19
2  cc -Wall -g -c -o object.o object.c

```

```

3 cc -Wall -g ex19.c object.o -o ex19
4 $ ./ex19
5 You enter the The great Hall.
6
7 > l
8 You can go:
9 NORTH
10
11 > n
12 You go north, into:
13 The throne room.
14
15 > l
16 You can go:
17 SOUTH
18 EAST
19 WEST
20
21 > e
22 You go east, into:
23 Kitchen, you have the knife now.
24
25 > w
26 You go west, into:
27 The throne room.
28
29 > s
30 You go south, into:
31 The great Hall.
32
33 > n
34 You go north, into:
35 The throne room.
36
37 > w
38 You go west, into:
39 The arena, with the minotaur.
40
41 > a
42 You attack The evil minotaur!
43 It is still alive.
44
45 > a
46 You attack The evil minotaur!
47 It is dead!
48
49 > ^D
50 Giving up? You suck.
51 $

```

20.5 Auditing The Game

As an exercise for you I have left out all of the assert checks I normally put into a piece of software. You've seen me use assert to make sure a program is running correctly, but now I want you to go back and do the following:

1. Look at each function you've defined, one file at a time.
2. At the top of each function, add asserts that make sure the input parameters are correct. For example, in Object_new you want a assert(description != NULL).
3. Go through each line of the function, and find any functions being called. Read the documentation (man page) for that function, and confirm what it returns for an error. Add another assert to check that the error didn't happen. For example, in Object_new you need one after the call to calloc that does assert(el != NULL).
4. If a function is expected to return a value, either make sure it returns an error value (like NULL), or have an assert to make sure that the returned variable isn't invalid. For example, in Object_new, you need to have assert(el != NULL) again before the last return since that part can never be NULL.
5. For every if-statement you write, make sure there's an else clause unless that if is an error check that causes an exit.
6. For every switch-statement you write, make sure that there's a default case that handles anything you didn't anticipate.

Take your time going through every line of the function and find any errors you make. Remember that the point of this exercise is to stop being a "coder" and switch your brain into being a "hacker". Try to see how you could break it, then write code to prevent it or abort early if you can.

20.6 Extra Credit

1. Update the Makefile so that when you do make clean it will also remove the object.o file.
2. Write a test script that works the game in different ways and augment the Makefile so you can run make test and it'll thrash the game with your script.
3. Add more rooms and monsters to the game.
4. Put the game mechanics into a third file, compile it to .o, and then use that to write another little game. If you're doing it right you should only have a new Map and a main function in the new game.

¹A.K.A. pound, hash, mesh, number symbol, pick whatever makes you happy

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 21

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 20: Zed's Awesome Debug Macros

There is a constant problem in C that you have been dancing around but which I am going to solve in this exercise using a set of macros I developed. You can thank me later when you realize how insanely awesome these macros are. Right now you won't realize how awesome they are, so you'll just have to use them and then you can walk up to me one day and say, "Zed, those Debug Macros were the bomb. I owe you my first born child because you saved me a decade of heartache and prevented me from killing myself more than once. Thank you good sir, here's a million dollars and the original Snakehead Telecaster prototype signed by Leo Fender."

Yes, they are that awesome.

21.1 The C Error Handling Problem

In almost every programming language handling errors is a difficult activity. There's entire programming languages that try as hard as they can to avoid even the concept of an error. Other languages invent complex control structures like exceptions to pass error conditions around. The problem exists mostly because programmers assume errors don't happen and this optimism infects the type of languages they use and create.

C tackles the problem by returning error codes and setting a global `errno` value that you check. This makes for complex code that simply exists to check if something you did had an error. As you write more and more C code you'll write code with the pattern:

1. Call a function.
2. If the return value is an error (must look that up each time too).
3. Then cleanup all the resource created so far.
4. and print out an error message that hopefully helps.

This means for every function call (and yes, every function) you are potentially writing 3–4 more lines just to make sure it worked. That doesn't include the problem of cleaning up all of the junk you've built to that point. If you have 10 different structures, 3 files, and a database connection, when you get an error then you would have 14 more lines.

In the past this wasn't a problem because C programs did what you've been doing when there's an error: die. No point in bothering with cleanup when the OS will do it for you. Today though many C programs need to run for weeks, months, or years and handle errors from many different sources gracefully. You can't just have your webserver die at the slightest touch, and you definitely can't have a library you've written nuke a the program its used in. That's just rude.

Other languages solve this problem with exceptions, but those have problems in C (and in other languages too). In C you only have one return value, but exceptions are an entire stack based return system with arbitrary values. Trying to

marshal exceptions up the stack in C is difficult, and no other libraries will understand it.

21.2 The Debug Macros

The solution I've been using for years is a small set of "debug macros" that implement a basic debugging and error handling system for C. This system is easy to understand, works with every library, and makes C code more solid and clearer.

It does this by adopting the convention that whenever there's an error, your function will jump to an "error:" part of the function that knows how to cleanup everything and return an error code. You use a macro called `check` to check return codes, print an error message, and then jump to the cleanup section. You combine that with a set of logging functions for printing out useful debug messages.

I'll now show you the entire contents of the most awesome set of brilliance you've ever seen:

Source 55: `dbg.h`

```

1  #ifndef __dbg_h__
2  #define __dbg_h__
3
4  #include <stdio.h>
5  #include <errno.h>
6  #include <string.h>
7
8  #ifdef NDEBUG
9  #define debug(M, ...)
10 #else
11 #define debug(M, ...) fprintf(stderr, "DEBUG %s:%d: " M "\n", __FILE__, __LINE__, ##__VA_ARGS__)
12 #endif
13
14 #define clean_errno() (errno == 0 ? "None" : strerror(errno))
15
16 #define log_err(M, ...) fprintf(stderr, "[ERROR] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__, clean_errno(), ##__VA_ARGS__)
17
18 #define log_warn(M, ...) fprintf(stderr, "[WARN] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__, clean_errno(), ##__VA_ARGS__)
19
20 #define log_info(M, ...) fprintf(stderr, "[INFO] (%s:%d) " M "\n", __FILE__, __LINE__, ##__VA_ARGS__)
21
22 #define check(A, M, ...) if(!(A)) { log_err(M, ##__VA_ARGS__); errno=0; goto error; }
23
24 #define sentinel(M, ...) { log_err(M, ##__VA_ARGS__); errno=0; goto error; }
25
26 #define check_mem(A) check((A), "Out of memory.")
27
28 #define check_debug(A, M, ...) if(!(A)) { debug(M, ##__VA_ARGS__); errno=0; goto error; }
29
30 #endif

```

Yes, that's it, and here's what every line does:

`dbg.h:1-2`

The usual defense against accidentally including the file twice, which you saw in the last exercise.

`dbg.h:4-6`

Includes for the functions that these macros need.

`dbg.h:8`

The start of a `#ifdef` which lets you recompile your program so that all the debug log messages are removed.

`dbg.h:9`

If you compile with `NDEBUG` defined, then "no debug" messages will remain. You can see in this case the `#define debug()` is just replaced with nothing (the right side is empty).

`dbg.h:10`

The matching `#else` for the above `#ifdef`.

dbg.h:11

The alternative `#define` `debug` that translates any use of `debug("format", arg1, arg2)` into an `fprintf` call to `stderr`. Many C programmers don't know, but you can create macros that actually work like `printf` and take variable arguments. Some C compilers (actually `cpp`) don't support this, but the ones that matter do. The magic here is the use of `##_VA_ARGS__` which says "put whatever they had for extra arguments (...) here". Also notice the use of `__FILE__` and `__LINE__` to get the current file:line for the debug message. Very helpful.

dbg.h:12

The end of the `#ifdef`.

dbg.h:14

The `clean_errno` macro that's used in the others to get a safe readable version of `errno`. That strange syntax in the middle is a "ternary operator" and you'll learn what it does later.

dbg.h:16–20

The `log_err`, `log_warn`, and `log_info`, macros for logging messages meant for the end user. Works like `debug` but can't be compiled out.

dbg.h:22

The best macro ever, `check` will make sure the condition `A` is true, and if not logs the error `M` (with variable arguments for `log_err`), then jumps to the function's error: for cleanup.

dbg.h:24

The 2nd best macro ever, `sentinel` is placed in any part of a function that shouldn't run, and if it does prints an error message then jumps to the error: label. You put this in `if`-statements and `switch`-statements to catch conditions that shouldn't happen, like the default:.

dbg.h:26

A short-hand macro `check_mem` that makes sure a pointer is valid, and if it isn't reports it as an error with "Out of memory."

dbg.h:28

An alternative macro `check_debug` that still checks and handles an error, but if the error is common then you don't want to bother reporting it. In this one it will use `debug` instead of `log_err` to report the message, so when you define `NDEBUG` the check still happens, the error jump goes off, but the message isn't printed.

21.3 Using dbg.h

Here's an example of using all of `dbg.h` in a small program. This doesn't actually do anything but demonstrate how to use each macro, but we'll be using these macros in all of the programs we write from now on, so be sure to understand how to use them.

Source 56: ex20.c

```

1  #include "dbg.h"
2  #include <stdlib.h>
3  #include <stdio.h>
4
5
6  void test_debug()
7  {
8      // notice you don't need the \n
9      debug("I have Brown Hair.");
10
11     // passing in arguments like printf
12     debug("I am %d years old.", 37);
13 }
14
15 void test_log_err()
16 {
17     log_err("I believe everything is broken.");
18     log_err("There are %d problems in %s.", 0, "space");
19

```

```

    }
20
21 void test_log_warn()
22 {
23     log_warn("You can safely ignore this.");
24     log_warn("Maybe consider looking at: %s.", "/etc/passwd");
25 }
26
27 void test_log_info()
28 {
29     log_info("Well I did something mundane.");
30     log_info("It happened %f times today.", 1.3f);
31 }
32
33 int test_check(char *file_name)
34 {
35     FILE *input = NULL;
36     char *block = NULL;
37
38     block = malloc(100);
39     check_mem(block); // should work
40
41     input = fopen(file_name, "r");
42     check(input, "Failed to open %s.", file_name);
43
44     free(block);
45     fclose(input);
46     return 0;
47
48 error:
49     if(block) free(block);
50     if(input) fclose(input);
51     return -1;
52 }
53
54 int test_sentinel(int code)
55 {
56     char *temp = malloc(100);
57     check_mem(temp);
58
59     switch(code) {
60         case 1:
61             log_info("It worked.");
62             break;
63         default:
64             sentinel("I shouldn't run.");
65     }
66
67     free(temp);
68     return 0;
69
70 error:
71     if(temp) free(temp);
72     return -1;
73 }
74
75 int test_check_mem()
76 {
77     char *test = NULL;
78     check_mem(test);
79
80     free(test);
81     return 1;
82
83 error:
84     return -1;
85 }
86
87 int test_check_debug()
88 {
89     int i = 0;
90     check_debug(i != 0, "Oops, I was 0.");
91
92     return 0;
93 error:
94     return -1;
95 }
96
97 int main(int argc, char *argv[])
98 {
99     check(argc == 2, "Need an argument.");
100

```

```

101 test_debug();
102 test_log_err();
103 test_log_warn();
104 test_log_info();
105
106 check(test_check("ex20.c") == 0, "failed with ex20.c");
107 check(test_check(argv[1]) == -1, "failed with argv");
108 check(test_sentinel(1) == 0, "test_sentinel failed.");
109 check(test_sentinel(100) == -1, "test_sentinel failed.");
110 check(test_check_mem() == -1, "test_check_mem failed.");
111 check(test_check_debug() == -1, "test_check_debug failed.");
112
113 return 0;
114
115 error:
116 return 1;
117 }

```

Pay attention to how check is used, and how when it is false it will jump to the error: label to do a cleanup. The way to read those lines is, "check that A is true and if not say M and jump out."

21.4 What You Should See

When you run this, give it some bogus first parameter and you should see this:

Source 57: ex20 output

```

1 $ make ex20
2 cc -Wall -g -DNDEBUG ex20.c -o ex20
3 $ ./ex20 test
4 [ERROR] (ex20.c:16: errno: None) I believe everything is broken.
5 [ERROR] (ex20.c:17: errno: None) There are 0 problems in space.
6 [WARN] (ex20.c:22: errno: None) You can safely ignore this.
7 [WARN] (ex20.c:23: errno: None) Maybe consider looking at: /etc/passwd.
8 [INFO] (ex20.c:28) Well I did something mundane.
9 [INFO] (ex20.c:29) It happened 1.300000 times today.
10 [ERROR] (ex20.c:38: errno: No such file or directory) Failed to open test.
11 [INFO] (ex20.c:57) It worked.
12 [ERROR] (ex20.c:60: errno: None) I shouldn't run.
13 [ERROR] (ex20.c:74: errno: None) Out of memory.

```

See how it reports the exact line number where the check failed? That's going to save you hours of debugging later. See also how it prints the error message for you when errno is set? Again, that will save you hours of debugging.

21.5 How The CPP Expands Macros

It's now time for you to get a small introduction to the CPP so that you know how these macros actually work. To do this, I'm going to break down the most complex macro from dbg.h and have you run cpp so you can see what it's actually doing.

Imagine I have a function called dosomething() that return the typical 0 for success and -1 for an error. Every time I call dosomething I have to check for this error code, so I'd write code like this:

```

1 int rc = dosomething();
2 if(rc != 0) {
3     fprintf(stderr, "There was an error: %s\n", strerror());
4     goto error;
5 }

```

What I want to use the CPP for is to encapsulate this if-statement I have to use all the time into a more readable and memorable line of code. I want what you've been doing in dbg.h with the check macro:

```

1 int rc = dosomething();
2 check(rc == 0, "There was an error.");

```

This is much clearer and explains exactly what's going on: check that the function worked, and if not report an error. To do this, we need some special CPP "tricks" that make the CPP useful as a code generation tool. Take a look at the check and log_err macros again:

```
1#define log_err(M, ...) fprintf(stderr, "[ERROR] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__, clean_errno(), ##__VA_ARGS__)
2#define check(A, M, ...) if(!(A)) { log_err(M, ##__VA_ARGS__); errno=0; goto error; }
```

The first macro, log_err is simpler and simply replace itself with a call to fprintf to stderr. The only tricky part of this macro is the use of ... in the definition log_err(M, ...). What this does is let you pass variable arguments to the macro, so you can pass in the arguments that should go to fprintf. How do they get injected into the fprintf call? Look at the end to the ##__VA_ARGS__ and that's telling the CPP to take the args entered where the ... is, and inject them at that part of the fprintf call. You can then do things like this:

```
log_err("Age: %d, name: %s", age, name);
```

The arguments age, name are the ... part of the definition, and those get injected into the fprintf output to become:

```
1fprintf(stderr, "[ERROR] (%s:%d: errno: %s) Age %d: name %d\n",
2  __FILE__, __LINE__, clean_errno(), age, name);
```

See the age, name at the end? That's how ... and ##__VA_ARGS__ work together, and it will work in macros that call other variable argument macros. Look at the check macro now and see it calls log_err, but check is also using the ... and ##__VA_ARGS__ to do the call. That's how you can pass full printf style format strings to check, which go to log_err, and then make both work like printf.

Next thing to study is how check crafts the if-statement for the error checking. If we strip out the log_err usage we see this:

```
if(!(A)) { errno=0; goto error; }
```

Which means, if A is false, then clear errno and goto the error label. That has check macro being replaced with the if-statement so if we manually expanded out the macro check(rc == 0, "There was an error.") we'd get:

```
1if(!(rc == 0) {
2  log_err("There was an error.");
3  errno=0;
4  goto error;
5}
```

What you should be getting from this trip through these two macros is that the CPP replaces macros with the expanded version of their definition, but that it will do this recursively, expanding all the macros in macros. The CPP then is just a recursive templating system, as I mentioned before. Its power comes from its ability to generate whole blocks of parameterized code thus becoming a handy code generation tool.

That leaves one question: Why not just use a function like die? The reason is you want file:line numbers and the goto operation for an error handling exit. If you did this inside a function, you wouldn't get a line number for where the error actually happened, and the goto would be much more complicated.

Another reason is you still have to write the raw if-statement, which looks like all the other if-statements in your code, so it's not as clear that this one is an error check. By wrapping the if-statement in a macro called check you make it clear that this is just error checking, and not part of the main flow.

Finally, CPP has the ability to conditionally compile portions of code, so you can have code that's only present when you build a developer or debug version of the program. You can see this already in the dbg.h file where the debug macro has a body only if it's asked for by the compiler. Without this ability, you'd need a wasted if-statement that checks for "debug mode", and then still wastes CPU doing that check for no value.

21.6 Extra Credit

1. Put `#define NDEBUG` at the top of the file and check that all the debug messages go away.
2. Undo that line, and add `-DNDEBUG` to `CFLAGS` at the top of the Makefile then recompile to see the same thing.
3. Modify the logging so that it include the function name as well as the file:line.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 22

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 21: Advanced Data Types And Flow Control

This exercise will be a complete compendium of the available C data types and flow control structures you can use. It will work as a reference to complete your knowledge, and won't have any code for you to enter. I'll have you memorize some of the information by creating flash cards so you can get the important concepts solid in your mind.

For this exercise to be useful, you should spend at least a week hammering the content and filling out all the element I have missing here. You'll be writing out what each one means, and then writing a program to confirm what you've researched.

22.1 Available Data Types

`int`

Stores a regular integer, defaulting to 32 bits in size.

`double`

Holds a large floating point number.

`float`

Holds a smaller floating point number.

`char`

Holds a single 1 byte character.

`void`

Indicates "no type" and used to say a function returns nothing, or a pointer has no type as in `void *thing`.

`enum`

Enumerated types, work as integers, convert to integers, but give you symbolic names for sets. Some compilers will warn you when you don't cover all elements of an enum in switch-statements.

22.1.1 Type Modifiers

`unsigned`

Changes the type so that it does not have negative numbers, giving you a larger upper bound but nothing lower than 0.

`signed`

Gives you negative and positive numbers, but halves your upper bound in exchange for the same lower bound negative.

`long`

Uses a larger storage for the type so that it can hold bigger numbers, usually doubling the current size.

short

Uses smaller storage for the type so it stores less, but takes half the space.

22.1.2 Type Qualifiers

const

Indicates the variable won't change after being initialized.

volatile

Indicates that all bets are off, and the compiler should leave this alone and try not to do any fancy optimizations to it. You usually only need this if you're doing really weird stuff to your variables.

register

Forces the compiler to keep this variable in a register, and the compiler can just ignore you. These days compilers are better at figuring out where to put variables, so only use this if you actually can measure it improving the speed.

22.1.3 Type Conversion

C uses a sort of "stepped type promotion" mechanism, where it looks at two operands on either side of an expression, and promotes the smaller side to match the larger side before doing the operation. If one side of an expression is on this list, then the other side is converted to that type before the operation is done, and this goes in this order:

1. long double
2. double
3. float
4. int (but only char and short int);
5. long

If you find yourself trying to figure out how your conversions are working in an expression, then don't leave it to the compiler. Use explicit casting operations to make it exactly what you want. For example, if you have:

```
long + char - int * double
```

Rather than trying to figure out if it will be converted to double correctly, just use casts:

```
(double)long - (double)char - (double)int * double
```

Putting the type you want in parenthesis before the variable name is how you force it into the type you really need. The important thing though is always promote up, not down. Don't cast long into char unless you know what you're doing.

22.1.4 Type Sizes

The `stdint.h` defines both a set of typedefs for exact sized integer types, as well as a set of macros for the sizes of all the types. This is easier to work with than the older `limits.h` since it is consistent. The types defined are:

`int8_t`

8 bit signed integer.

`uint8_t`

8 bit unsigned integer.

`int16_t`

16 bit signed integer.

uint16_t

16 bit unsigned integer.

int32_t

32 bit signed integer.

uint32_t

32 bit unsigned integer.

int64_t

64 bit signed integer.

uint64_t

64 bit unsigned integer.

The pattern here is of the form (u)int(BITS)_t where a u is put in front to indicate "unsigned", then BITS is a number for the number of bits. This pattern is then repeated for macros that return the maximum values of these types:

INTN_MAX

Maximum positive number of the signed integer of bits N.

INTN_MIN

Minimum negative number of signed integer of bits N.

UINTN_MAX

Maximum positive number of unsigned integer of bits N. Since it's unsigned the minimum is 0 and can't have a negative value.

There are also macros in stdint.h for sizes of the size_t type, integers large enough to hold pointers, and other handy size defining macros. Compilers have to at least have these, and then they can allow other larger types.

Here is a full list should be in stdint.h:

int_leastN_t

holds at least N bits.

uint_leastN_t

holds at least N bits unsigned.

INT_LEASTN_MAX

max value of the matching leastN type.

INT_LEASTN_MIN

min value of the matching leastN type.

UINT_LEASTN_MAX

unsigned maximum of the matching N type.

int_fastN_t

similar to int_leastN_t but asking for the "fastest" with at least that precision.

uint_fastN_t

unsigned fastest least integer.

INT_FASTN_MAX

max value of the matching fastestN type.

INT_FASTN_MIN

min value of the matching fastestN type.

UINT_FASTN_MAX

unsigned max value of the matching fastestN type.

intptr_t

a signed integer large enough to hold a pointer.

`uintptr_t`

an unsigned integer large enough to hold a pointer.

`INTPTR_MAX`

max value of a `intptr_t`.

`INTPTR_MIN`

min value of a `intptr_t`.

`UINTPTR_MAX`

unsigned max value of a `uintptr_t`.

`intmax_t`

biggest number possible on that system.

`uintmax_t`

biggest unsigned number possible.

`INTMAX_MAX`

largest value for the biggest signed number.

`INTMAX_MIN`

smallest value for the biggest signed number.

`UINTMAX_MAX`

largest value for the biggest unsigned number.

`PTRDIFF_MIN`

minimum value of `ptrdiff_t`.

`PTRDIFF_MAX`

maximum value of `ptrdiff_t`.

`SIZE_MAX`

maximum of a `size_t`.

22.2 Available Operators

This is a comprehensive list of all the operators you have in the C language. In this list, I'm indicating the following:

(binary)

The operator has a left and right: $X + Y$.

(unary)

The operator is on its own: $-X$.

(prefix)

The operator comes before the variable: $++X$.

(postfix)

Usually the same as the (prefix) version, but placing it after gives it a different meaning: $X++$.

(ternary)

There's only one of these, so it's actually called the ternary but it means "three operands": $X ? Y : Z$.

22.2.1 Math Operators

These are your basic math operations, plus I put `()` in with these since it calls a function and is close to a "math" operation.

`()`

Function call.

- * (binary)
multiply.
- /
divide.
- + (binary)
addition.
- + (unary)
positive number.
- ++ (postfix)
read, then increment.
- ++ (prefix)
increment, then read.
- (postfix)
read, then decrement.
- (prefix)
decrement, then read.
- (binary)
subtract.
- (unary)
negative number.

22.2.2 Data Operators

These are used to access data in different ways and forms.

- >
struct pointer access.
- .
struct value access.
- [] Array index.
- sizeof
size of a type or variable.
- & (unary)
Address of.
- * (unary)
Value of.

22.2.3 Logic Operators

These handle testing equality and inequality of variables.

- !=
does not equal.
- <
less than.
- <=
less than or equal.

`==`
equal (not assignment).
`>`
greater than.
`>=`
greater than or equal.

22.2.4 Bit Operators

These are more advanced and for shifting and modifying the raw bits in integers.

`&` (binary)
Bitwise and.
`<<`
Shift left.
`>>`
Shift right.
`^`
bitwise xor (exclusive or).
`|`
bitwise or.
`~`
compliment (flips all the bits).

22.2.5 Boolean Operators

Used in truth testing. Study the ternary operator carefully, it is very handy.

`!`
not.
`&&`
and.
`||`
or.
`?:`
Ternary truth test, read `X ? Y : Z` as "if X then Y else Z".

22.2.6 Assignment Operators

Compound assignment operators that assign a value, and/or perform an operation at the same time. Most of the above operations can also be combined into a compound assignment operator.

`=`
assign.
`%=`
modulus assign.
`&=`

bitwise and assign.

*=

multiply assign.

+=

plus assign.

-=

minus assign.

/=

divid assign.

<<=

shift left, assign.

>>=

shift right, assign.

^=

bitwise xor, assign.

|=

bitwise or, assign.

22.3 Available Control Structures

There's a few control structures you haven't encountered yet:

do-while

do { ... } while(X); First does the code in the block, then tests the X expression before exiting.

break

Put this in a loop, and it breaks out ending it early.

continue

Stops the body of a loop and jumps to the test so it can continue.

goto

Jumps to a spot in the code where you've placed a label:, and you've been using this in the dbg.h macros to go to the error: label.

22.3.1 Extra Credit

1. Read stdint.h or a description of it and write out all the possible available size identifiers.
2. Go through each item here and write out what it does in code. Research it so you know you got it right by looking it up online.
3. Get this information solid as well by making flash cards and spending 15 minutes a day memorizing it.
4. Create a program that prints out examples of each type and confirm that your research is right.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 23

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 22: The Stack, Scope, And Globals

The concept of "scope" seems to confuse quite a few people when they first start programming. Originally it came from the use of the system stack (which we lightly covered earlier) and how it was used to store temporary variables. In this exercise, we'll learn about scope by learning about how a stack data structure works, and then feeding that concept back in to how modern C does scoping.

The real purpose of this exercise though is to learn where the hell things live in C. When someone doesn't grasp the concept of scope, it's almost always a failure in understanding where variables are created, exist, and die. Once you know where things are, the concept of scope becomes easier.

This exercise will require three files:

ex22.h

A header file that sets up some external variables and some functions.

ex22.c

Not your main like normal, but instead a source file that will become a object file ex22.o which will have some functions and variables in it defined from ex22.h.

ex22_main.c

The actual main that will include the other two and demonstrate what they contain as well as other scope concepts.

23.0.2 ex22.h and ex22.c

Your first step is to create your own header file named ex22.h which defines the functions and "extern" variables you need:

Source 58: ex22.h

```

1  #ifndef _ex22_h
2  #define _ex22_h
3
4  // makes THE_SIZE in ex22.c available to other .c files
5  extern int THE_SIZE;
6
7  // gets and sets an internal static variable in ex22.c
8  int get_age();
9  void set_age(int age);
10
11 // updates a static variable that's inside update_ratio
12 double update_ratio(double ratio);
13
14 void print_size();
15
16 #endif

```

The important thing to see is the use of `extern int THE_SIZE`, which I'll explain after you also create the matching `ex22.c`:

Source 59: `ex22.c`

```

1  #include <stdio.h>
2  #include "ex22.h"
3  #include "dbg.h"
4
5  int THE_SIZE = 1000;
6
7  static int THE_AGE = 37;
8
9  int get_age()
10 {
11     return THE_AGE;
12 }
13
14 void set_age(int age)
15 {
16     THE_AGE = age;
17 }
18
19
20 double update_ratio(double new_ratio)
21 {
22     static double ratio = 1.0;
23
24     double old_ratio = ratio;
25     ratio = new_ratio;
26
27     return old_ratio;
28 }
29
30 void print_size()
31 {
32     log_info("I think size is: %d", THE_SIZE);
33 }
```

These two files introduce some new kinds of storage for variables:

`extern`

This keyword is a way to tell the compiler "the variable exists, but it's in another 'external' location". Typically this means that one `.c` file is going to use a variable that's been defined in another `.c` file. In this case, we're saying `ex22.c` has a variable `THE_SIZE` that will be accessed from `ex22_main.c`.

`static` (file)

This keyword is kind of the inverse of `extern` and says that the variable is only used in this `.c` file, and should not be available to other parts of the program. Keep in mind that `static` at the file level (as with `THE_AGE` here) is different than in other places.

`static` (function)

If you declare a variable in a function `static`, then that variable acts like a `static` defined in the file, but it's only accessible from that function. It's a way of creating constant state for a function, but in reality it's rarely used in modern C programming because they are hard to use with threads.

In these two files then, you have the following variables and functions that you should understand:

`THE_SIZE`

This is the variable you declared `extern` that you'll play with from `ex22_main.c`.

`get_age` and `set_age`

These are taking the `static` variable `THE_AGE`, but exposing it to other parts of the program through functions. You couldn't access `THE_AGE` directly, but these functions can.

`update_ratio`

This takes a new ratio value, and returns the old one. It uses a function level `static` variable `ratio` to keep track of what

the ratio currently is.

print_size

Prints out what ex22.c thinks THE_SIZE is currently.

23.0.3 ex22_main.c

Once you have that file written, you can then make the main function which uses all of these and demonstrates some more scope conventions:

Source 60: ex22_main.c

```

1  #include "ex22.h"
2  #include "dbg.h"
3
4  const char *MY_NAME = "Zed A. Shaw";
5
6  void scope_demo(int count)
7  {
8      log_info("count is: %d", count);
9
10     if(count > 10) {
11         int count = 100; // BAD! BUGS!
12
13         log_info("count in this scope is %d", count);
14     }
15
16     log_info("count is at exit: %d", count);
17
18     count = 3000;
19
20     log_info("count after assign: %d", count);
21 }
22
23 int main(int argc, char *argv[])
24 {
25     // test out THE_AGE accessors
26     log_info("My name: %s, age: %d", MY_NAME, get_age());
27
28     set_age(100);
29
30     log_info("My age is now: %d", get_age());
31
32     // test out THE_SIZE extern
33     log_info("THE_SIZE is: %d", THE_SIZE);
34     print_size();
35
36     THE_SIZE = 9;
37
38     log_info("THE SIZE is now: %d", THE_SIZE);
39     print_size();
40
41     // test the ratio function static
42     log_info("Ratio at first: %f", update_ratio(2.0));
43     log_info("Ratio again: %f", update_ratio(10.0));
44     log_info("Ratio once more: %f", update_ratio(300.0));
45
46     // test the scope demo
47     int count = 4;
48     scope_demo(count);
49     scope_demo(count * 20);
50
51     log_info("count after calling scope_demo: %d", count);
52
53     return 0;
54 }

```

I'll break this file down line-by-line, and as I do you should find each variable I mention and where it lives.

ex22_main.c:4

Making a const which stands for constant and is an alternative to using a define to create a constant variable.

ex22_main.c:6

A simple function that demonstrates more scope issues in a function.

ex22_main.c:8

Prints out the value of count as it is at the top of the function.

ex22_main.c:10

An if-statement that starts a new scope block, and then has another count variable in it. This version of count is actually a whole new variable. It's kind of like the if-statement started a new "mini function".

ex22_main.c:11

The count that is local to this block is actually different from the one in the function's parameter list. What happens as we continue.

ex22_main.c:13

Prints it out so you can see it's actually 100 here, not what was passed to scope_demo.

ex22_main.c:16

Now for the freaky part. You have count in two places: the parameters to this function, and in the if-statement. The if-statement created a new block, so the count on line 11 does not impact the parameter with the same name. This line prints it out and you'll see that it prints the value of the parameter, not 100.

ex22_main.c:18-20

Then I set the parameter count to 3000 and print that out, which will demonstrate that you can change function parameters and they don't impact the caller's version of the variable.

Make sure you trace through this function, but don't think that you understand scope quite yet. Just start to realize that if you make a variable inside a block (as in if-statements or while-loops), then those variables are new variables that exist only in that block. This is crucial to understand, and is also a source of many bugs. We'll address why you shouldn't do this shortly.

The rest of the ex22_main.c then demonstrates all of these by manipulating and printing them out:

ex22_main.c:26

Prints out the current values of MY_NAME and gets THE_AGE from ex22.c using the accessor function get_age.

ex22_main.c:27-30

Uses set_age in ex22.c to change THE_AGE and then print it out.

ex22_main.c:33-39

Then I do the same thing to THE_SIZE from ex22.c, but this time I'm accessing it directly, and also demonstrating that it's actually changing in that file by printing it here and with print_size.

ex22_main.c:42-44

Show how the static variable ratio inside update_ratio is maintained between function calls.

ex22_main.c:46-51

Finally running scope_demo a few times so you can see the scope in action. Big thing to notice is that the local count variable remains unchanged. You must get that passing in a variable like this will not let you change it in the function. To do that you need our old friend the pointer. If you were to pass a pointer to this count, then the called function has the address of it and can change it.

That explains what's going on in all of these files, but you should trace through them and make sure you know where everything is as you study it.

23.1 What You Should See

This time, instead of using your Makefile I want you to build these two files manually so you can see how they are actually put together by the compiler. Here's what you should do and what you should see for output.

Source 61: ex22 output

```

1  $ cc -Wall -g -DNDEBUG -c -o ex22.o ex22.c
2  $ cc -Wall -g -DNDEBUG ex22_main.c ex22.o -o ex22_main
3  $ ./ex22_main
4  [INFO] (ex22_main.c:26) My name: Zed A. Shaw, age: 37
5  [INFO] (ex22_main.c:30) My age is now: 100
6  [INFO] (ex22_main.c:33) THE_SIZE is: 1000
7  [INFO] (ex22.c:32) I think size is: 1000
8  [INFO] (ex22_main.c:38) THE SIZE is now: 9
9  [INFO] (ex22.c:32) I think size is: 9
10 [INFO] (ex22_main.c:42) Ratio at first: 1.000000
11 [INFO] (ex22_main.c:43) Ratio again: 2.000000
12 [INFO] (ex22_main.c:44) Ratio once more: 10.000000
13 [INFO] (ex22_main.c:8) count is: 4
14 [INFO] (ex22_main.c:16) count is at exit: 4
15 [INFO] (ex22_main.c:20) count after assign: 3000
16 [INFO] (ex22_main.c:8) count is: 80
17 [INFO] (ex22_main.c:13) count in this scope is 100
18 [INFO] (ex22_main.c:16) count is at exit: 80
19 [INFO] (ex22_main.c:20) count after assign: 3000
20 [INFO] (ex22_main.c:51) count after calling scope_demo: 4

```

Make sure you trace how each variable is changing and match it to the line that gets output. I'm using `log_info` from the `dbg.h` macros so you can get the exact line number where each variable is printed and find it in the files for tracing.

23.2 Scope, Stack, And Bugs

If you've done this right you should now see many of the different ways you can place variables in your C code. You can use `extern` or access functions like `get_age` to create globals. You can make new variables inside any blocks, and they'll retain their own values until that block exits, leaving the outer variables alone. You also can pass a value to a function, and change the parameter but not change the caller's version of it.

The most important thing to realize though is that all of this causes bugs. C's ability to place things in many places in your machine and then let you access it in those places means you get confused easily about where something lives. If you don't where it lives then there's a chance you'll not manage it properly.

With that in mind, here's some rules to follow when writing C code so you avoid bugs related to the stack:

1. Do not "shadow" a variable like I've done here with `count` in `scope_demo`. It leaves you open to subtle and hidden bugs where you think you're changing a value and you actually aren't.
2. Avoid too many globals, especially if across multiple files. If you have to then use accessor functions like I've done with `get_age`. This doesn't apply to constants, since those are read-only. I'm talking about variables like `THE_SIZE`. If you want people to modify or set this, then make accessor functions.
3. When in doubt, put it on the heap. Don't rely on the semantics of the stack or specialized locations and instead just create things with `malloc`.
4. Don't use function static variables like I did in `update_ratio`. They're rarely useful and end up being a huge pain when you need to make your code concurrent in threads. They are also hard as hell to find compared to a well done global variable.
5. Avoid reusing function parameters as it's confusing whether you're just reusing it or if you think you're changing the caller's version of it.

As with all things, these rules can be broken when it's practical. In fact, I guarantee you'll run into code that breaks all of these rules and is perfectly fine. The constraints of different platforms makes it necessary sometimes.

23.3 How To Break It

For this exercise, breaking the program involves trying to access or change things you can't:

1. Try to directly access variables in `ex22.c` from `ex22_main.c` that you think you can't. For example, you can't get at `ratio` inside `update_ratio`? What if you had a pointer to it?
2. Ditch the extern declaration in `ex22.h` to see what you get for errors or warnings.
3. Add static or const specifiers to different variables and then try to change them.

23.4 Extra Credit

1. Research the concept of "pass by value" vs. "pass by reference". Write an example of both.
2. Use pointers to gain access to things you shouldn't have access to.
3. Use `valgrind` to see what this kind of access looks like when you do it wrong.
4. Write a recursive function that causes a stack overflow. Don't know what a recursive function is? Try calling `scope_demo` at the bottom of `scope_demo` itself so that it loops.
5. Rewrite the Makefile so that it can build this.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 24

Exercise 23: Meet Duff's Device

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

This exercise is a brain teaser where I introduce you to one of the most famous hacks in C called "Duff's Device", named after Tom Duff the "inventor". This little slice of awesome (evil?) has nearly everything you've been learning wrapped in one tiny little package. Figuring out how it works is also a good fun puzzle.

Note 6: This Is Only An Exercise
 Part of the fun of C is that you can come up with crazy hacks like this, but this is also what makes C annoying to use. It's good to learn about these tricks because it gives you a deeper understanding of the language and your computer. But, you should never use this. Always strive for easy to read code.

Duff's device was "discovered" (created?) by Tom Duff and is a trick with the C compiler that actually shouldn't work. I won't tell you what it does yet since this is meant to be a puzzle for you to ponder and try to solve. You are to get this code running and then try to figure out what it does, and why it does it this way.

Source 62: ex23.c

```

1  #include <stdio.h>
2  #include <string.h>
3  #include "dbg.h"
4
5
6  int normal_copy(char *from, char *to, int count)
7  {
8      int i = 0;
9
10     for(i = 0; i < count; i++) {
11         to[i] = from[i];
12     }
13
14     return i;
15 }
16
17 int duffs_device(char *from, char *to, int count)
18 {
19     {
20         int n = (count + 7) / 8;
21
22         switch(count % 8) {
23             case 0: do { *to++ = *from++;
24                         case 7: *to++ = *from++;
25                         case 6: *to++ = *from++;
26                         case 5: *to++ = *from++;
27                         case 4: *to++ = *from++;
28                         case 3: *to++ = *from++;
29                         case 2: *to++ = *from++;
30                         case 1: *to++ = *from++;
31                     } while(--n > 0);
32         }
33     }
34 }
```

```

35     return count;
36 }
37
38 int zeds_device(char *from, char *to, int count)
39 {
40     {
41         int n = (count + 7) / 8;
42
43         switch(count % 8) {
44             case 0:
45                 again: *to++ = *from++;
46
47             case 7: *to++ = *from++;
48             case 6: *to++ = *from++;
49             case 5: *to++ = *from++;
50             case 4: *to++ = *from++;
51             case 3: *to++ = *from++;
52             case 2: *to++ = *from++;
53             case 1: *to++ = *from++;
54             if(--n > 0) goto again;
55         }
56     }
57
58     return count;
59 }
60
61 int valid_copy(char *data, int count, char expects)
62 {
63     int i = 0;
64     for(i = 0; i < count; i++) {
65         if(data[i] != expects) {
66             log_err("[%d] %c != %c", i, data[i], expects);
67             return 0;
68         }
69     }
70
71     return 1;
72 }
73
74
75 int main(int argc, char *argv[])
76 {
77     char from[1000] = {'a'};
78     char to[1000] = {'c'};
79     int rc = 0;
80
81     // setup the from to have some stuff
82     memset(from, 'x', 1000);
83     // set it to a failure mode
84     memset(to, 'y', 1000);
85     check(valid_copy(to, 1000, 'y'), "Not initialized right.");
86
87     // use normal copy to
88     rc = normal_copy(from, to, 1000);
89     check(rc == 1000, "Normal copy failed: %d", rc);
90     check(valid_copy(to, 1000, 'x'), "Normal copy failed.");
91
92     // reset
93     memset(to, 'y', 1000);
94
95     // duffs version
96     rc = duffs_device(from, to, 1000);
97     check(rc == 1000, "Duff's device failed: %d", rc);
98     check(valid_copy(to, 1000, 'x'), "Duff's device failed copy.");
99
100    // reset
101    memset(to, 'y', 1000);
102
103    // my version
104    rc = zeds_device(from, to, 1000);
105    check(rc == 1000, "Zed's device failed: %d", rc);
106    check(valid_copy(to, 1000, 'x'), "Zed's device failed copy.");
107
108    return 0;
109 error:
110    return 1;
111 }

```

In this code I have three versions of a copy function:

normal_copy

Which is just a plain for-loop that copies characters from one array to another.

duffs_device

This is the brain teaser called "Duff's Device", named after Tom Duff, the person to blame for this delicious evil.

zeds_device

A version of "Duff's Device" that just uses a goto so you can get a clue about what's happening with the weird do-while placement in duffs_device.

Study these three functions before continuing. Try to explain what's going on to yourself before continuing.

24.1 What You Should See

There's no output from this program, it just runs and exits. You should run it under valgrind and make sure there are no errors.

24.2 Solving The Puzzle

The first thing to understand is that C is rather loose regarding some of its syntax. This is why you can put half of a do-while in one part of a switch-statement, then the other half somewhere else and it will still work. If you look at my version with the goto again it's actually more clear what's going on, but make sure you understand how that part works.

The second thing is how the default fallthrough semantics of switch-statements means you can jump to a particular case, and then it will just keep running until the end of the switch.

The final clue is the count % 8 and the calculation of n at the top.

Now, to solve how these functions work, do the following:

1. Print this code out so you can write on some paper.
2. On a piece of paper, write each of the variables in a table as they are when they get initialized right before the switch-statement.
3. Follow the logic to the switch, then do the jump to the right case.
4. Update the variables, including the to, from, and the arrays they point at.
5. When you get to the while part or my goto alternative, check your variables and then follow the logic either back to the top of the do-while or to where the again label is located.
6. Follow through this manual tracing, updating the variables, until you are sure you see how this flows.

24.2.1 Why Bother?

When you've figured out how it actually works, the final question is: Why would you ever want to do this? The purpose of this trick is to manually do "loop unrolling". Large long loops can be slow, so one way to speed them up is to find some fixed chunk of the loop, and then just duplicate the code in the loop out that many times sequentially. For example, if you know a loop runs a minimum of 20 times, then you can put the contents of the loop 20 times in the source code.

Duff's device is basically doing this automatically by chunking up the loop into 8 iteration chunks. It's clever and actually works, but these days a good compiler will do this for you. You shouldn't need this except in the rare case where you have proven it would improve your speed.

24.3 Extra Credit

1. Never use this again.
2. Go look at the Wikipedia entry for "Duff's Device" and see if you can spot the error. Compare it to the version I have here and read the article carefully to try to understand why the Wikipedia code won't work for you but worked for Tom Duff.
3. Create a set of macros that lets you create any length device like this. For example, what if you wanted to have 32 case statements and didn't want to write out all of them? Can you do a macro that lays down 8 at a time?
4. Change the main to conduct some speed tests to see which one is really the fastest.
5. Read about memcpy, memmove, memset, and also compare their speed.
6. Never use this again!

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 25

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 24: Input, Output, Files

You've been using `printf` to print things, and that's great and all, but you need more. In this exercise program you're using the functions `fscanf` and `fgets` to build information about a person in a structure. After this simple introduction to reading input, you'll get a full list of the functions that C has for I/O. Some of these you've already seen and used, so this will be another memorization exercise.

Source 63: ex24.c

```

1  #include <stdio.h>
2  #include "dbg.h"
3
4  #define MAX_DATA 100
5
6  typedef enum EyeColor {
7      BLUE_EYES, GREEN_EYES, BROWN_EYES,
8      BLACK_EYES, OTHER_EYES
9  } EyeColor;
10
11 const char *EYE_COLOR_NAMES[] = {
12     "Blue", "Green", "Brown", "Black", "Other"
13 };
14
15 typedef struct Person {
16     int age;
17     char first_name[MAX_DATA];
18     char last_name[MAX_DATA];
19     EyeColor eyes;
20     float income;
21 } Person;
22
23
24 int main(int argc, char *argv[])
25 {
26     Person you = {.age = 0};
27     int i = 0;
28     char *in = NULL;
29
30     printf("What's your First Name? ");
31     in = fgets(you.first_name, MAX_DATA-1, stdin);
32     check(in != NULL, "Failed to read first name.");
33
34     printf("What's your Last Name? ");
35     in = fgets(you.last_name, MAX_DATA-1, stdin);
36     check(in != NULL, "Failed to read last name.");
37
38     printf("How old are you? ");
39     int rc = fscanf(stdin, "%d", &you.age);
40     check(rc > 0, "You have to enter a number.");
41
42     printf("What color are your eyes:\n");
43     for(i = 0; i <= OTHER_EYES; i++) {
44         printf("%d) %s\n", i+1, EYE_COLOR_NAMES[i]);
45     }
46     printf("> ");
47
48     int eyes = -1;
```

```

49 rc = fscanf(stdin, "%d", &eyes);
50 check(rc > 0, "You have to enter a number.");
51
52 you.eyes = eyes - 1;
53 check(you.eyes <= OTHER_EYES && you.eyes >= 0, "Do it right, that's not an option.");
54
55 printf("How much do you make an hour? ");
56 rc = fscanf(stdin, "%f", &you.income);
57 check(rc > 0, "Enter a floating point number.");
58
59 printf("----- RESULTS ----- \n");
60
61 printf("First Name: %s", you.first_name);
62 printf("Last Name: %s", you.last_name);
63 printf("Age: %d \n", you.age);
64 printf("Eyes: %s \n", EYE_COLOR_NAMES[you.eyes]);
65 printf("Income: %f \n", you.income);
66
67 return 0;
68 error:
69
70 return -1;
71 }

```

This program is deceptively simple, and introduces a function called `fscanf` which is the "file scanf". The `scanf` family of functions are the inverse of the `printf` versions. Where `printf` printed out data based on a format, `scanf` reads (or scans) input based on a format.

There's nothing original in the beginning of the file, so here's what the main is doing:

ex24.c:24–28

Set up some variables we'll need.

ex24.c:30–32

Get your first name using the `fgets` function, which reads a string from the input (in this case `stdin`) but makes sure it doesn't overflow the given buffer.

ex24.c:34–36

Same thing for `you.last_name`, again using `fgets`.

ex24.c:38–39

Uses `fscanf` to read an integer from `stdin` and put it into `you.age`. You can see that the same format string is used as `printf` to print an integer. You should also see that you have to give the address of `you.age` so that `fscanf` has a pointer to it and can modify it. This is a good example of using a pointer to a piece of data as an "out parameter".

ex24.c:41–45

Print out all the options available for eye color, with a matching number that works with the `EyeColor` enum above.

ex24.c:47–50

Using `fscanf` again, get a number for the `you.eyes`, but make sure the input is valid. This is important because someone can enter a value outside the `EYE_COLOR_NAMES` array and cause a segfault.

ex24.c:52–53

Get how much you make as a float for the `you.income`.

ex24.c:55–61

Print everything out so you can see if you have it right. Notice that `EYE_COLOR_NAMES` is used to print out what the `EyeColor` enum is actually called.

25.1 What You Should See

When you run this program you should see your inputs being properly converted. Make sure you try to give it bogus input too so you can see how it protects against the input.

Source 64: ex24 output

```

1  $ make ex24
2  cc -Wall -g -DNDEBUG  ex24.c  -o ex24
3  $ ./ex24
4  What's your First Name? Zed
5  What's your Last Name? Shaw
6  How old are you? 37
7  What color are your eyes:
8  1) Blue
9  2) Green
10 3) Brown
11 4) Black
12 5) Other
13 > 1
14 How much do you make an hour? 1.2345
15 ----- RESULTS -----
16 First Name: Zed
17 Last Name: Shaw
18 Age: 37
19 Eyes: Blue
20 Income: 1.234500

```

25.2 How To Break It

This is all fine and good, but the real important part of this exercise is how `scanf` actually sucks. It's fine for simple conversion of numbers, but fails for strings because it's difficult to tell `scanf` how big a buffer is before you read. There's also a problem with a function like `gets` (not `fgets`, the non-`f` version) which we avoided. That function has no idea how big the input buffer is at all and will just trash your program.

To demonstrate the problems with `fscanf` and strings, change the lines that use `fgets` so they are `fscanf(stdin, "%50s", you.first_name)` and then try to use it again. Notice it seems to read too much and then eat your enter key? This doesn't do what you think it does, and really rather than deal with weird `scanf` issues, just use `fgets`.

Next, change the `fgets` to use `gets`, then bust out your `valgrind` and do this: `valgrind ./ex24 < /dev/urandom` to feed random garbage into your program. This is called "fuzzing" your program, and it is a good way to find input bugs. In this case, you're feeding garbage from the `/dev/urandom` file, and then watching it crash. On some platforms you may have to do this a few times, or even adjust the `MAX_DATA` define so it's small enough.

The `gets` function is so bad that some platforms actually warn you when the program runs that you're using `gets`. You should never use this function, ever.

Finally, take the input for `you.eyes` and remove the check that the number given is within the right range. Then feed it bad numbers like `-1` or `1000`. Do this under `Valgrind` too so you can see what happens.

25.3 The I/O Functions

This is a short list of various I/O functions that you should look up and create index cards that have the function name, what it does, and all the variants similar to it.

1. `fscanf`
2. `fgets`
3. `fopen`
4. `freopen`
5. `fdopen`
6. `fclose`
7. `fcloseall`
8. `fgetpos`

9. fseek
10. ftell
11. rewind
12. fprintf
13. fwrite
14. fread

Go through these and memorize the different variants and what they do. For example, for the card on fscanf you'll have scanf, sscanf, vscanf, etc. and then what each of those do on the back.

Finally, to get the information you need for these cards, use man to read the help for it. For example, the page for fscanf comes from man fscanf.

25.4 Extra Credit

1. Rewrite this to not use fscanf at all. You'll need to use functions like atoi to convert the input strings to numbers.
2. Change this to use plain scanf instead of fscanf to see what the difference is.
3. Fix it so that the input names get stripped of the trailing newline characters and any whitespace.
4. Use scanf to write a function that reads a character at a time and files in the names but doesn't go past the end. Make this function generic so it can take a size for the string, and make sure you end the string with '\0' no matter what.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 26

Exercise 25: Variable Argument Functions

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

In C you can create your own versions of functions like printf and scanf by creating a "variable argument function". These functions use the header stdarg.h and with them you can create nicer interfaces to your library. They are handy for certain types of "builder" functions, formatting functions, and anything that takes variable arguments.

Understanding "vararg functions" is not essential to creating C programs. I think I've used it maybe a 20 times in my code in the years I've been programming. However, knowing how a vararg function works will help you debug the ones you use and gives you more understanding of the computer.

Source 65: ex25.c

```

1  /** WARNING: This code is fresh and potentially isn't correct yet. */
2
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <stdarg.h>
6  #include "dbg.h"
7
8  #define MAX_DATA 100
9
10 int read_string(char **out_string, int max_buffer)
11 {
12     *out_string = calloc(1, max_buffer + 1);
13     check_mem(*out_string);
14
15     char *result = fgets(*out_string, max_buffer, stdin);
16     check(result != NULL, "Input error.");
17
18     return 0;
19
20 error:
21     if(*out_string) free(*out_string);
22     *out_string = NULL;
23     return -1;
24 }
25
26 int read_int(int *out_int)
27 {
28     char *input = NULL;
29     int rc = read_string(&input, MAX_DATA);
30     check(rc == 0, "Failed to read number.");
31
32     *out_int = atoi(input);
33
34     free(input);
35     return 0;
36
37 error:
38     if(input) free(input);
39     return -1;
40 }
41
42 int read_scan(const char *fmt, ...)
43 {
44

```

```

    int i = 0;
45    int rc = 0;
46    int *out_int = NULL;
47    char *out_char = NULL;
48    char **out_string = NULL;
49    int max_buffer = 0;
50
51    va_list argp;
52    va_start(argp, fmt);
53
54    for(i = 0; fmt[i] != '\0'; i++) {
55        if(fmt[i] == '%') {
56            i++;
57            switch(fmt[i]) {
58                case '\0':
59                    sentinel("Invalid format, you ended with %%.");
60                    break;
61
62                case 'd':
63                    out_int = va_arg(argp, int *);
64                    rc = read_int(out_int);
65                    check(rc == 0, "Failed to read int.");
66                    break;
67
68                case 'c':
69                    out_char = va_arg(argp, char *);
70                    *out_char = fgetc(stdin);
71                    break;
72
73                case 's':
74                    max_buffer = va_arg(argp, int);
75                    out_string = va_arg(argp, char **);
76                    rc = read_string(out_string, max_buffer);
77                    check(rc == 0, "Failed to read string.");
78                    break;
79
80                default:
81                    sentinel("Invalid format.");
82            }
83        } else {
84            fgetc(stdin);
85        }
86    }
87    check(!feof(stdin) && !ferror(stdin), "Input error.");
88 }
89
90 va_end(argp);
91 return 0;
92
93 error:
94     va_end(argp);
95     return -1;
96 }
97
98
99
100 int main(int argc, char *argv[])
101 {
102     char *first_name = NULL;
103     char initial = '\0';
104     char *last_name = NULL;
105     int age = 0;
106
107     printf("What's your first name? ");
108     int rc = read_scan("%s", MAX_DATA, &first_name);
109     check(rc == 0, "Failed first name.");
110
111     printf("What's your initial? ");
112     rc = read_scan("%c\n", &initial);
113     check(rc == 0, "Failed initial.");
114
115     printf("What's your last name? ");
116     rc = read_scan("%s", MAX_DATA, &last_name);
117     check(rc == 0, "Failed last name.");
118
119     printf("How old are you? ");
120     rc = read_scan("%d", &age);
121
122     printf("---- RESULTS ----\n");
123     printf("First Name: %s", first_name);
124     printf("Initial: '%c'\n", initial);
125     printf("Last Name: %s", last_name);

```

```

126 printf("Age: %d\n", age);
127
128 free(first_name);
129 free(last_name);
130 return 0;
131 error:
132 return -1;
133 }

```

This program is similar to the previous exercise, except I have written my own `scanf` style function that handles strings the way I want. The main function should be clear to you, as well as the two functions `read_string` and `read_int` since they do nothing new.

The `varargs` function is called `read_scan` and it does the same thing that `scanf` is doing using the `va_list` data structure and it's supporting macros and functions. Here's how it works:

1. I set as the last parameter of the function the keyword `...` which indicates to C that this function will take any number of arguments after the `fmt` argument. I could put many other arguments before this, but I can't put anymore after this.
2. After setting up some variables, I create a `va_list` variable and initialize it with `va_start`. This configures the gear in `stdarg.h` that handles variable arguments.
3. I then use a `for`-loop to loop through the format string `fmt` and process the same kind of formats that `scanf` has, but much simpler. I just have integers, characters, and strings.
4. When I hit a format, I use the `switch`-statement to figure out what to do.
5. Now, to get a variable from the `va_list` `argp` I use the macro `va_arg(argp, TYPE)` where `TYPE` is the exact type of what I will assign this function parameter to. The downside to this design is you're flying blind, so if you don't have enough parameters then oh well, you'll most likely crash.
6. The interesting difference from `scanf` is I'm assuming that people want `read_scan` to create the strings it reads when it hits a 's' format sequence. When you give this sequence, the function takes two parameters off the `va_list` `argp` stack: the max function size to read, and the output character string pointer. Using that information it just runs `read_string` to do the real work.
7. This makes `read_scan` more consistent than `scanf` since you always give an address-of `&` on variables to have them set appropriately.
8. Finally, if it encounters a character that's not in the format, it just reads one char to skip it. It doesn't care what that char is, just that it should skip it.

26.1 What You Should See

When you run this one it's similar to the last one:

Source 66: `ex25.out`

```

1  $ make ex25
2  cc -Wall -g -DNDEBUG  ex25.c  -o ex25
3  $ ./ex25
4  What's your first name? Zed
5  What's your initial? A
6  What's your last name? Shaw
7  How old are you? 37
8  ---- RESULTS ----
9  First Name: Zed
10 Initial: 'A'
11 Last Name: Shaw
12 Age: 37

```

26.2 How To Break It

This program should be more robust against buffer overflows, but it doesn't handle the formatted input as well as `scanf`. To try breaking this, change the code that you forget to pass in the initial size for `'%s'` formats. Try also giving it more data than `MAX_DATA`, and then see how not using `calloc` in `read_string` changes how it works. Finally, there's a problem that `fgets` eats the newlines, so try to fix that using `fgetc` but leave out the `'\0'` that ends the string.

26.3 Extra Credit

1. Make double and triple sure that you know what each of the `out_` variables are doing. Most important is `out_string` and how it's a pointer to a pointer, so getting when you're setting the pointer vs. the contents is important. Break down each of the
2. Write a similar function to `printf` that uses the `varargs` system and rewrite `main` to use it.
3. As usual, read the man page on all of this so you know what it does on your platform. Some platforms will use macros and others use functions, and some have these do nothing. It all depends on the compiler and the platform you use.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 27

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 26: Write A First Real Program

You are at the half-way mark in the book, so you need to take a mid-term. In this mid-term you're going to recreate a piece of software I wrote specifically for this book called devpkg. You'll then extend it in a few key ways and improve the code, most importantly by writing some unit tests for it.

Note 7: WARNING: Beta Draft Content

I wrote this exercise before writing some of the exercises you might need to complete this. If you are attempting this one now, please keep in mind that the software may have bugs, that you might have problems because of my mistakes, and that you might not know everything you need to finish it. If so, tell me at help@learncodethehardway.org and then wait until I finish the other exercises.

27.1 What Is devpkg?

Devpkg is a simple C program that installs other software. I made it specifically for this book as a way to teach you how a real software project is structured, and also how to reuse other people's libraries. It uses a portability library called [The Apache Portable Runtime \(APR\)](#) that has many handy C functions which work on tons of platforms, including Windows. Other than that, it just grabs code from the internet (or local files) and does the usual `./configure ; make ; make install` every programmer does.

Your goal in this exercise is to build devpkg from source, finish each Challenge I give, and use the source to understand what devpkg does and why.

27.1.1 What We Want To Make

We want a tool that has three commands:

1. Sets up a new install on a computer.
2. Installs a piece of software from a URL.
3. Lists all the software that's been installed.
4. Fetches some source code for manual building.
5. Builds fetches source code and installs it, even if already installed.

We want devpkg to be able to take almost any URL, figure out what kind of project it is, download it, install it, and register that it downloaded that software. We'd also like it to process a simple dependency list so it can install all the software that a project might need as well.

27.1.2 The Design

To accomplish this goal devpkg will have a very simple design:

Use external commands

You'll do most of the work through external commands like curl, git, and tar. This reduces the amount of code devpkg needs to get things done.

Simple File Database

You could easily make it more complex, but to start you'll just make a single simple file database at /usr/local/.devpkg/db to keep track of what's installed.

/usr/local Always

Again you could make this more advanced, but for starters just assume it's always /usr/local which is a standard install path for most software on Unix.

configure, make, make install

It's assumed that most software can install with just a configure; make; make install and maybe configure is optional. If the software can't at a minimum do that, then there's some options to modify the commands, but otherwise devpkg won't bother.

The User Can Be root

We'll assume the user can become root using sudo, but that they don't want to become root until the end.

This will keep our program small at first and work well enough to get it going, at which point you'll be able to modify it further for this exercise.

27.1.3 The Apache Portable Runtime

One more thing you'll do is leverage the [The Apache Portable Runtime \(APR\)](#) libraries to get a good set of portable routines for doing this kind of work. The APR isn't necessary, and you could probably write this program without them, but it'd take more code than necessary. I'm also forcing you to use APR now so you get used to linking and using other libraries. Finally, the APR also works on Windows so your skills with it are transferable to many other platforms.

You should go get both the apr-1.4.5 and the apr-util-1.3 libraries, as well as browse through the documentation available at the [main APR site at apr.apache.org](#).

Here's a shell script that will install all the stuff you need. You should write this into a file by hand, and then run it until it can install APR without any errors.

Source 67: APR Install Script

```

1  set -e
2
3  # go somewhere safe
4  cd /tmp
5
6  # get the source to base APR 1.4.6
7  curl -L -O http://archive.apache.org/dist/apr/apr-1.4.6.tar.gz
8
9  # extract it and go into the source
10 tar -xzf apr-1.4.6.tar.gz
11 cd apr-1.4.6
12
13 # configure, make, make install
14 ./configure
15 make
16 sudo make install
17
18 # reset and cleanup
19 cd /tmp
20

```

```

rm -rf apr-1.4.6 apr-1.4.6.tar.gz
21
22 # do the same with apr-util
23 curl -L -O http://archive.apache.org/dist/apr/apr-util-1.4.1.tar.gz
24
25 # extract
26 tar -xzf apr-util-1.4.1.tar.gz
27 cd apr-util-1.4.1
28
29 # configure, make, make install
30 ./configure --with-apr=/usr/local/apr
31 # you need that extra parameter to configure because
32 # apr-util can't really find it because...who knows.
33
34 make
35 sudo make install
36
37 #cleanup
38 cd /tmp
39 rm -rf apr-util-1.4.1* apr-1.4.6*

```

I'm having you write this script out because this is basically what we want devpkg to do, but with extra options and checks. In fact, you could just do it all in shell with less code, but then that wouldn't be a very good program for a C book would it?

Simply run this script and fix it until it works, then you'll have the libraries you need to complete the rest of this project.

27.2 Project Layout

You need to setup some simple project files to get started. Here's how I usually craft a new project:

Source 68: Project Skeleton Directory

```

1 mkdir devpkg
2 cd devpkg
3 touch README Makefile

```

27.2.1 Other Dependencies

You should have already installed APR and APR-util, so now you need a few more files as basic dependencies:

1. dbg.h from Exercise 20.
2. bstrlib.h and bstrlib.c from <http://bstring.sourceforge.net/>. Download the .zip file, extract it, and copy just those two files out.
3. Type make bstrlib.o and if it doesn't work, read the "Fixing bstring" instructions below.

Note 8: Fixing bstring

In some platforms the bstring.c file will have an error like:

```
1 bstrlib.c:2762: error: expected declaration specifiers or '...' before numeric constant
```

This is from a bad define the authors added which doesn't work always. You just need to change the line 2759 that reads `#ifdef __GNUC__` and make it:

```
#if defined(__GNUC__) && !defined(__APPLE__)
```

Then it should work on Apple Mac OSX.

When that's all done, you should have a Makefile, README, dbg.h, bstrlib.h, and bstrlib.c ready to go.

27.3 The Makefile

A good place to start is the Makefile since this lays out how things are built and what source files you'll be creating.

Source 69: Makefile

```

1  PREFIX?=/usr/local
2  CFLAGS=-g -Wall -I${PREFIX}/apr/include/apr-1 -I${PREFIX}/apr/include/apr-util-1
3  LDFLAGS=-L${PREFIX}/apr/lib -lapr-1 -pthread -laprutil-1
4
5  all: devpkg
6
7  devpkg: bstrlib.o db.o shell.o commands.o
8
9  install: all
10     install -d $(DESTDIR)/$(PREFIX)/bin/
11     install devpkg $(DESTDIR)/$(PREFIX)/bin/
12
13  clean:
14     rm -f *.o
15     rm -f devpkg
16     rm -rf *.dSYM

```

There's nothing in this that you haven't seen before, except maybe the strange `?=` syntax, which says "set PREFIX equal to this unless PREFIX is already set".

Note 9: Ubuntu Annoyances

If you are on more recent versions of Ubuntu and you get errors about `apr_off_t` or `off64_t` then add `-D_LARGEFILE64_SOURCE=1` to `CFLAGS`.

Another thing is you need to add `/usr/local/apr/lib` to a file in `/etc/ld.conf.so.d/` then run `ldconfig` so that it picks up the libraries correctly.

27.4 The Source Files

From the make file, we see that there's four dependencies for `devpkg` which are:

`bstrlib.o`

Comes from `bstrlib.c` and header file `bstlib.h` which you already have.

`db.o`

From `db.c` and header file `db.h`, and it will contain code for our little "database" routines.

`shell.o`

From `shell.c` and header `shell.h`, with a couple functions that make running other commands like `curl` easier.

`commands.o`

From `command.c` and header `command.h`, and contains all the commands that `devpkg` needs to be useful.

`devpkg`

It's not explicitly mentioned, but instead is the target (on the left) in this part of the Makefile. It comes from `devpkg.c` which contains the main function for the whole program.

Your job is to now create each of these files and type in their code and get them correct.

Note 10: Don't Be Fooled By The Magic Show

You may read this description and think, "Man! How is it that Zed is so smart he just sat down and typed these files out like this!? I could never do that." I didn't magically craft `devpkg` in this form with my awesome code powers. Instead, what I did is this:

1. I wrote a quick little README to get an idea of how I wanted it to work.

2. I created a simple bash script (like the one you did) to figure out all the pieces that you need.
3. I made one .c file and hacked on it for a few days working through the idea and figuring it out.
4. I got it mostly working and debugged, then I started breaking up the one big file into these four files.
5. After getting these files laid down, I renamed and refined the functions and data structures so they'd be more logical and "pretty".
6. Finally, after I had it working the exact same but with the new structure, I added a few features like the -F and -B options.

You're reading this in the order I want to teach it to you, but don't think this is how I always build software. Sometimes I already know the subject and I use more planning. Sometimes I just hack up an idea and see how well it'd work. Sometimes I write one, then throw it away and plan out a better one. It all depends on what my experience tells me is best, or where my inspiration takes me.

If you run into an "expert" who tries to tell you there's only one way to solve a programming problem, then they're lying to you. Either they actually use multiple tactics, or they're not very good.

27.4.1 The DB Functions

There must be a way to record URLs that have been installed, list these URLs, and check if something has already been installed so we can skip it. I'll use a simple flat file database and the bstrlib.h library to do it.

First, create the db.h header file so you know what you'll be implementing.

Source 70: db.h

```

1  #ifndef _db_h
2  #define _db_h
3
4  #define DB_FILE "/usr/local/.devpkg/db"
5  #define DB_DIR "/usr/local/.devpkg"
6
7
8  int DB_init();
9  int DB_list();
10 int DB_update(const char *url);
11 int DB_find(const char *url);
12
13 #endif

```

Then implement those functions in db.c, as you build this, use make like you've been to get it to compile cleanly.

Source 71: db.c

```

1  #include <unistd.h>
2  #include <apr_errno.h>
3  #include <apr_file_io.h>
4
5  #include "db.h"
6  #include "bstrlib.h"
7  #include "dbg.h"
8
9  static FILE *DB_open(const char *path, const char *mode)
10 {
11     return fopen(path, mode);
12 }
13
14
15 static void DB_close(FILE *db)
16 {
17     fclose(db);
18 }
19
20
21 static bstring DB_load()
22 {
23     FILE *db = NULL;
24     bstring data = NULL;
25
26     db = DB_open(DB_FILE, "r");
27     check(db, "Failed to open database: %s", DB_FILE);
28
29     data = bread((bNread)fread, db);

```

```

30  check(data, "Failed to read from db file: %s", DB_FILE);
31
32  DB_close(db);
33  return data;
34
35  error:
36  if(db) DB_close(db);
37  if(data) bdestroy(data);
38  return NULL;
39 }
40
41
42 int DB_update(const char *url)
43 {
44     if(DB_find(url)) {
45         log_info("Already recorded as installed: %s", url);
46     }
47
48     FILE *db = DB_open(DB_FILE, "a+");
49     check(db, "Failed to open DB file: %s", DB_FILE);
50
51     bstring line = bfromcstr(url);
52     bconchar(line, '\n');
53     int rc = fwrite(line->data, blength(line), 1, db);
54     check(rc == 1, "Failed to append to the db.");
55
56     return 0;
57  error:
58  if(db) DB_close(db);
59  return -1;
60 }
61
62
63 int DB_find(const char *url)
64 {
65     bstring data = NULL;
66     bstring line = bfromcstr(url);
67     int res = -1;
68
69     data = DB_load(DB_FILE);
70     check(data, "Failed to load: %s", DB_FILE);
71
72     if(binstr(data, 0, line) == BSTR_ERR) {
73         res = 0;
74     } else {
75         res = 1;
76     }
77
78  error: // fallthrough
79  if(data) bdestroy(data);
80  if(line) bdestroy(line);
81
82  return res;
83 }
84
85
86 int DB_init()
87 {
88     apr_pool_t *p = NULL;
89     apr_pool_initialize();
90     apr_pool_create(&p, NULL);
91
92     if(access(DB_DIR, W_OK | X_OK) == -1) {
93         apr_status_t rc = apr_dir_make_recursive(DB_DIR,
94             APR_UREAD | APR_UWRITE | APR_UEXECUTE |
95             APR_GREAD | APR_GWRITE | APR_GEXECUTE, p);
96         check(rc == APR_SUCCESS, "Failed to make database dir: %s", DB_DIR);
97     }
98
99     if(access(DB_FILE, W_OK) == -1) {
100         FILE *db = DB_open(DB_FILE, "w");
101         check(db, "Cannot open database: %s", DB_FILE);
102         DB_close(db);
103     }
104
105     apr_pool_destroy(p);
106     return 0;
107
108  error:
109     apr_pool_destroy(p);
110     return -1;
111 }

```

```

112
113
114 int DB_list()
115 {
116     bstring data = DB_load(DB_FILE);
117     check(data, "Failed to read load: %s", DB_FILE);
118
119     printf("%s", bdata(data));
120     bdestroy(data);
121     return 0;
122
123 error:
124     return -1;
125 }

```

Challenge 1: Code Review

Before continuing, read every line of these files carefully and confirm that you have them entered in exactly. Read them line-by-line backwards to practice that. Also trace each function call and make sure you are using check to validate the return codes. Finally, look up every function that you don't recognize either on the APR web site documentation, or in the bstrlib.h and bstrlib.c source.

27.4.2 The Shell Functions

A key design decision for devpkg is to do most of the work using external tools like curl, tar, and git. We could find libraries to do all of this internally, but it's pointless if we just need the base features of these programs. There is no shame in running another command in Unix.

To do this I'm going to use the apr_thread_proc.h functions to run programs, but I also want to make a simple kind of "template" system. I'll use a struct Shell that holds all the information needed to run a program, but has "holes" in the arguments list where I can replace them with values.

Look at the shell.h file to see the structure and the commands I'll use. You can see I'm using extern to indicate that other .c files can access variables I'm defining in shell.c.

Source 72: shell.h

```

1  #ifndef _shell_h
2  #define _shell_h
3
4  #define MAX_COMMAND_ARGS 100
5
6  #include <apr_thread_proc.h>
7
8  typedef struct Shell {
9      const char *dir;
10     const char *exe;
11
12     apr_procattr_t *attr;
13     apr_proc_t proc;
14     apr_exit_why_e exit_why;
15     int exit_code;
16
17     const char *args[MAX_COMMAND_ARGS];
18 } Shell;
19
20 int Shell_run(apr_pool_t *p, Shell *cmd);
21 int Shell_exec(Shell cmd, ...);
22
23 extern Shell CLEANUP_SH;
24 extern Shell GIT_SH;
25 extern Shell TAR_SH;
26 extern Shell CURL_SH;
27 extern Shell CONFIGURE_SH;
28 extern Shell MAKE_SH;
29 extern Shell INSTALL_SH;
30

```



```
31 #endif
```

Make sure you've created shell.h exactly, and that you've got the same names and number of extern Shell variables. Those are used by the Shell_run and Shell_exec functions to run commands. I define these two functions, and create the real variables in shell.c.

Source 73: shell.c

```
1  #include "shell.h"
2  #include "dbg.h"
3  #include <stdarg.h>
4
5  int Shell_exec(Shell template, ...)
6  {
7      apr_pool_t *p = NULL;
8      int rc = -1;
9      apr_status_t rv = APR_SUCCESS;
10     va_list argp;
11     const char *key = NULL;
12     const char *arg = NULL;
13     int i = 0;
14
15     rv = apr_pool_create(&p, NULL);
16     check(rv == APR_SUCCESS, "Failed to create pool.");
17
18     va_start(argp, template);
19
20     for(key = va_arg(argp, const char *);
21         key != NULL;
22         key = va_arg(argp, const char *))
23     {
24         arg = va_arg(argp, const char *);
25
26         for(i = 0; template.args[i] != NULL; i++) {
27             if(strcmp(template.args[i], key) == 0) {
28                 template.args[i] = arg;
29                 break; // found it
30             }
31         }
32     }
33
34     rc = Shell_run(p, &template);
35     apr_pool_destroy(p);
36     va_end(argp);
37     return rc;
38 error:
39     if(p) {
40         apr_pool_destroy(p);
41     }
42     return rc;
43 }
44
45 int Shell_run(apr_pool_t *p, Shell *cmd)
46 {
47     apr_procattr_t *attr;
48     apr_status_t rv;
49     apr_proc_t newproc;
50
51     rv = apr_procattr_create(&attr, p);
52     check(rv == APR_SUCCESS, "Failed to create proc attr.");
53
54     rv = apr_procattr_io_set(attr, APR_NO_PIPE, APR_NO_PIPE,
55                             APR_NO_PIPE);
56     check(rv == APR_SUCCESS, "Failed to set IO of command.");
57
58     rv = apr_procattr_dir_set(attr, cmd->dir);
59     check(rv == APR_SUCCESS, "Failed to set root to %s", cmd->dir);
60
61     rv = apr_procattr_cmdtype_set(attr, APR_PROGRAM_PATH);
62     check(rv == APR_SUCCESS, "Failed to set cmd type.");
63
64     rv = apr_proc_create(&newproc, cmd->exe, cmd->args, NULL, attr, p);
65     check(rv == APR_SUCCESS, "Failed to run command.");
66
67     rv = apr_proc_wait(&newproc, &cmd->exit_code, &cmd->exit_why, APR_WAIT);
68     check(rv == APR_CHILD_DONE, "Failed to wait.");
69 }
```

```

70  check(cmd->exit_code == 0, "%s exited badly.", cmd->exe);
71  check(cmd->exit_why == APR_PROC_EXIT, "%s was killed or crashed", cmd->exe);
72
73  return 0;
74
75  error:
76  return -1;
77  }
78
79  Shell CLEANUP_SH = {
80  .exe = "rm",
81  .dir = "/tmp",
82  .args = {"rm", "-rf", "/tmp/pkg-build", "/tmp/pkg-src.tar.gz",
83          "/tmp/pkg-src.tar.bz2", "/tmp/DEPENDS", NULL}
84  };
85
86  Shell GIT_SH = {
87  .dir = "/tmp",
88  .exe = "git",
89  .args = {"git", "clone", "URL", "pkg-build", NULL}
90  };
91
92  Shell TAR_SH = {
93  .dir = "/tmp/pkg-build",
94  .exe = "tar",
95  .args = {"tar", "-xzf", "FILE", "--strip-components", "1", NULL}
96  };
97
98  Shell CURL_SH = {
99  .dir = "/tmp",
100 .exe = "curl",
101 .args = {"curl", "-L", "-o", "TARGET", "URL", NULL}
102 };
103
104 Shell CONFIGURE_SH = {
105 .exe = "./configure",
106 .dir = "/tmp/pkg-build",
107 .args = {"configure", "OPTS", NULL},
108 };
109
110 Shell MAKE_SH = {
111 .exe = "make",
112 .dir = "/tmp/pkg-build",
113 .args = {"make", "OPTS", NULL}
114 };
115
116 Shell INSTALL_SH = {
117 .exe = "sudo",
118 .dir = "/tmp/pkg-build",
119 .args = {"sudo", "make", "TARGET", NULL}
120 };

```

Read the shell.c from the bottom to the top (which is a common C source layout) and you see I've created the actual Shell variables that you indicated were extern in shell.h. They live here, but are available to the rest of the program. This is how you make global variables that live in one .o file but are used everywhere. You shouldn't make many of these, but they are handy for things like this.

Continuing up the file we get to the Shell_run function, which is a "base" function that just runs a command based on what's in a Shell struct. It uses many of the functions defined in apr_thread_proc.h so go look up each one to see how it works. This seems like a lot of work compared to just using the system function call, but this also gives you more control over the other program's execution. For example, in our Shell struct we have a .dir attribute which forces the program to be in a specific directory before running.

Finally, I have the Shell_exec function, which is a "variable arguments" function. You've seen this before, but make sure you grasp the stdarg.h functions and how to write one of these. In the challenge for this section you are going to analyze this function.

Challenge 2: Analyze Shell_exec

Challenge for these files (in addition to a full code review just like you did in Challenge 1) is to fully analyze Shell_exec

and break down exactly how it works. You should be able to understand each line, how the two for-loops work, and how arguments are being replaced.

Once you have it analyzed, add a field to struct Shell that gives the number of variable args that must be replaced. Update all the commands to have the right count of args, and then have an error check that confirms these args have been replaced and error exit.

27.4.3 The Command Functions

Now you get to make the actual commands that do the work. These commands will use functions from APR, db.h and shell.h to do the real work of downloading and building software you want it to build. This is the most complex set of files, so do them carefully. As before, you start by making the commands.h file, then implementing its functions in the commands.c file.

Source 74: commands.h

```

1  #ifndef _commands_h
2  #define _commands_h
3
4  #include <apr_pools.h>
5
6  #define DEPENDS_PATH "/tmp/DEPENDS"
7  #define TAR_GZ_SRC "/tmp/pkg-src.tar.gz"
8  #define TAR_BZ2_SRC "/tmp/pkg-src.tar.bz2"
9  #define BUILD_DIR "/tmp/pkg-build"
10 #define GIT_PAT "*.git"
11 #define DEPEND_PAT "*DEPENDS"
12 #define TAR_GZ_PAT "*.tar.gz"
13 #define TAR_BZ2_PAT "*.tar.bz2"
14 #define CONFIG_SCRIPT "/tmp/pkg-build/configure"
15
16 enum CommandType {
17     COMMAND_NONE, COMMAND_INSTALL, COMMAND_LIST, COMMAND_FETCH,
18     COMMAND_INIT, COMMAND_BUILD
19 };
20
21
22 int Command_fetch(apr_pool_t *p, const char *url, int fetch_only);
23
24 int Command_install(apr_pool_t *p, const char *url, const char *configure_opts,
25     const char *make_opts, const char *install_opts);
26
27 int Command_depends(apr_pool_t *p, const char *path);
28
29 int Command_build(apr_pool_t *p, const char *url, const char *configure_opts,
30     const char *make_opts, const char *install_opts);
31
32 #endif

```

There's not much in commands.h that you haven't seen already. You should see that there's some defines for strings that are used everywhere. The real interesting code is in commands.c.

Source 75: commands.c

```

1  #include <apr_uri.h>
2  #include <apr_fnmatch.h>
3  #include <unistd.h>
4
5  #include "commands.h"
6  #include "dbg.h"
7  #include "bstrlib.h"
8  #include "db.h"
9  #include "shell.h"
10
11
12 int Command_depends(apr_pool_t *p, const char *path)
13 {
14

```

```

FILE *in = NULL;
15 bstring line = NULL;
16
17 in = fopen(path, "r");
18 check(in != NULL, "Failed to open downloaded depends: %s", path);
19
20 for(line = bgets((bNgetc)fgetc, in, '\n'); line != NULL;
21     line = bgets((bNgetc)fgetc, in, '\n'))
22 {
23     btrimws(line);
24     log_info("Processing depends: %s", bdata(line));
25     int rc = Command_install(p, bdata(line), NULL, NULL, NULL);
26     check(rc == 0, "Failed to install: %s", bdata(line));
27     bdestroy(line);
28 }
29
30 fclose(in);
31 return 0;
32
33 error:
34 if(line) bdestroy(line);
35 if(in) fclose(in);
36 return -1;
37 }
38
39 int Command_fetch(apr_pool_t *p, const char *url, int fetch_only)
40 {
41     apr_uri_t info = {.port = 0};
42     int rc = 0;
43     const char *depends_file = NULL;
44     apr_status_t rv = apr_uri_parse(p, url, &info);
45
46     check(rv == APR_SUCCESS, "Failed to parse URL: %s", url);
47
48     if(apr_fnmatch(GIT_PAT, info.path, 0) == APR_SUCCESS) {
49         rc = Shell_exec(GIT_SH, "URL", url, NULL);
50         check(rc == 0, "git failed.");
51     } else if(apr_fnmatch(DEPEND_PAT, info.path, 0) == APR_SUCCESS) {
52         check(!fetch_only, "No point in fetching a DEPENDS file.");
53
54         if(info.scheme) {
55             depends_file = DEPENDS_PATH;
56             rc = Shell_exec(CURL_SH, "URL", url, "TARGET", depends_file, NULL);
57             check(rc == 0, "Curl failed.");
58         } else {
59             depends_file = info.path;
60         }
61
62         // recursively process the devpkg list
63         log_info("Building according to DEPENDS: %s", url);
64         rv = Command_depends(p, depends_file);
65         check(rv == 0, "Failed to process the DEPENDS: %s", url);
66
67         // this indicates that nothing needs to be done
68         return 0;
69     } else if(apr_fnmatch(TAR_GZ_PAT, info.path, 0) == APR_SUCCESS) {
70         if(info.scheme) {
71             rc = Shell_exec(CURL_SH,
72                             "URL", url,
73                             "TARGET", TAR_GZ_SRC, NULL);
74             check(rc == 0, "Failed to curl source: %s", url);
75         }
76     }
77
78     rv = apr_dir_make_recursive(BUILD_DIR,
79                                APR_UREAD | APR_UWRITE | APR_UEXECUTE, p);
80     check(rv == APR_SUCCESS, "Failed to make directory %s", BUILD_DIR);
81
82     rc = Shell_exec(TAR_SH, "FILE", TAR_GZ_SRC, NULL);
83     check(rc == 0, "Failed to untar %s", TAR_GZ_SRC);
84 } else if(apr_fnmatch(TAR_BZ2_PAT, info.path, 0) == APR_SUCCESS) {
85     if(info.scheme) {
86         rc = Shell_exec(CURL_SH, "URL", url, "TARGET", TAR_BZ2_SRC, NULL);
87         check(rc == 0, "Curl failed.");
88     }
89
90     apr_status_t rc = apr_dir_make_recursive(BUILD_DIR,
91                                               APR_UREAD | APR_UWRITE | APR_UEXECUTE, p);
92
93     check(rc == 0, "Failed to make directory %s", BUILD_DIR);
94     rc = Shell_exec(TAR_SH, "FILE", TAR_BZ2_SRC, NULL);
95     check(rc == 0, "Failed to untar %s", TAR_BZ2_SRC);

```

```

96     } else {
97         sentinel("Don't now how to handle %s", url);
98     }
99
100    // indicates that an install needs to actually run
101    return 1;
102 error:
103    return -1;
104 }
105
106 int Command_build(apr_pool_t *p, const char *url, const char *configure_opts,
107                  const char *make_opts, const char *install_opts)
108 {
109     int rc = 0;
110
111     check(access(BUILD_DIR, X_OK | R_OK | W_OK) == 0,
112           "Build directory doesn't exist: %s", BUILD_DIR);
113
114     // actually do an install
115     if(access(CONFIG_SCRIPT, X_OK) == 0) {
116         log_info("Has a configure script, running it.");
117         rc = Shell_exec(CONFIGURE_SH, "OPTS", configure_opts, NULL);
118         check(rc == 0, "Failed to configure.");
119     }
120
121     rc = Shell_exec(MAKE_SH, "OPTS", make_opts, NULL);
122     check(rc == 0, "Failed to build.");
123
124     rc = Shell_exec(INSTALL_SH,
125                   "TARGET", install_opts ? install_opts : "install",
126                   NULL);
127     check(rc == 0, "Failed to install.");
128
129     rc = Shell_exec(CLEANUP_SH, NULL);
130     check(rc == 0, "Failed to cleanup after build.");
131
132     rc = DB_update(url);
133     check(rc == 0, "Failed to add this package to the database.");
134
135     return 0;
136
137 error:
138     return -1;
139 }
140
141 int Command_install(apr_pool_t *p, const char *url, const char *configure_opts,
142                   const char *make_opts, const char *install_opts)
143 {
144     int rc = 0;
145     check(Shell_exec(CLEANUP_SH, NULL) == 0, "Failed to cleanup before building.");
146
147     rc = DB_find(url);
148     check(rc != -1, "Error checking the install database.");
149
150     if(rc == 1) {
151         log_info("Package %s already installed.", url);
152         return 0;
153     }
154
155     rc = Command_fetch(p, url, 0);
156
157     if(rc == 1) {
158         rc = Command_build(p, url, configure_opts, make_opts, install_opts);
159         check(rc == 0, "Failed to build: %s", url);
160     } else if(rc == 0) {
161         // no install needed
162         log_info("Depends successfully installed: %s", url);
163     } else {
164         // had an error
165         sentinel("Install failed: %s", url);
166     }
167
168     Shell_exec(CLEANUP_SH, NULL);
169     return 0;
170
171 error:
172     Shell_exec(CLEANUP_SH, NULL);
173     return -1;
174 }

```

After you have this entered in and compiling, you can analyze it. If you've don the challenges until now, you should see how the shell.c functions are being used to run shells and how the arguments are being replaced. If not then go back and make sure you truly understand how Shell_exec actually works.

Challenge 3: Critique My Design

As before, do a complete review of this code and make sure it's exactly the same. Then go through each function and make sure you know how it works and what it's doing. You also should trace how each function calls the other functions you've written in this file and other files. Finally, confirm that you understand all the functions you're calling from APR here.

Once you have the file correct and analyzed, go back through and assume I'm an idiot. Then, criticize the design I have to see how you can improve it if you can. Don't actually change the code, just create a little notes.txt file and write down your thoughts and what you might change.

27.4.4 The devpkg Main Function

The last and most important file, but probably the simplest, is devpkg.c where the main function lives. There's no .h file for this, since this one includes all the others. Instead this just creates the executable devpkg when combined with the other .o files from our Makefile. Enter in the code for this file, and make sure it's correct.

Source 76: devpkg.c

```

1  #include <stdio.h>
2  #include <apr_general.h>
3  #include <apr_getopt.h>
4  #include <apr_strings.h>
5  #include <apr_lib.h>
6
7  #include "dbg.h"
8  #include "db.h"
9  #include "commands.h"
10
11 int main(int argc, const char const *argv[])
12 {
13     apr_pool_t *p = NULL;
14     apr_pool_initialize();
15     apr_pool_create(&p, NULL);
16
17     apr_getopt_t *opt;
18     apr_status_t rv;
19
20     char ch = '\0';
21     const char *optarg = NULL;
22     const char *config_opts = NULL;
23     const char *install_opts = NULL;
24     const char *make_opts = NULL;
25     const char *url = NULL;
26     enum CommandType request = COMMAND_NONE;
27
28
29     rv = apr_getopt_init(&opt, p, argc, argv);
30
31     while(apr_getopt(opt, "l:Lc:m:i:d:SF:B:", &ch, &optarg) == APR_SUCCESS) {
32         switch (ch) {
33             case 'I':
34                 request = COMMAND_INSTALL;
35                 url = optarg;
36                 break;
37
38             case 'L':
39                 request = COMMAND_LIST;
40                 break;
41
42             case 'c':
43                 config_opts = optarg;
44                 break;
45

```

```

46     case 'm':
47         make_opts = optarg;
48         break;
49
50     case 'i':
51         install_opts = optarg;
52         break;
53
54     case 'S':
55         request = COMMAND_INIT;
56         break;
57
58     case 'F':
59         request = COMMAND_FETCH;
60         url = optarg;
61         break;
62
63     case 'B':
64         request = COMMAND_BUILD;
65         url = optarg;
66         break;
67 }
68 }
69
70 switch(request) {
71     case COMMAND_INSTALL:
72         check(url, "You must at least give a URL.");
73         Command_install(p, url, config_opts, make_opts, install_opts);
74         break;
75
76     case COMMAND_LIST:
77         DB_list();
78         break;
79
80     case COMMAND_FETCH:
81         check(url != NULL, "You must give a URL.");
82         Command_fetch(p, url, 1);
83         log_info("Downloaded to %s and in /tmp/", BUILD_DIR);
84         break;
85
86     case COMMAND_BUILD:
87         check(url, "You must at least give a URL.");
88         Command_build(p, url, config_opts, make_opts, install_opts);
89         break;
90
91     case COMMAND_INIT:
92         rv = DB_init();
93         check(rv == 0, "Failed to make the database.");
94         break;
95
96     default:
97         sentinel("Invalid command given.");
98 }
99
100
101 return 0;
102
103 error:
104 return 1;
105 }

```

Challenge 4: The README And Test Files

The challenge for this file is to understand how the arguments are being processed, what the arguments are, and then create the README file with instructions on how to use it. As you write the README, also write a simple test.sh that runs ./devpkg to check that each command is actually working against real live code. Use the set -e at the top of your script so that it aborts on the first error.

Finally, run the program under valgrind and make sure it's all working before moving on to the mid-term exam.

27.5 The Mid-Term Exam

Your final challenge is the mid-term exam and it involves three things:

1. Compare your code to my code available online and starting with 100%, remove 1% for each line you got wrong.
2. Take your notes.txt on how you would improve the code and functionality of devpkg and implement your improvements.
3. Write an alternative version of devpkg using your other favorite language or the one you think can do this the best. Compare the two, then improve your C version of devpkg based on what you've learned.

To compare your code with mine, do the following:

```
1 cd .. # get one directory above your current one
2 git clone git://gitorious.org/devpkg/devpkg.git devpkgzed
3 diff -r devpkg devpkgzed
```

This will clone my version of devpkg into a directory devpkgzed and then use the tool diff to compare what you've done to what I did. The files you're working with in this book come directly from this project, so if you get different lines then that's an error.

Keep in mind that there's no real pass or fail on this exercise, just a way for you to challenge yourself to be as exact and meticulous as possible.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Part II

Data Structures And Algorithms

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

28 [Exercise 27: Creative And Defensive Programming](#)

28.1 [The Creative Programmer Mindset](#)

28.2 [The Defensive Programmer Mindset](#)

28.3 [The Eight Defensive Programmer Strategies](#)

28.4 [Applying The Eight Strategies](#)

28.5 [Order Is Not Important](#)

28.6 [Extra Credit](#)

29 [Exercise 28: Intermediate Makefiles](#)

29.1 [The Basic Project Structure](#)

29.2 [Makefile](#)

29.3 [What You Should See](#)

29.4 [Extra Credit](#)

30 [Exercise 29: Libraries And Linking](#)

30.1 [What You Should See](#)

30.2 [How To Break It](#)

30.3 [Extra Credit](#)

31 [Exercise 30: Automated Testing](#)

31.1 [Wiring Up The Test Framework](#)

31.2 [Extra Credit](#)

32 [Exercise 31: Debugging Code](#)

32.1 [Debug Printing Vs. GDB Vs. Valgrind](#)

32.2 [A Debugging Strategy](#)

32.3 [Using GDB](#)

32.4 [Process Attaching](#)

32.5 [GDB Tricks](#)

32.6 [Extra Credit](#)

33 [Exercise 32: Double Linked Lists](#)

33.1 [What Are Data Structures](#)

33.2 [Making The Library](#)

33.3 [Double Linked Lists](#)

33.4 [Tests](#)

33.5 [What You Should See](#)

33.6 [How To Improve It](#)

33.7 [Extra Credit](#)

34 [Exercise 33: Linked List Algorithms](#)

34.1 [What You Should See](#)

34.2 [How To Improve It](#)

34.3 [Extra Credit](#)

35 [Exercise 34: Dynamic Array](#)

35.1 [Advantages And Disadvantages](#)

35.2 [How To Improve It](#)

35.3 [Extra Credit](#)

36 [Exercise 35: Sorting And Searching](#)

36.1 [Radix Sort And Binary Search](#)

36.2 [How To Improve It](#)

36.3 [Extra Credit](#)

37 [Exercise 36: Safer Strings](#)

37.1 [Why C Strings Were A Horrible Idea](#)

37.2 [Using bstrlib](#)

37.3 [Learning The Library](#)

38 [Exercise 37: Hashmaps](#)

38.1 [How To Improve It](#)

38.2 [Extra Credit](#)

39 [Exercise 38: Hashmap Algorithms](#)

39.1 [What You Should See](#)

39.2 [How To Break It](#)

39.3 [Extra Credit](#)

40 [Exercise 39: String Algorithms](#)

40.1 [What You Should See](#)

40.2 [Analyzing The Results](#)

40.3 [Extra Credit](#)

41 [Exercise 40: Binary Search Trees](#)

41.1 [How To Improve It](#)

41.2 [Extra Credit](#)

42 [Exercise 41: Using Cachegrind And Callgrind For Performance Tuning](#)

42.1 [Running Callgrind](#)

42.2 [Callgrind Annotating Source](#)

42.3 [Analyzing Memory Access With Cachegrind](#)

42.4 [Judo Tuning](#)

42.5 [Using KCachegrind](#)

42.6 [Extra Credit](#)

43 [Exercise 42: Stacks and Queues](#)

43.1 [What You Should See](#)

43.2 [How To Improve It](#)

43.3 [Extra Credit](#)

44 [Exercise 43: A Simple Statistics Engine](#)

44.1 [Rolling Standard Deviation And Mean](#)

44.2 [Implementation](#)

44.3 [How To Use It](#)

44.4 [Extra Credit](#)

45 [Exercise 44: Ring Buffer](#)

- 45.1 [The Unit Test](#)
- 45.2 [What You Should See](#)
- 45.3 [How To Improve It](#)
- 45.4 [Extra Credit](#)
- 46 [Exercise 45: A Simple TCP/IP Client](#)
- 46.1 [Augment The Makefile](#)
- 46.2 [The netclient Code](#)
- 46.3 [What You Should See](#)
- 46.4 [How To Break It](#)
- 46.5 [Extra Credit](#)
- 47 [Exercise 46: Ternary Search Tree](#)
- 47.1 [What You Should See](#)
- 47.2 [How To Break It](#)
- 47.3 [Extra Credit](#)
- 48 [Exercise 47: Ternary Search Tree Algorithms](#)
- 48.1 [What You Should See](#)
- 48.2 [How To Break It](#)
- 48.3 [Extra Credit](#)
- 49 [Exercise 48: A Tiny Virtual Machine Part 1](#)
- 49.1 [What You Should See](#)
- 49.2 [How To Break It](#)
- 49.3 [Extra Credit](#)
- 50 [Exercise 48: A Tiny Virtual Machine Part 2](#)
- 50.1 [What You Should See](#)
- 50.2 [How To Break It](#)
- 50.3 [Extra Credit](#)
- 51 [Exercise 50: A Tiny Virtual Machine Part 3](#)
- 51.1 [What You Should See](#)
- 51.2 [How To Break It](#)
- 51.3 [Extra Credit](#)
- 52 [Exercise 51: A Tiny Virtual Machine Part 4](#)
- 52.1 [What You Should See](#)
- 52.2 [How To Break It](#)
- 52.3 [Extra Credit](#)
- 53 [Exercise 52: A Tiny Virtual Machine Part 5](#)
- 53.1 [What You Should See](#)
- 53.2 [How To Break It](#)
- 53.3 [Extra Credit](#)
- 54 [Next Steps](#)

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 28

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 27: Creative And Defensive Programming

You have now learned most of the basics of C programming and are ready to start becoming a serious programmer. This is where you go from beginner to expert, both with C and hopefully with core computer science concepts. I will be teaching you a few of the core data structures and algorithms that every programmer should know, and then a few very interesting ones I've used in real software for years.

Before I can do that I have to teach you some basic skills and ideas that will help you make better software. Exercises 27 through 31 will teach you advanced concepts and feature more talking than code, but after those you'll apply what you learn to making a core library of useful data structures.

The first step in getting better at writing C code (and really any language) is to learn a new mindset called "defensive programming". Defensive programming assumes that you are going to make many mistakes and then attempts to prevent them at every possible step. In this exercise I'm going to teach you how to think about programming defensively.

28.1 The Creative Programmer Mindset

It's not possible to tell you how to be creative in a short exercise like this, but I will tell you that creativity involves taking risks and being open minded. Fear will quickly kill creativity, so the mindset I adopt, and many programmers adopt on, accident is designed to make me unafraid of taking chances and looking like an idiot:

1. I can't make a mistake.
2. It doesn't matter what people think.
3. Whatever my brain comes up with is going to be a great idea.

I only adopt this mindset temporarily, and even have little tricks to turn it on. By doing this I can come up with ideas, find creative solutions, open my thoughts to odd connections, and just generally invent weirdness without fear. In this mindset I will typically write a horrible first version of something just to get the idea out.

However, when I've finished my creative prototype I will throw it out and get serious about making it solid. Where other people make a mistake is carrying the creative mindset into their implementation phase. This then leads to a very different destructive mindset that is the dark side of the creative mindset:

1. It is possible to write perfect software.
2. My brain tells me the truth, and it can't find any errors, therefore I have written perfect software.
3. My code is who I am and people who criticize its perfection are criticizing me.

These are lies. You will frequently run into programmers who feel intense pride about what they've created, which is

natural, but this pride gets in the way of their ability to objectively improve their craft. Because of pride and attachment to what they've written, they can continue to believe that what they write is perfect. As long as they ignore other people's criticism of their code they can protect their fragile ego and never improve.

The trick to being creative and making solid software is to also be able to adopt a defensive programming mindset.

28.2 The Defensive Programmer Mindset

After you have a working creative prototype and you're feeling good about the idea, it's time to switch to being a defensive programmer. The defensive programmer basically hates your code and believes these things:

1. Software has errors.
2. You are not your software, yet you are responsible for the errors.
3. You can never remove the errors, only reduce their probability.

This mindset lets you be honest about your work and critically analyze it for improvements. Notice that it doesn't say you are full of errors? It says your code is full of errors. This is a significant thing to understand because it gives you the power of objectivity for the next implementation.

Just like the creative mindset, the defensive programming mindset has a dark side as well. The defensive programmer is a paranoid who is afraid of everything, and this fear prevents them from possibly being wrong or making mistakes. That's great when you are trying to be ruthlessly consistent and correct, but it is murder on creative energy and concentration.

28.3 The Eight Defensive Programmer Strategies

Once you've adopted this mindset, you can then rewrite your prototype and follow a set of eight strategies I use to make my code as solid as I can. While I work on the "real" version I ruthlessly follow these strategies and try to remove as many errors as I can, thinking like someone who wants to break the software.

Never Trust Input

Never trust the data you are given and always validate it.

Prevent Errors

If an error is possible, no matter how probable, try to prevent it.

Fail Early And Openly

Fail early, cleanly, and openly, stating what happened, where and how to fix it.

Document Assumptions

Clearly state the pre-conditions, post-conditions, and invariants.

Prevention Over Documentation

Do not do with documentation, that which can be done with code or avoided completely.

Automate Everything

Automate everything, especially testing.

Simplify And Clarify

Always simplify the code to the smallest, cleanest form that works without sacrificing safety.

Question Authority

Do not blindly follow or reject rules.

These aren't the only ones, but they're the core things I feel programmers have to focus on when trying to make good

solid code. Notice that I don't really say exactly how to do these. I'll go into each of these in more detail, and some of the exercises actually cover them extensively.

28.4 Applying The Eight Strategies

These ideas are all great pop-psychology platitudes, but how do you actually apply them to working code? I'm now going to give you a set of things to always do in this book's code that demonstrate each one with a concrete example. The ideas aren't limited to these examples, and you should use these as a guide to making your own code tougher.

28.4.1 Never Trust Input

Let's look at an example of bad design and "better" design. I won't say good design because this could be done even better. Take a look at two functions that both copy a string and a simple main to test out the better one.

Source 77: ex27_1.c

```

1  #undef NDEBUG
2  #include "dbg.h"
3  #include <stdio.h>
4  #include <assert.h>
5
6  /*
7   * Naive copy that assumes all inputs are always valid
8   * taken from K&R C and cleaned up a bit.
9   */
10 void copy(char to[], char from[])
11 {
12     int i = 0;
13
14     // while loop will not end if from isn't '\0' terminated
15     while((to[i] = from[i]) != '\0') {
16         ++i;
17     }
18 }
19
20 /*
21 * A safer version that checks for many common errors using the
22 * length of each string to control the loops and termination.
23 */
24 int safercopy(int from_len, char *from, int to_len, char *to)
25 {
26     assert(from != NULL && to != NULL && "from and to can't be NULL");
27     int i = 0;
28     int max = from_len > to_len - 1 ? to_len - 1 : from_len;
29
30     // to_len must have at least 1 byte
31     if(from_len < 0 || to_len <= 0) return -1;
32
33     for(i = 0; i < max; i++) {
34         to[i] = from[i];
35     }
36
37     to[to_len - 1] = '\0';
38
39     return i;
40 }
41
42
43 int main(int argc, char *argv[])
44 {
45     // careful to understand why we can get these sizes
46     char from[] = "0123456789";
47     int from_len = sizeof(from);
48
49     // notice that it's 7 chars + \0
50     char to[] = "0123456";
51     int to_len = sizeof(to);
52
53     debug("Copying '%s':%d to '%s':%d", from, from_len, to, to_len);
54
55     int rc = safercopy(from_len, from, to_len, to);

```

```

56  check(rc > 0, "Failed to safercopy.");
57  check(to[to_len - 1] == '\0', "String not terminated.");
58
59  debug("Result is: '%s':%d", to, to_len);
60
61  // now try to break it
62  rc = safercopy(from_len * -1, from, to_len, to);
63  check(rc == -1, "safercopy should fail #1");
64  check(to[to_len - 1] == '\0', "String not terminated.");
65
66  rc = safercopy(from_len, from, 0, to);
67  check(rc == -1, "safercopy should fail #2");
68  check(to[to_len - 1] == '\0', "String not terminated.");
69
70  return 0;
71
72  error:
73  return 1;
74  }

```

The copy function is typical C code and it's the source of a huge number of buffer overflows. It is flawed because it assumes that it will always receive a validly terminated C string (with '\0') and just uses a while-loop to process it. Problem is, ensuring that is incredibly difficult, and if not handled right it causes the while-loop to loop infinitely. A cornerstone of writing solid code is never writing loops that can possibly loop forever.

The safercopy function tries to solve this by requiring the caller to give the lengths of the two strings it must deal with. By doing this it can make certain checks about these strings that the copy function can't. It can check the lengths are right, that the to string has enough space, and it will always terminate. It's impossible for this function to run on forever like the copy function.

This is the idea behind never trusting the inputs you receive. If you assume that your function is going to get a string that's not terminated (which is common) then you design your function to not rely on that to function properly. If you need the arguments to never be NULL then you should check for that too. If the sizes should be within sane levels, then check that. You simply assume that whoever is calling you got it wrong and try to make it difficult for them to give you bad state.

This then extends out to software you write that gets input from the external universe. The famous last words of the programmer are, "Nobody's going to do that." I've seen them say that and then the next day someone does exactly that, crashing or hacking their application. If you say nobody is going to do that, just throw in the code to make sure they simply can't hack your application. You'll be glad you did.

There is a diminishing returns on this, but here's a list of things I try to do with all of my functions I write in C:

1. For each parameter identify what its preconditions are, and whether the precondition should cause a failure or return an error. If you are writing a library, favor errors over failures.
2. Add assert calls at the beginning that checks for each failure precondition using `assert(test && "message");` This little hack does the test, and when it fails the OS will typically print the assert line for you, which then includes that message. Very helpful when you're trying to figure out why that assert is there.
3. For the other preconditions, return the error code or use my check macro to do that and give an error message. I didn't use check in this example since it would confuse the comparison.
4. Document why these preconditions exist so that when a programmer hits the error they can figure out if they are really necessary or not.
5. If you are modifying the inputs, make sure that they are correctly formed when the function exits, or abort if they aren't.
6. Always check the error codes of functions you use. For example, people frequently forget to check the return codes from `fopen` or `fread` which causes them to use the resources they give despite the error. This causes your program to crash or gives an avenue for an attack.

7. You also need to be returning consistent error codes so that you can do this for all of your functions too. Once you get in this habit you will then understand why my check macros work the way they do.

Just doing these simple things will improve your resource handling and prevent quite a few errors.

28.4.2 Prevent Errors

In the previous example you may hear people say, "Well it's not very likely someone will use copy wrong." Despite the mountain of attacks made against this very kind of function they still believe that the probability of this error is very low. Probability is a funny thing because people are incredibly bad at guessing the probability of any event. People are however much better at determining if something is possible. They may say the error in copy is not probably, but they can't deny that it's possible.

The key reason is that for something to be probable, it first has to be possible. Determining the possibility is easy, since we can all imagine something happening. What's not so easy is determining its possibility after that. Is the chance that someone might use copy wrong 20%, 10%, or 1%? Who knows, and to determine that you'd need to gather evidence, look at rates of failure in many software packages, and probably survey real programmers and how they use the function.

This means, if you're going to prevent errors then you need to try to prevent what is possible, but focus your energies on what's most probable first. It may not be feasible to handle all the possible ways your software can be broken, but you have to attempt it. But, at the same time, if you don't constrain your efforts to the most probably events with the least effort then you'll be wasting time on irrelevant attacks.

Here's a process for determining what to prevent in your software:

1. List all the possible errors that can happen, no matter how probable.¹
2. Give each one a probability that's a percentage of operations that can be vulnerable. If you are handling requests from the internet, then it's the percentage of requests that can cause the error. If it's function calls, then it's what percentage of function calls can cause it.
3. Give each one an effort in number of hours or amount of code to prevent it. You could also just give an easy or hard metric. Any metric that prevents you from working on the impossible when there's easier things to fix still on the list.
4. Rank them by effort (lowest to highest), and probability (highest to lowest). This is now your task list.
5. Prevent all the errors you can in this list, aiming for removing the possibility, then reducing the probability if you can't make it impossible.
6. If there are errors you can't fix, then document them so someone else can fix it.

This little process will give you a nice list of things to do, but more importantly keep you from working on useless things when there's other more important things to work on. You can also be more or less formal with this process. If you're doing a full security audit this will be better done with a whole team and a nice spreadsheet. If you're just writing a function then simply reviewing the code and scratching out these into some comments is good enough. What's important is you stop assuming that errors don't happen, and you work on removing them when you can without wasting effort.

28.4.3 Fail Early And Openly

If you encounter an error in C you have two choices:

1. Return an error code.
2. Abort the process.

This is just how it is, so what you need to do is make sure the failures happen quickly, are clearly documented, give an error message, and are easy for the programmer to avoid. This is why the check macros I've given you work the way they do. For every error you find it prints a message, the file and line number where it happened, and force a return code. If you just use my macros you'll end up doing the right thing anyway.

I tend to prefer returning error code to aborting the program. If it's catastrophic then I will, but very few errors are truly catastrophic. A good example of when I'll abort a program is if I'm given an invalid pointer, as I did in safecopy. Instead of having the programmer experience a segmentation fault explosion "somewhere", I catch it right away and abort. However, if it's common to pass in a NULL then I'll probably change that to a check instead so that the caller can adapt and keep running.

In libraries however, I try my hardest to never abort. The software using my library can decide if it should abort, and typically I'll only abort if the library is very badly used.

Finally, a big part of being "open" about errors is not using the same message or error code for more than one possible error. You typically see this with errors on external resources. A library will receive an error on a socket, and then simply report "bad socket". What they should do is return exactly what the error was on the socket so it can be debugged properly and fixed. When designing your error reporting, make sure you give a different error message for the different possible errors.

28.4.4 Document Assumptions

If you're following along and doing this advice then what you'll be doing is building a "contract" of how your functions expect the world to be. You've created preconditions for each argument, you've handled possible errors, and you're failing elegantly. The next step is to complete the contract and add "invariants" and "postconditions".

An invariant is some condition that must be held true in some state while the function runs. This isn't very common in simple functions, but when you're dealing with complex structures it becomes more necessary. A good example of an invariant is that a structure is always initialized properly while it's being used. Another would be that a sorted data structure is always sorted during processing.

A postcondition is a guarantee on the exit value or result of a function running. This can blend together with invariants, but this is something as simple as "function always returns 0 or -1 on error". Usually these are documented, but if your function returns an allocated resource, you can add a postcondition that checks to make sure it's returning something and not NULL. Or, you can use NULL to indicate an error, so in that case your postcondition is now checking the resource is deallocated on any errors.

In C programming invariants and postconditions are usually more documentation than actual code and assertions. The best way to handle them is add assert calls for the ones you can, then document the rest. If you do that then when people hit an error they can see what assumptions you made when writing the function.

28.4.5 Prevention Over Documentation

A common problem when programmers write code is they will document a common bug rather than simply fix it. My favorite is when the Ruby on Rails system simply assumed that all months had 30 days. Calendars are hard, so rather than fix it they threw a tiny little comment somewhere that said this was on purpose, and then they refused to fix it for years. Every time someone would complain they would then bluster and yell, "But it's documented!"

Documentation doesn't matter if you can actually fix the problem, and if the function has a fatal flaw then simply don't include it until you can fix it. In the case of Ruby on Rails, not having date functions would have been better than including purposefully broken ones that nobody could use.

As you go through your defensive programming cleanups, try to fix everything you can. If you find yourself documenting more and more problems you can't fix, then consider redesigning the feature or simply removing it. If you really have to keep this horribly broken feature, then I suggest you write it, document it and find a new job before you are blamed for it.

28.4.6 Automate Everything

You are a programmer, and that means your job is putting other people out of jobs with automation. The pinnacle of this is putting yourself out of a job with your own automation. Obviously you won't completely remove what you do, but if are spending your whole day rerunning manual tests in your terminal, then your job is not programming. You are doing QA, and you should automate yourself out of this QA job you probably don't really want anyway.

The easiest way to do this is to write automated tests, or unit tests. In this book I'm going to get into how to do this easily, and I'll avoid most of the dogma of when you should write tests. I'll focus on how to write them, what to test, and how to be efficient at the testing.

Common things programmers fail to automate but they should:

1. Testing and validation.
2. Build processes.
3. Deployment of software.
4. System administration.²
5. Error reporting.

Try to devote some of your time to automating this and you'll have more time to work on the fun stuff. Or, if this is fun to you, then maybe you should work on software that makes automating these things easier.

28.4.7 Simplify And Clarify

The concept of "simplicity" is a slippery one to many people, especially smart people. They generally confuse "comprehension" with "simplicity". If they understand it well, clearly it's simple. The actual test of simplicity is by comparison with something else that could be simpler. But, you'll see people who write code go running to the most complex obtuse structures possible because they think the simpler version of the same thing is "dirty". A love affair with complexity is a programming sickness.

You can fight this disease by first telling yourself, "Simple and clear is not dirty, no matter what everyone else is doing." If everyone else is writing insane visitor patterns involving 19 classes over 12 interfaces and you can do it with two string operations, then you win. They are wrong, no matter how "elegant" they think their complex monstrosity is.

The simplest test of which function to use is:

1. Make sure both functions have no errors. It doesn't matter how fast or simple a function is if it has errors.
2. If you can't fix one, then pick the other.
3. Do they produce the same result? If not then pick the one that has the result you need.
4. If they produce the same result, then pick the one that either has fewer features, fewer branches, or you just think is simpler.
5. Make sure you're not just picking the one that is most impressive. Simple and dirty beats complex and clean any day.

You'll notice that I mostly give up at the end and tell you to use your judgment. Simplicity is ironically a very complex thing, so using your tastes as a guide is the best way to go. Just make sure you adjust your view of what's "good" as you grow and gain more experience.

28.4.8 Question Authority

The final strategy is the most important because it breaks you out of the defensive programming mindset and lets you transition into the creative mindset. Defensive programming is authoritarian and it can be cruel. The job of this mindset is to make you follow rules because without them you'll miss something or get distracted.

This authoritarian attitude has the disadvantage of disabling independent creative thought. Rules are necessary for getting things done, but being a slave to them will kill your creativity.

This final strategy means you should question the rules you follow periodically and assume that they could be wrong, just like the software you are reviewing. What I will typically do is, after a session of defensive programming, I'll go take a non-programming break and let the rules go. Then I'll be ready to do some creative work or do more defensive coding if need to.

28.5 Order Is Not Important

The final thing I'll say on this philosophy is that I'm not telling you to do this in a strict order of "CREATE! DEFEND! CREATE! DEFEND!" At first you may want to do that, but I will actually do either in varying amounts depend on what I want to do, and I may even meld them together with no defined boundary.

I also don't think one mindset is better than another, or that there are strict separation between them. You need both creativity and strictness to do programming well, so work on both if you want to improve.

28.6 Extra Credit

1. The code in the book up to this point (and for the rest of it) potentially violates these rules. Go back through and apply what you've learned to one exercise to see if you can improve it or find bugs.
2. Find an open source project and give some of the files a similar code review. Submit a patch that fixes a bug if you find it.

¹Within reason of course. No point listing aliens sucking your memories out to steal your passwords.

²I'm really guilty of this one.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 29

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 28: Intermediate Makefiles

In the next three Exercises you'll create a skeleton project directory to use in building your C programs later. This skeleton directory will be used in the rest of the book, and in this exercise I'll cover just the Makefile so you can understand it.

The purpose of this structure is to make it easy to build medium sized programs without having to resort to configure tools. If done right you can get very far with just GNU make and some small shell scripts.

29.1 The Basic Project Structure

The first thing to do is make a c-skeleton directory and then put a set of basic files and directories in it that many projects have. Here's my starter:

Source 78: Initial C Project Skeleton

```

1  $ mkdir c-skeleton
2  $ cd c-skeleton/
3  $ touch LICENSE README.md Makefile
4  $ mkdir bin src tests
5  $ cp dbg.h src/ # this is from Ex20
6  $ ls -l
7  total 8
8  -rw-r--r-- 1 zedshaw staff  0 Mar 31 16:38 LICENSE
9  -rw-r--r-- 1 zedshaw staff 1168 Apr  1 17:00 Makefile
10 -rw-r--r-- 1 zedshaw staff  0 Mar 31 16:38 README.md
11 drwxr-xr-x 2 zedshaw staff  68 Mar 31 16:38 bin
12 drwxr-xr-x 2 zedshaw staff  68 Apr  1 10:07 build
13 drwxr-xr-x 3 zedshaw staff 102 Apr  3 16:28 src
14 drwxr-xr-x 2 zedshaw staff  68 Mar 31 16:38 tests
15 $ ls -l src
16 total 8
17 -rw-r--r-- 1 zedshaw staff 982 Apr  3 16:28 dbg.h
18 $
```

At the end you see me do an `ls -l` so you can see the final results.

Here's what each of these does:

LICENSE

If you release the source of your projects you'll want to include a license. If you don't though, the code is copyright by you and nobody has rights to it by default.

README.md

Basic instructions for using your project go here. It ends in `.md` so that it will be interpreted as markdown.

Makefile

The main build file for the project.

bin/

Where programs that users can run go. This is usually empty and the Makefile will create it if it's not there.

build/

Where libraries and other build artifacts go. Also empty and the Makefile will create it if it's not there.

src/

Where the source code goes, usually .c and .h files.

tests/

Where automated tests go.

src/dbg.h

I copied the dbg.h from Exercise 20 into src/ for later.

I'll now break down each of the components of this skeleton project so you can understand how it works.

29.2 Makefile

The first thing I'll cover is the Makefile because from that you can understand how everything else works. The Makefile in this exercise is much more detailed than ones you've used so far, so I'm going to break it down after you type it in:

Source 79: c-skeleton/Makefile

```
1  CFLAGS=-g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG $(OPTFLAGS)
2  LIBS=-ldl $(OPTLIBS)
3  PREFIX?=/usr/local
4
5  SOURCES=$(wildcard src/**/*.c src/*.c)
6  OBJECTS=$(patsubst %.c,%.o,$(SOURCES))
7
8  TEST_SRC=$(wildcard tests/*_tests.c)
9  TESTS=$(patsubst %.c,%, $(TEST_SRC))
10
11 TARGET=build/libYOUR_LIBRARY.a
12 SO_TARGET=$(patsubst %.a,%.so,$(TARGET))
13
14 # The Target Build
15 all: $(TARGET) $(SO_TARGET) tests
16
17 dev: CFLAGS=-g -Wall -Isrc -Wall -Wextra $(OPTFLAGS)
18 dev: all
19
20 $(TARGET): CFLAGS += -fPIC
21 $(TARGET): build $(OBJECTS)
22     ar rcs $@ $(OBJECTS)
23     ranlib $@
24
25 $(SO_TARGET): $(TARGET) $(OBJECTS)
26     $(CC) -shared -o $@ $(OBJECTS)
27
28 build:
29     @mkdir -p build
30     @mkdir -p bin
31
32 # The Unit Tests
33 .PHONY: tests
34 tests: CFLAGS += $(TARGET)
35 tests: $(TESTS)
36     sh ./tests/runtests.sh
37
38 valgrind:
39     VALGRIND="valgrind --log-file=/tmp/valgrind-%p.log" $(MAKE)
40
41 # The Cleaner
42 clean:
43     rm -rf build $(OBJECTS) $(TESTS)
44     rm -f tests/tests.log
45     find . -name "*.gc*" -exec rm {} \;
46     rm -rf 'find . -name "*.dSYM" -print'
47
```

```

48 # The Install
49 install: all
50     install -d $(DESTDIR)/$(PREFIX)/lib/
51     install $(TARGET) $(DESTDIR)/$(PREFIX)/lib/
52
53 # The Checker
54 BADFUNCS='[^_.>a-zA-Z0-9](str(n?cpy|n?cat|xfrm|n?dup|str|pbrk|tok|_)|stpn?cpy|a?sn?printf|byte_)'
55 check:
56     @echo Files with potentially dangerous functions.
57     @egrep $(BADFUNCS) $(SOURCES) || true

```

Remember that you need to indent the Makefile consistently with tab characters. Your editor should know that and do the right thing, but if it doesn't then get a different text editor. No programmer should use an editor that fails at something so simple.

29.2.1 The Header

This makefile is designed to build a library we'll be working on later and to do so reliably on almost any platform by using special features of GNU make. I'll break down each part in sections, starting with the header.

Makefile:1

These are the usual CFLAGS that you set in all of your projects, but with a few others that may be needed to build libraries. You may need to adjust these for different platforms. Notice the OPTFLAGS variable at the end which lets people augment the build options as needed.

Makefile:2

Options used when linking a library, and allows someone else to augment the linking options using the OPTLIBS variable.

Makefile:3

Setting an optional variable called PREFIX that will only have this value if the person running the Makefile didn't already give a PREFIX setting. That's what the ?= does.

Makefile:5

This fancy line of awesome dynamically creates the SOURCES variable by doing a wildcard search for all *.c files in the src/ directory. You have to give both src/**/*.* and src/*.* so that GNU make will include the files in src and also the ones below it.

Makefile:6

Once you have the list of source files, you can then use the patsubst to take the SOURCES list of *.c files and make a new list of all the object files. You do this by telling patsubst to change all *.c extensions to *.o and then those are assigned to OBJECTS.

Makefile:8

Using the wildcard again to find all the test source files for the unit tests. These are separate from the library's source files.

Makefile:9

Then using the same patsubst trick to dynamically get all the TEST targets. In this case I'm stripping away the .c extension so that a full program will be made with the same name. Previously I had replaced the .c with .o so an object file is created.

Makefile:11

Finally, we say the ultimate target is build/libYOUR_LIBRARY.a, which you will change to be whatever library you are actually trying to build.

This completes the top of the Makefile, but I should explain what I mean by "lets people augment the build". When you run make you can do this:

Source 80: A Changing A Make Build

```
# WARNING! Just a demonstration, won't really work right now.
# this installs the library into /tmp
$ make PREFIX=/tmp install
# this tells it to add pthreads
$ make OPTFLAGS=-pthread
```

If you pass in options that match the same kind of variables you have in your Makefile, then those will show up in your build. You can then use this to change how the Makefile runs. The first one alters the PREFIX so that it installs into /tmp instead. The second one sets OPTFLAGS so that the -pthread option is present.

29.2.2 The Target Build

Continuing with the breakdown of the Makefile I have actually building the object files and targets:

Makefile:14

Remember that the first target is what make will run by default when no target is given. In this case it's called all: and it gives \$(TARGET) tests as the targets to build. Look up at the TARGET variable and you see that's the library, so all: will first build the library. The tests target is then further down in the Makefile and builds the unit tests.

Makefile:16

Another target for making "developer builds" that introduces a technique for changing options for just one target. If I do a "dev build" I want the CFLAGS to include options like -Wextra that are useful for finding bugs. If you place them on the target line as options like this, then give another line that says the original target (in this case all) then it will change the options you set. I use this for setting different flags on different platforms that need it.

Makefile:19

Builds the TARGET library, whatever that is, and also uses the same trick from line 15 of giving a target with just options changes to alter them for this run. In this case I'm adding -fPIC just for the library build using the += syntax to add it on.

Makefile:20

Now the real target where I say first make the build directory, then compile all the OBJECTS.

Makefile:21

Runs the ar command which actually makes the TARGET. The syntax \$@ \$(OBJECTS) is a way of saying, "put the target for this Makefile source here and all the OBJECTS after that". In this case the \$@ maps back to the \$(TARGET) on line 19, which maps to build/libYOUR_LIBRARY.a. It seems like a lot to keep track of this indirection, and it can be, but once you get it working this means you just change TARGET at the top and build a whole new library.

Makefile:22

Finally, to make the library you run ranlib on the TARGET and it's built.

Makefile:24-24

This just makes the build/ or bin/ directories if they don't exist. This is then referenced from line 19 when it gives the build target to make sure the build/ directory is made.

You now have all the stuff you need to build the software, so we'll create a way to build and run unit tests to do automated testing.

29.2.3 The Unit Tests

C is different from other languages because it's easier to create one tiny little program for each thing you're testing. Some testing frameworks try to emulate the module concept other languages have and do dynamic loading, but this doesn't work well in C. It's also unnecessary because you can just make a single program that's run for each test instead.

I'll cover this part of the Makefile, and then later you'll see the contents of the tests/ directory that make it actually work.

Makefile:29

If you have a target that's not "real", but there is a directory or file with that name, then you need to tag the target with .PHONY: so make will ignore the file and always run.

Makefile:30

I use the same trick for modifying the CFLAGS variable to add the TARGET to the build so that each of the test programs will be linked with the TARGET library. In this case it will add build/libYOUR_LIBRARY.a to the linking.

Makefile:31

Then I have the actual tests: target which depends on all the programs listed in the TESTS variable we created in the header. This one line actually says, "Make, use what you know about building programs and the current CFLAGS settings to build each program in TESTS."

Makefile:32

Finally, when all of the TESTS are built, there's a simple shell script I'll create later that knows how to run them all and report their output. This line actually runs it so you can see the test results.

Makefile:34-35

In order to be able to dynamically rerun the tests with Valgrind there's a valgrind: target that sets the right variable and runs itself again. This puts the valgrind logs into /tmp/valgrind-*.log so you can go look and see what might be going on. The tests/runtests.sh then knows to run the test programs under Valgrind when it sees this VALGRIND variable.

For the unit testing to work you'll need to create a little shell script that knows how to run the programs. Go ahead and create this tests/runtests.sh script:

Source 81: tests/runtests.sh

```
1  echo "Running unit tests:"
2
3  for i in tests/*_tests
4  do
5      if test -f $i
6      then
7          if $VALGRIND ./ $i 2>> tests/tests.log
8          then
9              echo $i PASS
10         else
11             echo "ERROR in test $i: here's tests/tests.log"
12             echo "-----"
13             tail tests/tests.log
14             exit 1
15         fi
16     fi
17 done
18
19 echo ""
```

I'll be using this later when I cover how unit tests work.

29.2.4 The Cleaner

I now have fully working unit tests, so next up is making things clean when I need to reset everything.

Makefile:38

The clean: target starts things off whenever we need to clean up the project.

Makefile:39-42

This cleans out most of the junk that various compilers and tools leave behind. It also gets rid of the build/ directory

and uses a trick at the end to cleanly erase the weird *.dSYM directories Apple's XCode leaves behind for debugging purposes.

If you run into junk that you need to clean out, simply augment the list of things being deleted in this target.

29.2.5 The Install

After that I'll need a way to install the project, and for a Makefile that's building a library I just need to put something in the common PREFIX directory, which is usually /usr/local/lib.

Makefile:45

This makes install: depend on the all: target so that when you run make install it will be sure to build everything.

Makefile:46

I then use the program install to create the target lib directory if it doesn't exist. In this case I'm trying to make the install as flexible as possible by using two variables that are conventions for installers. DESTDIR is handed to make by installers that do their builds in secure or odd locations to build packages. PREFIX is used when people want the project to be installed in someplace other than /usr/local.

Makefile:47

After that I'm just using install to actually install the library where it needs to go.

The purpose of the install program is to make sure things have the right permissions set. When you run make install you usually have to do it as the root user, so the typical build process is make && sudo make install.

29.2.6 The Checker

The very last part of this Makefile is a bonus that I include in my C projects to help me dig out any attempts to use the "bad" functions in C. Namely the string functions and other "unprotected buffer" functions.

Makefile:50

Sets a variable which is a big regex looking for bad functions like strcpy.

Makefile:51

The check: target so you can run a check whenever you need.

Makefile:52

Just a way to print a message, but doing @echo tells make to not print the command, just its output.

Makefile:53

Run the egrep command on the source files looking for any bad patterns. The || true at the end is a way to prevent make from thinking that egrep not finding things is a failure.

When you run this it will have the odd effect that you'll get an error when there is nothing bad going on.

29.3 What You Should See

I have two more exercises to go before I'm done building the project skeleton directory, but here's me testing out the features of the Makefile.

Source 82: Checking The Makefile

```
1 $ make clean
2 rm -rf build
3 rm -f tests/tests.log
```

```

4 find . -name "*.gc*" -exec rm {} \;
5 rm -rf 'find . -name "*.dSYM" -print'
6 $ make check
7 Files with potentially dangerous functions.
8 ^Cmake: *** [check] Interrupt: 2
9
10 $ make
11 ar rcs build/libYOUR_LIBRARY.a
12 ar: no archive members specified
13 usage: ar -d [-TLsv] archive file ...
14 ar -m [-TLsv] archive file ...
15 ar -m [-abiTLsv] position archive file ...
16 ar -p [-TLsv] archive [file ...]
17 ar -q [-cTLsv] archive file ...
18 ar -r [-cuTLsv] archive file ...
19 ar -r [-abciuTLsv] position archive file ...
20 ar -t [-TLsv] archive [file ...]
21 ar -x [-ouTLsv] archive [file ...]
22 make: *** [build/libYOUR_LIBRARY.a] Error 1
23 $ make valgrind
24 VALGRIND="valgrind --log-file=/tmp/valgrind-%p.log" make
25 ar rcs build/libYOUR_LIBRARY.a
26 ar: no archive members specified
27 usage: ar -d [-TLsv] archive file ...
28 ar -m [-TLsv] archive file ...
29 ar -m [-abiTLsv] position archive file ...
30 ar -p [-TLsv] archive [file ...]
31 ar -q [-cTLsv] archive file ...
32 ar -r [-cuTLsv] archive file ...
33 ar -r [-abciuTLsv] position archive file ...
34 ar -t [-TLsv] archive [file ...]
35 ar -x [-ouTLsv] archive [file ...]
36 make[1]: *** [build/libYOUR_LIBRARY.a] Error 1
37 make: *** [valgrind] Error 2
38 $

```

When I run the `clean:` target that works, but because I don't have any source files in the `src/` directory none of the other commands really work. I'll finish that up in the next exercises.

29.4 Extra Credit

1. Try to get the Makefile to actually work by putting a source and header file in `src/` and making the library. You shouldn't need a main function in the source file.
2. Research what functions the `check:` target is looking for in the `BADFUNCS` regular expression it's using.
3. If you don't do automated unit testing, then go read about it so you're prepared later.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 30

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 29: Libraries And Linking

A central part of any C program is the ability to link it to libraries that your operating system provides. Linking is how you get additional features for your program that someone else created and packaged on the system. You've been using some standard libraries that are automatically included, but I'm going to explain the different types of libraries and what they do.

First off, libraries are poorly designed in every programming language. I have no idea why, but it seems language designers think of linking as something they just slap on later. They are usually confusing, hard to deal with, can't do versioning right, and end up being linked differently everywhere.

C is no different, but the way linking and libraries are done in C is an artifact of how the Unix operating system and executable formats were designed years ago. Learning how C links things helps you understand how your OS works and how it runs your programs.

To start off there are two basic types of libraries:

static

You've made one of these when you used `ar` and `ranlib` to create the `libYOUR_LIBRARY.a` in the last exercise. This kind of library is nothing more than a container for a set of `.o` object files and their functions, and you can treat it like one big `.o` file when building your programs.

dynamic

These typically end in `.so`, `.dll` or about 1 million other endings on OSX depending on the version and who happened to be working that day. Seriously though, OSX adds `.dylib`, `.bundle`, and `.framework` with not much distinction between the three. These files are built and then placed in a common location. When you run your program the OS dynamically loads these files and links them to your program on the fly.

I tend to like static libraries for small to medium sized projects because they are easier to deal with and work on more operating systems. I also like to put all of the code I can into a static library so that I can then link it to unit tests and to the file programs as needed.

Dynamic libraries are good for larger systems, when space is tight, or if you have a large number of programs that use common functionality. In this case you don't want to statically link all of the code for the common features to every program, so you put it in a dynamic library so that it is loaded only once for all of them.

In the previous exercise I laid out how to make a static library (a `.a` file), and that's what I'll use in the rest of the book. In this exercise I'm going to show you how to make a simple `.so` library, and how to dynamically load it with the Unix `dlopen` system. I'll have you do this manually so that you understand everything that's actually happening, then the Extra Credit will be to use the `c-skeleton` skeleton to create it.

30.0.1 Dynamically Loading A Shared Library

To do this I will create two source files. One will be used to make a libex29.so library, the other will be a program called ex29 that can load this library and run functions from it.

Source 83: libex29.c

```

1  #include <stdio.h>
2  #include <ctype.h>
3  #include "dbg.h"
4
5
6  int print_a_message(const char *msg)
7  {
8      printf("A STRING: %s\n", msg);
9
10     return 0;
11 }
12
13
14 int uppercase(const char *msg)
15 {
16     int i = 0;
17
18     // BUG: \0 termination problems
19     for(i = 0; msg[i] != '\0'; i++) {
20         printf("%c", toupper(msg[i]));
21     }
22     printf("\n");
23     return 0;
24 }
25
26
27 int lowercase(const char *msg)
28 {
29     int i = 0;
30
31     // BUG: \0 termination problems
32     for(i = 0; msg[i] != '\0'; i++) {
33         printf("%c", tolower(msg[i]));
34     }
35     printf("\n");
36     return 0;
37 }
38
39
40 int fail_on_purpose(const char *msg)
41 {
42     return 1;
43 }
44
45

```

There's nothing fancy in there, although there's some bugs I'm leaving in on purpose to see if you've been paying attention. You'll fix those later.

What we want to do is use the functions `dlopen`, `dlsym` and `dlclose` to work with the above functions.

Source 84: ex29.c

```

1  #include <stdio.h>
2  #include "dbg.h"
3  #include <dlfcn.h>
4
5  typedef int (*lib_function)(const char *data);
6
7
8  int main(int argc, char *argv[])
9  {
10     int rc = 0;
11     check(argc == 4, "USAGE: ex29 libex29.so function data");
12
13

```

```

14 char *lib_file = argv[1];
15 char *func_to_run = argv[2];
16 char *data = argv[3];
17
18 void *lib = dlopen(lib_file, RTLD_NOW);
19 check(lib != NULL, "Failed to open the library %s: %s", lib_file, dlerror());
20
21 lib_function func = dlsym(lib, func_to_run);
22 check(func != NULL, "Did not find %s function in the library %s: %s", func_to_run, lib_file, dlerror());
23
24 rc = func(data);
25 check(rc == 0, "Function %s return %d for data: %s", func_to_run, rc, data);
26
27 rc = dlclose(lib);
28 check(rc == 0, "Failed to close %s", lib_file);
29
30 return 0;
31
32 error:
33 return 1;
34 }

```

I'll now break this down so you can see what's going on in this small bit of useful code:

ex29.c:5

I'll use this function pointer definition later to call functions in the library. This is nothing new, but make sure you understand what it's doing.

ex29.c:17

After the usual setup for a small program, I use the dlopen function to load up the library indicated by lib_file. This function returns a handle that we use later and works a lot like opening a file.

ex29.c:18

If there's an error, I do the usual check and exit, but notice at the end that I'm using dlerror to find out what the library related error was.

ex29.c:20

I use dlsym to get a function out of the lib by its string name in func_to_run. This is the powerful part, since I'm dynamically getting a pointer to a function based on a string I got from the command line argv.

ex29.c:23

I then call the func function that was returned, and check its return value.

ex29.c:26

Finally, I close the library up just like I would a file. Usually you keep these open the whole time the program is running, so closing at the end isn't as useful, but I'm demonstrating it here.

30.1 What You Should See

Now that you know what this file does, here's a shell session of me building the libex29.so, ex29 and then working with it. Follow along so you learn how these things are built manually.

Source 85: Building And Using libex29.so

```

1 # compile the lib file and make the .so
2 $ cc -c libex29.c -o libex29.o
3 $ cc -shared -o libex29.so libex29.o
4
5 # make the loader program
6 $ cc -Wall -g -DNDEBUG ex29.c -ldl -o ex29
7
8 # try it out with some things that work
9 $ ex29 ./libex29.so print_a_message "hello there"
10 -bash: ex29: command not found
11 $ ./ex29 ./libex29.so print_a_message "hello there"
12 A STRING: hello there
13 $ ./ex29 ./libex29.so uppercase "hello there"

```



```

14 HELLO THERE
15 $ ./ex29 ./libex29.so lowercase "HELLO tHeRe"
16 hello there
17 $ ./ex29 ./libex29.so fail_on_purpose "i fail"
18 [ERROR] (ex29.c:23: errno: None) Function fail_on_purpose return 1 for data: i fail
19
20 # try to give it bad args
21 $ ./ex29 ./libex29.so fail_on_purpose
22 [ERROR] (ex29.c:11: errno: None) USAGE: ex29 libex29.so function data
23
24 # try calling a function that is not there
25 $ ./ex29 ./libex29.so adfasfasdf asdfadff
26 [ERROR] (ex29.c:20: errno: None) Did not find adfasfasdf
27   function in the library libex29.so: dlsym(0x1076009b0, adfasfasdf): symbol not found
28
29 # try loading a .so that is not there
30 $ ./ex29 ./libex.so adfasfasdf asdfadfas
31 [ERROR] (ex29.c:17: errno: No such file or directory) Failed to open
32   the library libex.so: dlopen(libex.so, 2): image not found
33 $

```

One thing that you may run into is that every OS, every version of every OS, and every compiler on every version of every OS, seems to want to change the way you build a shared library every other month that some new programmer thinks it's wrong. If the line I use to make the libex29.so file is wrong, then let me know and I'll add some comments for other platforms.

Note 11: Irritating .so Ordering

Sometimes you'll do what you think is normal and run this command `cc -Wall -g -DNDEBUG -ldl ex29.c -o ex29` thinking everything will work, but nope. You see, on some platforms the order of where libraries goes makes them work or not, and for no real reason. On Debian or Ubuntu you have to do `cc -Wall -g -DNDEBUG ex29.c -ldl -o ex29` for no reason at all. It's just the way it is, so since this works on OSX I'm doing it here, but in the future, if you link against a dynamic library and it can't find a function, try shuffling things around.

The irritation here is there is an actual platform difference on nothing more than order of command line arguments. On no rational planet should putting an `-ldl` at one position be different from another. It's an option, and having to know these things is incredibly annoying.

30.2 How To Break It

Open libex29.so and edit it with an editor that can handle binary files. Change a couple bytes, then close it. Try to see if you can get the dlopen function to load it even though you've corrupted it.

30.3 Extra Credit

1. Were you paying attention to the bad code I have in the libex29.c functions? See how, even though I use a for-loop they still check for `'\0'` endings? Fix this so the functions always take a length for the string to work with inside the function.
2. Take the c-skeleton skeleton, and create a new project for this exercise. Put the libex29.c file in the src/ directory. Change the Makefile so that it builds this as build/libex29.so.
3. Take the ex29.c file and put it in tests/ex29_tests.c so that it runs as a unit test. Make this all work, which means you have to change it so that it loads the build/libex29.so file and runs tests similar to what I did manually above.
4. Read the man dlopen documentation and read about all the related functions. Try some of the other options to dlopen beside `RTLD_NOW`.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 31

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 30: Automated Testing

Automated testing is used frequently in other languages like Python and Ruby, but rarely used in C. Part of the reason comes from the difficulty of automatically loading and testing pieces of C code. In this chapter we'll create a very small little testing "framework" and get your skeleton directory building an example test case.

The frameworks I'm going to use, and which you'll include in your c-skeleton skeleton is called "minunit" which started with code from a tiny snippet of code by [Jera Design](#). I then evolved it further, to be this:

Source 86: tests/minunit.h

```

1  #undef NDEBUG
2  #ifndef _minunit_h
3  #define _minunit_h
4
5  #include <stdio.h>
6  #include <dbg.h>
7  #include <stdlib.h>
8
9  #define mu_suite_start() char *message = NULL
10
11 #define mu_assert(test, message) if (!(test)) { log_err(message); return message; }
12 #define mu_run_test(test) debug("\n-----%s", "" #test); \
13     message = test(); tests_run++; if (message) return message;
14
15 #define RUN_TESTS(name) int main(int argc, char *argv[]) { \
16     argc = 1; \
17     debug("----- RUNNING: %s", argv[0]); \
18     printf("-----\nRUNNING: %s\n", argv[0]); \
19     char *result = name(); \
20     if (result != 0) { \
21         printf("FAILED: %s\n", result); \
22     } \
23     else { \
24         printf("ALL TESTS PASSED\n"); \
25     } \
26     printf("Tests run: %d\n", tests_run); \
27     exit(result != 0); \
28 }
29
30
31 int tests_run;
32
33 #endif

```

There's mostly nothing left of the original, as now I'm using the dbg.h macros and I've created a large macro at the end for the boilerplate test runner. Even with this tiny amount of code we'll create a fully functioning unit test system you can use in your C code once it's combined with a shell script to run the tests.

31.1 Wiring Up The Test Framework

To continue this exercise, you should have your `src/libex29.c` working and that you completed the Exercise 29 extra credit where you got the `ex29.c` loader program to properly run. In Exercise 29 I had an extra credit to make it work like a unit test, but I'm going to start over and show you how to do that with `minunit.h`.

The first thing to do is create a simple empty unit test name `tests/libex29_tests.c` with this in it:

Source 87: `tests/libex29_tests.c.h`

```

1  #include "minunit.h"
2
3  char *test_dlopen()
4  {
5
6      return NULL;
7  }
8
9  char *test_functions()
10 {
11
12     return NULL;
13 }
14
15 char *test_failures()
16 {
17
18     return NULL;
19 }
20
21 char *test_dlclose()
22 {
23
24     return NULL;
25 }
26
27 char *all_tests() {
28     mu_suite_start();
29
30     mu_run_test(test_dlopen);
31     mu_run_test(test_functions);
32     mu_run_test(test_failures);
33     mu_run_test(test_dlclose);
34
35     return NULL;
36 }
37
38 RUN_TESTS(all_tests);

```

This code is demonstrating the `RUN_TESTS` macro in `tests/minunit.h` and how to use the other test runner macros. I have the actual test functions stubbed out so that you can see how to structure a unit test. I'll break this file down first:

`libex29_tests.c:1`

Include the `minunit.h` framework.

`libex29_tests.c:3-7`

A first test. Tests are structured so they take no arguments and return a `char *` which is `NULL` on success. This is important because the other macros will be used to return an error message to the test runner.

`libex29_tests.c:9-25`

More tests that are the same as the first one.

`libex29_tests.c:27`

The runner function that will control all the other tests. It has the same form as any other test case, but it gets configured with some additional gear.

`libex29_tests.c:28`

Sets up some common stuff for a test with `mu_suite_start`.

`libex29_tests.c:30`

This is how you say what test to run, using the `mu_run_test` macro.

libex29_tests.c:35

After you say what tests to run, you then return NULL just like a normal test function.

libex29_tests.c:38

Finally, you just use the big RUN_TESTS macro to wire up the main method with all the goodies and tell it to run the all_tests starter.

That's all there is to running a test, now you should try getting just this to run within the project skeleton. Here's what it looks like when I do it:

Source 88: First run of libex29_tests

```

1  $ make clean
2  rm -rf build src/libex29.o tests/libex29_tests
3  rm -f tests/tests.log
4  find . -name "*.gc*" -exec rm {} \;
5  rm -rf 'find . -name "*.dSYM" -print'
6  $ make
7  cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG -fPIC
8  -c -o src/libex29.o src/libex29.c
9  src/libex29.c: In function fail_on_purpose:
10 src/libex29.c:42: warning: unused parameter msg
11 ar rcs build/libYOUR_LIBRARY.a src/libex29.o
12 ranlib build/libYOUR_LIBRARY.a
13 cc -shared -o build/libYOUR_LIBRARY.so src/libex29.o
14 cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG build/libYOUR_LIBRARY.a
15 tests/libex29_tests.c -o tests/libex29_tests
16 sh ./tests/runtests.sh
17 Running unit tests:
18 ----
19 RUNNING: ./tests/libex29_tests
20 ALL TESTS PASSED
21 Tests run: 4
22 tests/libex29_tests PASS
23
24 $
```

I first did a make clean and then I ran the build, which remade the template libYOUR_LIBRARY.a and libYOUR_LIBRARY.so files. Remember that you had to do this in the extra credit for Exercise 29, but just in case you didn't figure it out, here's the diff for the Makefile I'm using now:

Source 89: Makefile changes for .so builds

```

diff --git a/code/c-skeleton/Makefile b/code/c-skeleton/Makefile
index 135d538..21b92bf 100644
--- a/code/c-skeleton/Makefile
+++ b/code/c-skeleton/Makefile
@@ -9,9 +9,10 @@ TEST_SRC=$(wildcard tests/*_tests.c)
TESTS=$(patsubst %.c,%, $(TEST_SRC))

TARGET=build/libYOUR_LIBRARY.a
+SO_TARGET=$(patsubst %.a,%.so, $(TARGET))

# The Target Build
-all: $(TARGET) tests
+all: $(TARGET) $(SO_TARGET) tests

dev: CFLAGS=-g -Wall -Isrc -Wall -Wextra $(OPTFLAGS)
dev: all
@@ -21,6 +22,9 @@ $(TARGET): build $(OBJECTS)
ar rcs $@ $(OBJECTS)
ranlib $@

+$(SO_TARGET): $(TARGET) $(OBJECTS)
+$(CC) -shared -o $@ $(OBJECTS)
+
build:
@mkdir -p build
@mkdir -p bin
```

With those changes you should be now building everything and you can finally fill in the remaining unit test functions:

Source 90: Final version of tests/libex29_tests.c

```

1  #include "minunit.h"
2  #include <dlfcn.h>
3
4  typedef int (*lib_function)(const char *data);
5  char *lib_file = "build/libYOUR_LIBRARY.so";
6  void *lib = NULL;
7
8  int check_function(const char *func_to_run, const char *data, int expected)
9  {
10     lib_function func = dlsym(lib, func_to_run);
11     check(func != NULL, "Did not find %s function in the library %s: %s", func_to_run, lib_file, dlerror());
12
13     int rc = func(data);
14     check(rc == expected, "Function %s return %d for data: %s", func_to_run, rc, data);
15
16     return 1;
17 error:
18     return 0;
19 }
20
21 char *test_dlopen()
22 {
23     lib = dlopen(lib_file, RTLD_NOW);
24     mu_assert(lib != NULL, "Failed to open the library to test.");
25
26     return NULL;
27 }
28
29 char *test_functions()
30 {
31     mu_assert(check_function("print_a_message", "Hello", 0), "print_a_message failed.");
32     mu_assert(check_function("uppercase", "Hello", 0), "uppercase failed.");
33     mu_assert(check_function("lowercase", "Hello", 0), "lowercase failed.");
34
35     return NULL;
36 }
37
38 char *test_failures()
39 {
40     mu_assert(check_function("fail_on_purpose", "Hello", 1), "fail_on_purpose should fail.");
41
42     return NULL;
43 }
44
45 char *test_dlclose()
46 {
47     int rc = dlclose(lib);
48     mu_assert(rc == 0, "Failed to close lib.");
49
50     return NULL;
51 }
52
53 char *all_tests() {
54     mu_suite_start();
55
56     mu_run_test(test_dlopen);
57     mu_run_test(test_functions);
58     mu_run_test(test_failures);
59     mu_run_test(test_dlclose);
60
61     return NULL;
62 }
63
64 RUN_TESTS(all_tests);

```

Hopefully by now you can figure out what's going on, since there's nothing new in this except for the check_function function. This is a common pattern where I see that I'll be doing a chunk of code repeatedly, and then simply automate it either by creating a function or a macro for it. In this case I'm going to run functions in the .so I load so I just made a little function to do it.

31.2 Extra Credit

1. This works but it's probably a bit messy. Clean the c-skeleton directory up so that it has all these files, but remove any of the code related to Exercise 29. You should be able to copy this directory over and kickstart new projects without much editing.
2. Study the runtests.sh and go read about bash syntax so you know what it does. Think you could write a C version of this script?

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 32

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 31: Debugging Code

I've already taught you about my awesome debug macros and you've been using them. When I debug code I use the `debug()` macro almost exclusively to analyze what's going on and track down the problem. In this exercise I'm going to teach you the basics of using `gdb` to inspect a simple program that runs and doesn't exit. You'll learn how to use `gdb` to attach to a running process, stop it, and see what's happening. After that I'll give you some little tips and tricks that you can use with `gdb`.

32.1 Debug Printing Vs. GDB Vs. Valgrind

I approach debugging primarily with a "scientific method" style, where I come up with possible causes and then rule them out or prove they cause the defect. The problem many programmers have though is their panic and rush to solve a bug makes them feel like this approach will "slow them down". In their rush to solve they fail to notice that they're really just flailing around and gathering no useful information. I find that logging (debug printing) forces me to solve a bug scientifically and it's also just easier to gather information in more situations.

In addition to that, I also have these reasons for using debug printing as my primary debugging tool:

1. You see an entire tracing of a program's execution with debug printing of variables which lets you track how things are going wrong. With `gdb` you have to place watch and debug statements all over for every thing you want and it's difficult to get a solid trace of the execution.
2. The debug prints can stay in the code, and when you need them you can recompile and they come back. With `gdb` you have to configure the same information uniquely for every defect you have to hunt down.
3. It's easier to turn on debug logging on a server that's not working right and then inspect the logs while it runs to see what's going on. System administrators know how to handle logging, they don't know how to use `gdb`.
4. Printing things is just easier. Debuggers are always obtuse and weird with their own quirky interface and inconsistencies. There's nothing complicated about `debug("Yo, dis right? %d", my_stuff);`.
5. Writing debug prints to find a defect forces you to actually analyze the code and use the scientific method. You can think of a debug usage as, "I hypothesize that the code is broken here." Then when you run it you get your hypothesis tested and if it's not broken then you can move to another part where it could be. This may seem like it takes longer, but it's actually faster because you go through a process of "differential diagnosis" and rule out possible causes until you find the real one.
6. Debug printing works better with unit testing. You can actually just compile the debugs in all the time while you work, and when a unit test explodes just go look at the logs any time. With `gdb` you'd have to rerun the unit test under `gdb` and then trace through it to see what's going on.
7. With `valgrind` you get the equivalent of debug prints for many memory related errors, so you don't need to use something like `gdb` to find those defects anymore.

Despite all these reasons that I rely on debug over gdb, I still use gdb in a few situations and I think you should have any tool that helps you get your work done. Sometimes, you just have to connect to a broken program and poke around. Or, maybe you've got a server that's crashing and you can only get at core files to see why. In these and a few other cases, gdb is the way to go, and it's always good to have as many tools as possible to help solve problems.

I then break down when I use gdb vs. valgrind vs. debug printing like this:

1. Valgrind is used to catch all memory errors. I use gdb if valgrind is having problems or if using valgrind would slow the program down too much.
2. Print with debug to diagnose and fix defects related to logic or usage. This amounts to about 90% of the defects after you start using Valgrind.
3. Use gdb for the remaining "mystery weird stuff" or emergency situations to gather information. If Valgrind isn't turning anything up and I can't even print out the information I need, then I bust out gdb and start poking around. My use of gdb in this case is entirely to gather information. Once I have an idea of what's going on I go back to writing a unit test to cause the defect, and then do print statements to find out why.

32.2 A Debugging Strategy

This process will actually work with any debugging technique you're going to use, whether that's Valgrind, debug printing, or using a debugger. I'm going to describe it in terms of using gdb since it seems people skip this process the most when using debuggers, but use this for every bug until you only need it on the very difficult ones.

1. Start a little text file called notes.txt and use it as a kind of "lab notes" for ideas, bugs, problems, etc.
2. Before you use gdb, write out the bug you're going to fix and what could be causing it.
3. For each cause, write out the files and functions where you think the cause is coming from, or just write that you don't know.
4. Now start gdb and pick the first possible cause with good file:function possibilities and set breakpoints there.
5. Use gdb to then run the program and confirm if that is the cause. The best way is to see if you can use the set command to either fix the program easily or cause the error immediately.
6. If this isn't the cause, then mark in the notes.txt that it wasn't and why. Move on to the next possible cause that's easiest to debug, and keep adding information you gather.

In case you haven't noticed, this is basically the scientific method. You write down a set of hypotheses, then you use debugging to prove or disprove them. This gives you insight into more possible causes and then eventually you find it. This process helps you avoid going over the same possible causes repeatedly even though you've found they aren't possible.

You can also do this with debug printing, the only difference is you actually write out your hypotheses in the source code where you think the problem is instead of the notes.txt. In a way, debug printing forces you to tackle bugs scientifically since you have to write out hypotheses as print statements.

32.3 Using GDB

The program I'll debug in this exercise is just a while-loop that doesn't terminate correctly. I'm putting a small usleep call in it so that there's something interesting to troll through as well.

Source 91: ex31.c

```

1  #include <unistd.h>
2
3  int main(int argc, char *argv[])
4  {
5      int i = 0;
6
7      while(i < 100) {
8          usleep(3000);
9      }
10
11     return 0;
12 }
```

Compile this like normal and then start it under gdb like this: `gdb ./ex31`

Once it's running I want you to play around with these gdb commands to see what they do and how to use them.

help COMMAND

Get a short help with COMMAND.

break file.c:(line—function)

Sets a break point where you want to pause execution. You can give lines or function names to break at after the file.

run ARGS

Runs the program, using the ARGS as arguments to the program.

cont

Continues execution until a new breakpoint or error.

step

Step through the code, but move into functions. Use this to trace into a function and see what it's doing.

next

Just like step, but go over functions by just running them.

backtrace (or bt)

Does a "backtrace", which dumps the trace of function calls leading to the current point in the program. Very useful for figuring out how you got there, since it also prints the parameters that were passed to each function. It's also similar to what Valgrind reports when you have a memory error.

set var X = Y

Set variable X equal to Y.

print X

Prints out the value of X, and you can usually use C syntax to access the values of pointers and contents of structs.

ENTER

The ENTER key just repeats the last command.

quit

Exits gdb

Those are the majority of commands I use with gdb. Your job is to now play with these and ex31 so you can get familiar with the output.

Once you're familiar with gdb you'll want to play with it some more. Try using it on more complicated programs like devpkg to see if you can alter the program's execution or analyze what it's doing.

32.4 Process Attaching

The most useful thing about gdb is the ability to attach to a running program and debug it right there. When you have a crashing server or a GUI program, you can't usually start it under gdb like you just did. Instead, you have to start it, hope

it doesn't crash right away, then attach to it and set a breakpoint. In this part of the exercise I'll show you how to do that.

After you exit gdb I want you to restart ex31 if you stopped it, and then start another Terminal window so you can process attach to it. Process attaching is where you tell gdb to connect to a program that's already running so you can inspect it live. It stops the program and then you can walk through it, and when you're done it'll continue just like normal.

Here's a session of me doing it to ex31, stepping through it, then fixing the while-loop to make it exit.

Source 92: ex31.sh-session

```

1  $ ps ax | grep ex31
2  10026 s000 S+   0:00.11 ./ex31
3  10036 s001 R+   0:00.00 grep ex31
4
5  $ gdb ./ex31 10026
6  GNU gdb 6.3.50-20050815 (Apple version gdb-1705) (Fri Jul  1 10:50:06 UTC 2011)
7  Copyright 2004 Free Software Foundation, Inc.
8  GDB is free software, covered by the GNU General Public License, and you are
9  welcome to change it and/or distribute copies of it under certain conditions.
10 Type "show copying" to see the conditions.
11 There is absolutely no warranty for GDB.  Type "show warranty" for details.
12 This GDB was configured as "x86_64-apple-darwin"...Reading symbols for shared libraries .. done
13
14 /Users/zedshaw/projects/books/learn-c-the-hard-way/code/10026: No such file or directory
15 Attaching to program: '/Users/zedshaw/projects/books/learn-c-the-hard-way/code/ex31', process 10026.
16 Reading symbols for shared libraries + done
17 Reading symbols for shared libraries ++..... done
18 Reading symbols for shared libraries + done
19 0x00007fff862c9e42 in __semwait_signal ()
20
21 (gdb) break 8
22 Breakpoint 1 at 0x107babf14: file ex31.c, line 8.
23
24 (gdb) break ex31.c:11
25 Breakpoint 2 at 0x107babf1c: file ex31.c, line 12.
26
27 (gdb) cont
28 Continuing.
29
30 Breakpoint 1, main (argc=1, argv=0x7fff677aabd8) at ex31.c:8
31 8   while(i < 100) {
32
33 (gdb) p i
34 $1 = 0
35
36 (gdb) cont
37 Continuing.
38
39 Breakpoint 1, main (argc=1, argv=0x7fff677aabd8) at ex31.c:8
40 8   while(i < 100) {
41
42 (gdb) p i
43 $2 = 0
44
45 (gdb) list
46 3
47 4 int main(int argc, char *argv[])
48 5 {
49 6     int i = 0;
50 7
51 8     while(i < 100) {
52 9         usleep(3000);
53 10    }
54 11
55 12    return 0;
56
57 (gdb) set var i = 200
58
59 (gdb) p i
60 $3 = 200
61
62 (gdb) next
63
64 Breakpoint 2, main (argc=1, argv=0x7fff677aabd8) at ex31.c:12
65 12    return 0;
66
67 (gdb) cont

```

```

68 Continuing.
69
70 Program exited normally.
71 (gdb) quit
72 $

```

Note 12: OSX Problems

On OSX you may see a GUI prompt for the root password, and even after you give it you still get an error from gdb saying "Unable to access task for process-id XXX: (os/kern) failure." In that case stop both gdb and the ex31 program, then start over and it should work as long as you successfully entered the root password.

I'll walk through this session and explain what I did:

gdb:1

I use ps to find out what the process id is of the ex31 I want to attach.

gdb:5

I'm attaching using gdb ./ex31 PID replacing PID with the process id I have.

gdb:6-19

gdb prints out a bunch of information about it's license and then all the things it's reading. ¹

gdb:21

The program is attached and stopped at this point, so now I set a breakpoint at line 8 in the file with break. I'm assuming that I'm already in the file I want to break when I do this.

gdb:24

A better way to do a break, is give file.c:line format so you can be sure you did the right location. I do that in this break.

gdb:27

I use cont to continue processing until I hit a breakpoint.

gdb:30-31

The breakpoint is reached so gdb prints out variables I need to know about (argc and argv) and where it's stopped, then the line of code for the breakpoint.

gdb:33-34

I use the abbreviation for print "p" to print out the value of the i variable. It's 0.

gdb:36

Continue again to see if i changes.

gdb:42

Print out i again, and nope it's not changing.

gdb:45-55

Use list to see what the code is, and then I realize it's not exiting because I'm not incrementing i.

gdb:57

Confirm my hypothesis that i needs to change by using the set command to change it to be i = 200. This is one of the best features of gdb as it lets you "fix" a program really quick to see if you're right.

gdb:59

Print out i just to make sure it changed.

gdb:62

Use next to move to the next piece of code, and I see that the breakpoint at ex31.c:12 is hit, so that means the while-loop exited. My hypothesis is correct, I need to make i change.

gdb:67

Use cont to continue and the program exits like normal.

gdb:71

I finally use quit to get out of gdb.

32.5 GDB Tricks

Here's a list of simple tricks you can do with GDB:

`gdb -args`

Normally gdb takes arguments you give it and assumes they are for itself. Using `--args` passes them to the program.

`thread apply all bt`

Dumps a backtrace for all threads. Very useful.

`gdb -batch -ex r -ex bt -ex q -args`

Runs the program so that, if it bombs you get a backtrace.

?

Got one? Leave it in the comments.

32.6 Extra Credit

1. Find a graphical debugger and compare using it to raw gdb. These are useful when the program you're looking at is local, but they are pointless if you have to debug a program on a server.
2. You can enable "core dumps" on your OS, and when a program crashes you'll get a core file. This core file is like a post-mortem of the program so you can load up what happened right at the crash and see what caused it. Change `ex31.c` so that it crashes after a few iterations, then try to get a core dump and analyze it.

¹Just in case you missed it that gdb really was the GNU debugger and just in case you didn't know it was doing all this stuff.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)

3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 33

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 32: Double Linked Lists

The purpose of this book is to teach you how your computer really works, and included in that is how various data structures and algorithms function. Computers by themselves don't do a lot of useful processing. To make them do useful things you need to structure the data and then organize processing on these structures. Other programming languages either include libraries that implement all of these structures, or they have direct syntax for them. C makes you implement all the data structures you need yourself, which makes it the perfect language to learn how they actually work.

My goal in teaching you these data structures and these algorithms is to help you do three things:

1. Understand what is really going on in Python, Ruby, or JavaScript code like: `data = {"name": "Zed"}`
2. Get even better at C code by applying what you know to a set of solved problems using the data structures.
3. Learn a core set of data structures and algorithms so that you are better informed about what ones work best in certain situations.

33.1 What Are Data Structures

The name "data structure" is self-explanatory. It is an organization of data that fits a certain model. Maybe the model is designed to allow processing the data in a new way. Maybe it's just organized to store it on disk efficiently. In this book I'll follow a simple pattern for making data structures that works reliably:

1. Define a struct for the main "outer structure".
2. Define a struct for the contents, usually nodes with links between them.
3. Create functions that operate on these two.

There's other styles of data structures in C, but this pattern works well and is consistent for most data structures you'll make.

33.2 Making The Library

For the rest of this book you'll be creating a library that you can use when you're done with this book. This library will have the following elements:

1. Header (.h) files for each data structure.
2. Implementation (.c) files for the algorithms.
3. Unit tests that test all of them to make sure they keep working.
4. Documentation we'll autogenerate from the header files.

You already have the c-skeleton so use it to create a liblcthw project:

Source 93: ex32.sh-session

```

1  $ cp -r c-skeleton liblcthw
2  $ cd liblcthw/
3  $ ls
4  LICENSE Makefile README.md bin build src tests
5  $ vim Makefile
6  $ ls src/
7  dbg.h libex29.c libex29.o
8  $ mkdir src/lcthw
9  $ mv src/dbg.h src/lcthw
10 $ vim tests/minunit.h
11 $ rm src/libex29.* tests/libex29*
12 $ make clean
13 rm -rf build tests/libex29_tests
14 rm -f tests/tests.log
15 find . -name "*.gc*" -exec rm {} \;
16 rm -rf 'find . -name "*.dSYM" -print'
17 $ ls tests/
18 minunit.h runtests.sh
19 $

```

In this session I'm doing the following:

1. Copy the c-skeleton over.
2. Edit the Makefile to change libYOUR_LIBRARY.a to liblcthw.a as the new TARGET.
3. Make the src/lcthw directory where we'll put our code.
4. Move the src/dbg.h into this new directory.
5. Edit tests/minunit.h so that it uses #include <lcthw/dbg.h> as the include.
6. Get rid of the source and test files we don't need for libex29.*.
7. Clean up everything that's left over.

With that you're ready to start building the library, and the first data structure I'll build is the Double Linked List.

33.3 Double Linked Lists

The first data strucutre we'll add to liblcthw is a double linked list. This is the simplest data structure you can make, and it has useful properties for certain operations. A linked list works by nodes having pointers to their next or previous element. A "double linked list" contains pointers to both, while a "single linked list" only points at the next element.

Because each node has pointers to the next and previous, and because you keep track of the first and last element of the list, you can do some operations very quickly. Anything that involves inserting or deleting an element will be very fast. They are also easy to implement by most people.

The main disadvantage of a linked list is that traversing it involves processing every single pointer along the way. This means that searching, most sorting, or iterating over the elements will be slow. It also means that you can't really jump to random parts of the list. If you had an array of elements you could just index right into the middle of the list, but a linked list uses a stream of pointers. That means if you want the 10th element, you have to go through elements 1-9.

33.3.1 Definition

As I said in the introduction to this exercise, the process to follow is to first write a header file with the right C struct statements in it.

Source 94: src/lcthw/list.h

```

1  #ifndef lcthw_List_h
2  #define lcthw_List_h
3
4  #include <stdlib.h>
5
6  struct ListNode;
7
8  typedef struct ListNode {
9      struct ListNode *next;
10     struct ListNode *prev;
11     void *value;
12 } ListNode;
13
14 typedef struct List {
15     int count;
16     ListNode *first;
17     ListNode *last;
18 } List;
19
20 List *List_create();
21 void List_destroy(List *list);
22 void List_clear(List *list);
23 void List_clear_destroy(List *list);
24
25 #define List_count(A) ((A)->count)
26 #define List_first(A) ((A)->first != NULL ? (A)->first->value : NULL)
27 #define List_last(A) ((A)->last != NULL ? (A)->last->value : NULL)
28
29 void List_push(List *list, void *value);
30 void *List_pop(List *list);
31
32 void List_shift(List *list, void *value);
33 void *List_unshift(List *list);
34
35 void *List_remove(List *list, ListNode *node);
36
37 #define LIST_FOREACH(L, S, M, V) ListNode *_node = NULL; \
38     ListNode *V = NULL; \
39     for(V = _node = L->S; _node != NULL; V = _node = _node->M)
40
41 #endif

```

The first thing I do is create two structs for the `ListNode` and the `List` that will contain those nodes. This creates the data structure I'll use in the functions and macros I define after that. If you read through these functions they seem rather simple. I'll be explaining them when I cover the implementation, but hopefully you can guess what they do.

How the data structure works is each `ListNode` has three components:

1. A value, which is a pointer to anything and stores the thing we want to put in the list.
2. A `ListNode *next` pointer which points at another `ListNode` that holds the next element in the list.
3. A `ListNode *prev` that holds the previous element. Complex right? Calling the previous thing "previous". I could have used "anterior" and "posterior" but only a jerk would do that.

The `List` struct is then nothing more than a container for these `ListNode` structs that have been linked together in a chain. It keeps track of the count, first and last element of the list.

Finally, take a look at `src/lcthw/list.h:37` where I define the `LIST_FOREACH` macro. This is a common idiom where you make a macro that generates iteration code so people can't mess it up. Getting this kind of processing right can be difficult with data structures, so writing macros helps people out. You'll see how I use this when I talk about the implementation.

33.3.2 Implementation

Once you understand that, you mostly understand how a double linked list works. It is nothing more than nodes with two pointers to the next and previous element of the list. You can then write the `src/lcthw/list.c` code to see how each

operation is implemented.

Source 95: src/lcthw/list.c

```

1  #include <lcthw/list.h>
2  #include <lcthw/dbg.h>
3
4  List *List_create()
5  {
6      return calloc(1, sizeof(List));
7  }
8
9  void List_destroy(List *list)
10 {
11     LIST_FOREACH(list, first, next, cur) {
12         if(cur->prev) {
13             free(cur->prev);
14         }
15     }
16
17     free(list->last);
18     free(list);
19 }
20
21
22 void List_clear(List *list)
23 {
24     LIST_FOREACH(list, first, next, cur) {
25         free(cur->value);
26     }
27 }
28
29
30 void List_clear_destroy(List *list)
31 {
32     List_clear(list);
33     List_destroy(list);
34 }
35
36
37 void List_push(List *list, void *value)
38 {
39     ListNode *node = calloc(1, sizeof(ListNode));
40     check_mem(node);
41
42     node->value = value;
43
44     if(list->last == NULL) {
45         list->first = node;
46         list->last = node;
47     } else {
48         list->last->next = node;
49         node->prev = list->last;
50         list->last = node;
51     }
52
53     list->count++;
54
55 error:
56     return;
57 }
58
59 void *List_pop(List *list)
60 {
61     ListNode *node = list->last;
62     return node != NULL ? List_remove(list, node) : NULL;
63 }
64
65 void List_shift(List *list, void *value)
66 {
67     ListNode *node = calloc(1, sizeof(ListNode));
68     check_mem(node);
69
70     node->value = value;
71
72     if(list->first == NULL) {
73         list->first = node;
74         list->last = node;
75     } else {
76         node->next = list->first;

```

```

77     list->first->prev = node;
78     list->first = node;
79 }
80
81     list->count++;
82
83 error:
84     return;
85 }
86
87 void *List_unshift(List *list)
88 {
89     ListNode *node = list->first;
90     return node != NULL ? List_remove(list, node) : NULL;
91 }
92
93 void *List_remove(List *list, ListNode *node)
94 {
95     void *result = NULL;
96
97     check(list->first && list->last, "List is empty.");
98     check(node, "node can't be NULL");
99
100    if(node == list->first && node == list->last) {
101        list->first = NULL;
102        list->last = NULL;
103    } else if(node == list->first) {
104        list->first = node->next;
105    } else if (node == list->last) {
106        list->last = node->prev;
107    } else {
108        ListNode *after = node->next;
109        ListNode *before = node->prev;
110        after->prev = before;
111        before->next = after;
112    }
113
114    list->count--;
115    result = node->value;
116    free(node);
117
118 error:
119    return result;
120 }

```

I then implement all of the operations on a double linked list that can't be done with simple macros. Rather than cover every tiny little line of this file, I'm going to give high-level overview of every operation in both the list.h and list.c file, then leave you to read the code.

list.h:List_count

Returns the number of elements in the list, which is maintained as elements are added and removed.

list.h:List_first

Returns the first element of the list, but does not remove it.

list.h:List_last

Returns the last element of the list, but does not remove it.

list.h:LIST_FOREACH

Iterates over the elements in the list.

list.c:List_create

Simply creates the main List struct.

list.c:List_destroy

Destroys a List and any elements it might have.

list.c:List_clear

Convenience function for freeing the values in each node, not the nodes.

list.c:List_clear_destroy

Clears and destroys a list. It's not very efficient since it loops through them twice.

list.c:List_push

The first operation that demonstrates the advantage of a linked list. It adds a new element to the end of the list, and because that's just a couple of pointer assignments, does it very fast.

list.c:List_pop

The inverse of List_push, this takes the last element off and returns it.

list.c:List_shift

The other thing you can easily do to a linked list is add elements to the front of the list very fast. In this case I call that List_shift for lack of a better term.

list.c:List_unshift

Just like List_pop, this removes the first element and returns it.

list.c:List_remove

This is actually doing all of the removal when you do List_pop or List_unshift. Something that seems to always be difficult in data structures is removing things, and this function is no different. It has to handle quite a few conditions depending on if the element being removed is at the front; the end; both front and end; or middle.

Most of these functions are nothing special, and you should be able to easily digest this and understand it from just the code. You should definitely focus on how the LIST_FOREACH macro is used in List_destroy so you can understand how much it simplifies this common operation.

33.4 Tests

After you have those compiling it's time to create the test that makes sure they operate correctly.

Source 96: tests/list_tests.c

```

1  #include "minunit.h"
2  #include <lcshw/list.h>
3  #include <assert.h>
4
5  static List *list = NULL;
6  char *test1 = "test1 data";
7  char *test2 = "test2 data";
8  char *test3 = "test3 data";
9
10
11 char *test_create()
12 {
13     list = List_create();
14     mu_assert(list != NULL, "Failed to create list.");
15
16     return NULL;
17 }
18
19
20 char *test_destroy()
21 {
22     List_clear_destroy(list);
23
24     return NULL;
25 }
26
27
28
29 char *test_push_pop()
30 {
31     List_push(list, test1);
32     mu_assert(List_last(list) == test1, "Wrong last value.");
33
34     List_push(list, test2);
35     mu_assert(List_last(list) == test2, "Wrong last value.");
36
37     List_push(list, test3);
38     mu_assert(List_last(list) == test3, "Wrong last value.");
39     mu_assert(List_count(list) == 3, "Wrong count on push.");
40
41     char *val = List_pop(list);
42     mu_assert(val == test3, "Wrong value on pop.");

```

```

43
44     val = List_pop(list);
45     mu_assert(val == test2, "Wrong value on pop.");
46
47     val = List_pop(list);
48     mu_assert(val == test1, "Wrong value on pop.");
49     mu_assert(List_count(list) == 0, "Wrong count after pop.");
50
51     return NULL;
52 }
53
54 char *test_shift()
55 {
56     List_shift(list, test1);
57     mu_assert(List_first(list) == test1, "Wrong first value.");
58
59     List_shift(list, test2);
60     mu_assert(List_first(list) == test2, "Wrong first value");
61
62     List_shift(list, test3);
63     mu_assert(List_first(list) == test3, "Wrong last value.");
64     mu_assert(List_count(list) == 3, "Wrong count on shift.");
65
66     return NULL;
67 }
68
69 char *test_remove()
70 {
71     // we only need to test the middle remove case since push/shift
72     // already tests the other cases
73
74     char *val = List_remove(list, list->first->next);
75     mu_assert(val == test2, "Wrong removed element.");
76     mu_assert(List_count(list) == 2, "Wrong count after remove.");
77     mu_assert(List_first(list) == test3, "Wrong first after remove.");
78     mu_assert(List_last(list) == test1, "Wrong last after remove.");
79
80     return NULL;
81 }
82
83
84 char *test_unshift()
85 {
86     char *val = List_unshift(list);
87     mu_assert(val == test3, "Wrong value on unshift.");
88
89     val = List_unshift(list);
90     mu_assert(val == test1, "Wrong value on unshift.");
91     mu_assert(List_count(list) == 0, "Wrong count after unshift.");
92
93     return NULL;
94 }
95
96
97
98 char *all_tests() {
99     mu_suite_start();
100
101     mu_run_test(test_create);
102     mu_run_test(test_push_pop);
103     mu_run_test(test_shift);
104     mu_run_test(test_remove);
105     mu_run_test(test_unshift);
106     mu_run_test(test_destroy);
107
108     return NULL;
109 }
110
111 RUN_TESTS(all_tests);

```

This test simply goes through every operation and makes sure it works. I use a simplification in the test where I create just one List *list for the whole program, then have the tests work on it. This saves the trouble of building a List for every test, but it could mean that some tests only pass because of how the previous test ran. In this case I try to make each test keep the list clear or actually use the previous test's results.

33.5 What You Should See

If you did everything right, then when you do a build and run the unit tests it should look like this:

Source 97: Ex32 Session

```

1  $ make
2  cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG -fPIC -c -o src/lcthw/list.o src/lcthw/list.c
3  ar rcs build/liblcthw.a src/lcthw/list.o
4  ranlib build/liblcthw.a
5  cc -shared -o build/liblcthw.so src/lcthw/list.o
6  cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG build/liblcthw.a tests/list_tests.c -o tests/list_tests
7  sh ./tests/runtests.sh
8  Running unit tests:
9  ----
10 RUNNING: ./tests/list_tests
11 ALL TESTS PASSED
12 Tests run: 6
13 tests/list_tests PASS
14 $

```

Make sure 6 tests ran, that it builds without warnings or errors, and that it's making the build/liblcthw.a and build/liblcthw.so files.

33.6 How To Improve It

Instead of breaking this, I'm going to tell you how to improve the code:

1. You can make List_clear_destroy more efficient by using LIST_FOREACH and doing both free calls inside one loop.
2. You can add asserts for preconditions that it isn't given a NULL value for the List *list parameters.
3. You can add invariants that check the list's contents are always correct, such as count is never < 0, and if count > 0 then first isn't NULL.
4. You can add documentation to the header file in the form of comments before each struct, function, and macro that describes what it does.

These amount to going through the defensive programming practices I talked about and "hardening" this code against flaws or improving usability. Go ahead and do these things, then find as many other ways to improve the code.

33.7 Extra Credit

1. Research double vs. single linked lists and when one is preferred over the other.
2. Research the limitations of a double linked list. For example, while they are efficient for inserting and deleting elements, they are very slow for iterating over them all.
3. What operations are missing that you can imagine needing? Some examples are copying, joining, splitting. Implement these operations and write the unit tests for them.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 34

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 33: Linked List Algorithms

I'm going to cover two algorithms you can do on a linked list that involve sorting. I'm going to warn you first that if you need to sort the data, then don't use a linked list. These are horrible for sorting things, and there's much better data structures you can use if that's a requirement. I'm covering these two algorithms because they are slightly difficult to pull off with a linked list and get you thinking about manipulating them efficiently.

In the interest of writing this book, I'm going to put the algorithms in two different files `list_algos.h` and `list_algos.c` then write a test in `list_algos_test.c`. For now just follow my structure, as it does keep things clean, but if you ever work on other libraries remember this isn't a common structure.

In this exercise I'm going to also give you an extra challenge and I want you to try not to cheat. I'm going to give you the unit test first, and I want you to type it in. Then I want you to try and implement the two algorithms based on their descriptions in Wikipedia before seeing if your code is like my code.

34.0.1 Bubble And Merge Sort

You know what's awesome about the Internet? I can just link you to the [Bubble Sort page](#) and [Merge Sort page](#) on Wikipedia and tell you to read that. Man, that saves me a boat load of typing. Now I can tell you how to actually implement each of these using the pseudo-code they have there. Here's how you can tackle an algorithm like this:

1. Read the description and look at any visualizations it has.
2. Either draw the algorithm on paper using boxes and lines, or actually take a deck of numbered cards (like Poker Cards) and try to do the algorithm manually. This gives you a concrete demonstration of how the algorithm works.
3. Create the skeleton functions in your `list_algos.c` file and make a working `list_algos.h` file, then setup your test harness.
4. Write your first failing test and get everything to compile.
5. Go back to the Wikipedia page and copy-paste the pseudo-code (not the C code!) into the guts of the first function you're making.
6. Translate this pseudo-code into good C code like I've taught you, using your unit test to make sure it's working.
7. Fill out some more tests for edge cases like, empty lists, already sorted lists, etc.
8. Repeat for the next algorithm and test.

I just gave you the secret to figuring out most of the algorithms out there, that is until you get to some of the more insane ones. In this case you're just doing the Bubble and Merge Sorts from Wikipedia, but those will be good starters.

34.0.2 The Unit Test

Here is the unit test you should try to get passing:

Source 98: tests/list_algos_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/list_algos.h>
3  #include <assert.h>
4  #include <string.h>
5
6  char *values[] = {"XXXX", "1234", "abcd", "xjvef", "NDSS"};
7  #define NUM_VALUES 5
8
9  List *create_words()
10 {
11     int i = 0;
12     List *words = List_create();
13
14     for(i = 0; i < NUM_VALUES; i++) {
15         List_push(words, values[i]);
16     }
17
18     return words;
19 }
20
21 int is_sorted(List *words)
22 {
23     LIST_FOREACH(words, first, next, cur) {
24         if(cur->next && strcmp(cur->value, cur->next->value) > 0) {
25             debug("%s %s", (char *)cur->value, (char *)cur->next->value);
26             return 0;
27         }
28     }
29
30     return 1;
31 }
32
33 char *test_bubble_sort()
34 {
35     List *words = create_words();
36
37     // should work on a list that needs sorting
38     int rc = List_bubble_sort(words, (List_compare)strcmp);
39     mu_assert(rc == 0, "Bubble sort failed.");
40     mu_assert(is_sorted(words), "Words are not sorted after bubble sort.");
41
42     // should work on an already sorted list
43     rc = List_bubble_sort(words, (List_compare)strcmp);
44     mu_assert(rc == 0, "Bubble sort of already sorted failed.");
45     mu_assert(is_sorted(words), "Words should be sort if already bubble sorted.");
46
47     List_destroy(words);
48
49     // should work on an empty list
50     words = List_create(words);
51     rc = List_bubble_sort(words, (List_compare)strcmp);
52     mu_assert(rc == 0, "Bubble sort failed on empty list.");
53     mu_assert(is_sorted(words), "Words should be sorted if empty.");
54
55     List_destroy(words);
56
57     return NULL;
58 }
59
60 char *test_merge_sort()
61 {
62     List *words = create_words();
63
64     // should work on a list that needs sorting
65     List *res = List_merge_sort(words, (List_compare)strcmp);
66     mu_assert(is_sorted(res), "Words are not sorted after merge sort.");
67
68     List *res2 = List_merge_sort(res, (List_compare)strcmp);
69     mu_assert(is_sorted(res), "Should still be sorted after merge sort.");
70     List_destroy(res2);
71     List_destroy(res);
72
73     List_destroy(words);
74     return NULL;
75 }
76

```

```

77
78 char *all_tests()
79 {
80     mu_suite_start();
81
82     mu_run_test(test_bubble_sort);
83     mu_run_test(test_merge_sort);
84
85     return NULL;
86 }
87
88 RUN_TESTS(all_tests);

```

I suggest that you start with the bubble sort and get that working, then move on to the merge sort. What I would do is lay out the function prototypes and skeletons that get all three files compiling, but not passing the test. Then just fill in the implementation until it starts working.

34.0.3 The Implementation

Are you cheating? In future exercises I will do exercises where I just give you a unit test and tell you to implement it, so it'll be good practice for you to not look at this code until you get your own working. Here's the code for the `list_algos.c` and `list_algos.h`:

Source 99: `src/lcthw/list_algos.h`

```

1  #ifndef lcthw_List_algos_h
2  #define lcthw_List_algos_h
3
4  #include <lcthw/list.h>
5
6  typedef int (*List_compare)(const void *a, const void *b);
7
8  int List_bubble_sort(List *list, List_compare cmp);
9
10 List *List_merge_sort(List *list, List_compare cmp);
11
12 #endif

```

Source 100: `src/lcthw/list_algos.c`

```

1  #include <lcthw/list_algos.h>
2  #include <lcthw/dbg.h>
3
4  inline void ListNode_swap(ListNode *a, ListNode *b)
5  {
6      void *temp = a->value;
7      a->value = b->value;
8      b->value = temp;
9  }
10
11 int List_bubble_sort(List *list, List_compare cmp)
12 {
13     int sorted = 1;
14
15     if(List_count(list) <= 1) {
16         return 0; // already sorted
17     }
18
19     do {
20         sorted = 1;
21         LIST_FOREACH(list, first, next, cur) {
22             if(cur->next) {
23                 if(cmp(cur->value, cur->next->value) > 0) {
24                     ListNode_swap(cur, cur->next);
25                     sorted = 0;
26                 }
27             }
28         }
29     } while(!sorted);

```

```

30
31     return 0;
32 }
33
34 inline List *List_merge(List *left, List *right, List_compare cmp)
35 {
36     List *result = List_create();
37     void *val = NULL;
38
39     while(List_count(left) > 0 || List_count(right) > 0) {
40         if(List_count(left) > 0 && List_count(right) > 0) {
41             if(cmp(List_first(left), List_first(right)) <= 0) {
42                 val = List_unshift(left);
43             } else {
44                 val = List_unshift(right);
45             }
46
47             List_push(result, val);
48         } else if(List_count(left) > 0) {
49             val = List_unshift(left);
50             List_push(result, val);
51         } else if(List_count(right) > 0) {
52             val = List_unshift(right);
53             List_push(result, val);
54         }
55     }
56
57     return result;
58 }
59
60 List *List_merge_sort(List *list, List_compare cmp)
61 {
62     if(List_count(list) <= 1) {
63         return list;
64     }
65
66     List *left = List_create();
67     List *right = List_create();
68     int middle = List_count(list) / 2;
69
70     LIST_FOREACH(list, first, next, cur) {
71         if(middle > 0) {
72             List_push(left, cur->value);
73         } else {
74             List_push(right, cur->value);
75         }
76
77         middle--;
78     }
79
80     List *sort_left = List_merge_sort(left, cmp);
81     List *sort_right = List_merge_sort(right, cmp);
82
83     if(sort_left != left) List_destroy(left);
84     if(sort_right != right) List_destroy(right);
85
86     return List_merge(sort_left, sort_right, cmp);
87 }

```

The bubble sort isn't too bad to figure out, although it is really slow. The merge sort is much more complicated, and honestly I could probably spend a bit more time optimizing this code if I wanted to sacrifice clarity.

There is another way to implement merge sort using a "bottom up" method, but it's a little harder to understand so I didn't do it. As I've already said, sorting algorithms on linked lists are entirely pointless. You could spend all day trying to make this faster and it will still be slower than almost any other sortable data structure. The nature of linked lists is such that you simply don't use them if you need to sort things.

34.1 What You Should See

If everything works then you should get something like this:

```

1  $ make clean all
2  rm -rf build src/lcthw/list.o src/lcthw/list_algos.o tests/list_algos_tests tests/list_tests
3  rm -f tests/tests.log
4  find . -name "*.gc*" -exec rm {} \;
5  rm -rf 'find . -name "*.dSYM" -print'
6  cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG -fPIC -c -o src/lcthw/list.o src/lcthw/list.c
7  cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG -fPIC -c -o src/lcthw/list_algos.o src/lcthw/list_algos.c
8  ar rcs build/liblcthw.a src/lcthw/list.o src/lcthw/list_algos.o
9  ranlib build/liblcthw.a
10 cc -shared -o build/liblcthw.so src/lcthw/list.o src/lcthw/list_algos.o
11 cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG build/liblcthw.a tests/list_algos_tests.c -o tests/list_algos_tests
12 cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG build/liblcthw.a tests/list_tests.c -o tests/list_tests
13 sh ./tests/runtests.sh
14 Running unit tests:
15 ----
16 RUNNING: ./tests/list_algos_tests
17 ALL TESTS PASSED
18 Tests run: 2
19 tests/list_algos_tests PASS
20 ----
21 RUNNING: ./tests/list_tests
22 ALL TESTS PASSED
23 Tests run: 6
24 tests/list_tests PASS
25 $

```

After this exercise I'm not going to show you this output unless it's necessary to show you how it works. From now on you should know that I ran the tests and they all passed and everything compiled.

34.2 How To Improve It

Going back to the description of the algorithms, there's several ways to improve these implementations, and there's a few obvious ones:

1. The merge sort does a crazy amount of copying and creating lists, find ways to reduce this.
2. The bubble sort Wikipedia description mentions a few optimizations, implement them.
3. Can you use the `List_split` and `List_join` (if you implemented them) to improve merge sort?
4. Go through all the defensive programming checks and improve the robustness of this implementation, protecting against bad NULL pointers, and create an optional debug level invariant that does what `is_sorted` does after a sort.

34.3 Extra Credit

1. Create a unit test that compares the performance of the two algorithms. You'll want to look at `man 3 time` for a basic timer function, and you'll want to run enough iterations to at least have a few seconds of samples.
2. Play with the amount of data in the lists that need to be sorted and see if that changes your timing.
3. Find a way to simulate filling different sized random lists and measuring how long they take, then graph it and see how it compares to the description of the algorithm.
4. Try to explain why sorting linked lists is a really bad idea.
5. Implement a `List_insert_sorted` that will take a given value, and using the `List_compare`, insert the element at the right position so that the list is always sorted. How does using this method compare to sorting a list after you've built it?
6. Try implementing the "bottom up" merge sort on the wikipedia page. The code there is already C so it should be easy to recreate, but try to understand how it's working compared to the slower one I have here.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 35

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 34: Dynamic Array

This is an array that grows on its own and has most of the same features as a linked list. It will usually take up less space, run faster, and has other beneficial properties. This exercise will cover a few of the disadvantages like very slow removal from the front, with a solution (just do it at the end).

A dynamic array is simply an array of void ** pointers that is pre-allocated in one shot and that point at the data. In the linked list you had a full struct that stored the void *value pointer, but in a dynamic array there's just a single array with all of them. This means you don't need any other pointers for next and previous records since you can just index into it directly.

To start, I'll give you the header file you should type up for the implementation:

Source 102: src/lcthw/darray.h

```

1  #ifndef _DArray_h
2  #define _DArray_h
3  #include <stdlib.h>
4  #include <assert.h>
5  #include <lcthw/dbg.h>
6
7  typedef struct DArray {
8      int end;
9      int max;
10     size_t element_size;
11     size_t expand_rate;
12     void **contents;
13 } DArray;
14
15 DArray *DArray_create(size_t element_size, size_t initial_max);
16
17 void DArray_destroy(DArray *array);
18
19 void DArray_clear(DArray *array);
20
21 int DArray_expand(DArray *array);
22
23 int DArray_contract(DArray *array);
24
25 int DArray_push(DArray *array, void *el);
26
27 void *DArray_pop(DArray *array);
28
29 void DArray_clear_destroy(DArray *array);
30
31 #define DArray_last(A) ((A)->contents[(A)->end - 1])
32 #define DArray_first(A) ((A)->contents[0])
33 #define DArray_end(A) ((A)->end)
34 #define DArray_count(A) DArray_end(A)
35 #define DArray_max(A) ((A)->max)
36
37 #define DEFAULT_EXPAND_RATE 300
38
39
```

```

40 static inline void DArray_set(DArray *array, int i, void *el)
41 {
42     check(i < array->max, "darray attempt to set past max");
43     array->contents[i] = el;
44 error:
45     return;
46 }
47
48 static inline void *DArray_get(DArray *array, int i)
49 {
50     check(i < array->max, "darray attempt to get past max");
51     return array->contents[i];
52 error:
53     return NULL;
54 }
55
56 static inline void *DArray_remove(DArray *array, int i)
57 {
58     void *el = array->contents[i];
59
60     array->contents[i] = NULL;
61
62     return el;
63 }
64
65 static inline void *DArray_new(DArray *array)
66 {
67     check(array->element_size > 0, "Can't use DArray_new on 0 size darrays.");
68
69     return calloc(1, array->element_size);
70
71 error:
72     return NULL;
73 }
74
75 #define DArray_free(E) free((E))
76
77 #endif

```

This header file is showing you a new technique where I put static inline functions right in the header. These function definitions will work similar to the #define macros you've been making, but they're cleaner and easier to write. If you need to create a block of code for a macro and you don't need code generation, then use a static inline function.

Compare this technique to the LIST_FOREACH that generates a proper for-loop for a list. This would be impossible to do with a static inline function because it actually has to generate the inner block of code for the loop. The only way to do that is with a callback function, but that's not as fast and is harder to use.

I'll then change things up and have you create the unit test for DArray:

Source 103: tests/darray_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/darray.h>
3
4  static DArray *array = NULL;
5  static int *val1 = NULL;
6  static int *val2 = NULL;
7
8  char *test_create()
9  {
10     array = DArray_create(sizeof(int), 100);
11     mu_assert(array != NULL, "DArray_create failed.");
12     mu_assert(array->contents != NULL, "contents are wrong in darray");
13     mu_assert(array->end == 0, "end isn't at the right spot");
14     mu_assert(array->element_size == sizeof(int), "element size is wrong.");
15     mu_assert(array->max == 100, "wrong max length on initial size");
16
17     return NULL;
18 }
19
20 char *test_destroy()
21 {
22     DArray_destroy(array);
23

```

```

24     return NULL;
25 }
26
27 char *test_new()
28 {
29     val1 = DArray_new(array);
30     mu_assert(val1 != NULL, "failed to make a new element");
31
32     val2 = DArray_new(array);
33     mu_assert(val2 != NULL, "failed to make a new element");
34
35     return NULL;
36 }
37
38 char *test_set()
39 {
40     DArray_set(array, 0, val1);
41     DArray_set(array, 1, val2);
42
43     return NULL;
44 }
45
46 char *test_get()
47 {
48     mu_assert(DArray_get(array, 0) == val1, "Wrong first value.");
49     mu_assert(DArray_get(array, 1) == val2, "Wrong second value.");
50
51     return NULL;
52 }
53
54 char *test_remove()
55 {
56     int *val_check = DArray_remove(array, 0);
57     mu_assert(val_check != NULL, "Should not get NULL.");
58     mu_assert(*val_check == *val1, "Should get the first value.");
59     mu_assert(DArray_get(array, 0) == NULL, "Should be gone.");
60     DArray_free(val_check);
61
62     val_check = DArray_remove(array, 1);
63     mu_assert(val_check != NULL, "Should not get NULL.");
64     mu_assert(*val_check == *val2, "Should get the first value.");
65     mu_assert(DArray_get(array, 1) == NULL, "Should be gone.");
66     DArray_free(val_check);
67
68     return NULL;
69 }
70
71 char *test_expand_contract()
72 {
73     int old_max = array->max;
74     DArray_expand(array);
75     mu_assert((unsigned int)array->max == old_max + array->expand_rate, "Wrong size after expand.");
76
77     DArray_contract(array);
78     mu_assert((unsigned int)array->max == array->expand_rate + 1, "Should stay at the expand_rate at least.");
79
80     DArray_contract(array);
81     mu_assert((unsigned int)array->max == array->expand_rate + 1, "Should stay at the expand_rate at least.");
82
83     return NULL;
84 }
85
86 char *test_push_pop()
87 {
88     int i = 0;
89     for(i = 0; i < 1000; i++) {
90         int *val = DArray_new(array);
91         *val = i * 333;
92         DArray_push(array, val);
93     }
94
95     mu_assert(array->max == 1201, "Wrong max size.");
96
97     for(i = 999; i >= 0; i--) {
98         int *val = DArray_pop(array);
99         mu_assert(val != NULL, "Shouldn't get a NULL.");
100        mu_assert(*val == i * 333, "Wrong value.");
101        DArray_free(val);
102    }
103
104    return NULL;
105 }

```



```

106
107
108 char * all_tests() {
109     mu_suite_start();
110
111     mu_run_test(test_create);
112     mu_run_test(test_new);
113     mu_run_test(test_set);
114     mu_run_test(test_get);
115     mu_run_test(test_remove);
116     mu_run_test(test_expand_contract);
117     mu_run_test(test_push_pop);
118     mu_run_test(test_destroy);
119
120     return NULL;
121 }
122
123 RUN_TESTS(all_tests);

```

This shows you how all of the operations are used, which then makes implementing the DArray much easier:

Source 104: src/lcthw/darray.c

```

1  #include <lcthw/darray.h>
2  #include <assert.h>
3
4
5  DArray *DArray_create(size_t element_size, size_t initial_max)
6  {
7      DArray *array = malloc(sizeof(DArray));
8      check_mem(array);
9      array->max = initial_max;
10     check(array->max > 0, "You must set an initial_max > 0.");
11
12     array->contents = calloc(initial_max, sizeof(void *));
13     check_mem(array->contents);
14
15     array->end = 0;
16     array->element_size = element_size;
17     array->expand_rate = DEFAULT_EXPAND_RATE;
18
19     return array;
20
21 error:
22     if(array) free(array);
23     return NULL;
24 }
25
26 void DArray_clear(DArray *array)
27 {
28     int i = 0;
29     if(array->element_size > 0) {
30         for(i = 0; i < array->max; i++) {
31             if(array->contents[i] != NULL) {
32                 free(array->contents[i]);
33             }
34         }
35     }
36 }
37
38 static inline int DArray_resize(DArray *array, size_t newsize)
39 {
40     array->max = newsize;
41     check(array->max > 0, "The newsize must be > 0.");
42
43     void *contents = realloc(array->contents, array->max * sizeof(void *));
44     // check contents and assume realloc doesn't harm the original on error
45
46     check_mem(contents);
47
48     array->contents = contents;
49
50     return 0;
51 error:
52     return -1;
53 }
54
55 int DArray_expand(DArray *array)

```

```

56 {
57     size_t old_max = array->max;
58     check(DArray_resize(array, array->max + array->expand_rate) == 0,
59         "Failed to expand array to new size: %d",
60         array->max + (int)array->expand_rate);
61
62     memset(array->contents + old_max, 0, array->expand_rate + 1);
63     return 0;
64
65 error:
66     return -1;
67 }
68
69 int DArray_contract(DArray *array)
70 {
71     int new_size = array->end < (int)array->expand_rate ? (int)array->expand_rate : array->end;
72
73     return DArray_resize(array, new_size + 1);
74 }
75
76
77 void DArray_destroy(DArray *array)
78 {
79     if(array) {
80         if(array->contents) free(array->contents);
81         free(array);
82     }
83 }
84
85 void DArray_clear_destroy(DArray *array)
86 {
87     DArray_clear(array);
88     DArray_destroy(array);
89 }
90
91 int DArray_push(DArray *array, void *el)
92 {
93     array->contents[array->end] = el;
94     array->end++;
95
96     if(DArray_end(array) >= DArray_max(array)) {
97         return DArray_expand(array);
98     } else {
99         return 0;
100     }
101 }
102
103 void *DArray_pop(DArray *array)
104 {
105     check(array->end - 1 >= 0, "Attempt to pop from empty array.");
106
107     void *el = DArray_remove(array, array->end - 1);
108     array->end--;
109
110     if(DArray_end(array) > (int)array->expand_rate && DArray_end(array) % array->expand_rate) {
111         DArray_contract(array);
112     }
113
114     return el;
115 error:
116     return NULL;
117 }

```

This shows you another way to tackle complex code. Instead of diving right into the .c implementation, look at the header file, then read the unit test. This gives you an "abstract to concrete" understanding how the pieces work together and making it easier to remember.

35.1 Advantages And Disadvantages

A DArray is better when you need to optimize these operations:

1. Iteration. You can just use a basic for-loop and DArray_count with DArray_get and you're done. No special macros needed, and it's faster because you aren't walking pointers.

2. Indexing. You can use `DArray_get` and `DArray_set` to access any element at random, but with a List you have to go through N elements to get to $N+1$.
3. Destroying. You just free the struct and the contents in two operations. A List requires a series of free calls and also walking every element.
4. Cloning. You can also clone it in just two operations (plus whatever it's storing) by copying the struct and contents. A list again requires walking the whole thing and copying every `ListNode` plus its value.
5. Sorting. As you saw, List is horrible if you need to keep the data sorted. A DArray opens up a whole class of great sorting algorithms because now you can access elements randomly.
6. Large Data. If you need to keep around a lot of data, then a DArray wins since it's base contents takes up less memory than the same number of `ListNode` structs.

The List however wins on these operations:

1. Insert and remove on the front (what I called shift). A DArray needs special treatment to be able to do this efficiently, and usually has to do some copying.
2. Splitting or joining. A List can just copy some pointers and it's done, but with a DArray you have to do copying of the arrays involved.
3. Small Data. If you only need to store a few elements, then typically the storage will be less in a List than a generic DArray because the DArray needs to expand the backing store to accommodate future inserts, but a List only makes what it needs.

With this, I prefer to use a DArray for most of the things you see other people use a List. I reserve using List for any data structure that requires small number of nodes that are inserted and removed from either end. I'll show you two similar data structures called a Stack and Queue where this is important.

35.2 How To Improve It

As usual, go through each function and operation and add the defensive programming checks, pre-conditions, invariants, and anything else you can find to make the implementation more bulletproof.

35.3 Extra Credit

1. Improve the unit tests to cover more of the operations and test that using a for-loop to iterate works.
2. Research what it would take to implement bubble sort and merge sort for DArray, but don't do it yet. I'll be implementing DArray algorithms next and you'll do this then.
3. Write some performance tests for common operations and compare them to the same operations in List. You did some of this, but this time, write a unit test that repeatedly does the operation in question, then in the main runner do the timing.
4. Look at how the `DArray_expand` is implemented using a constant increase ($\text{size} + 300$). Typically dynamic arrays are implemented with a multiplicative increase ($\text{size} * 2$), but I've found this to cost needless memory for no real performance gain. Test my assertion and see when you'd want a multiplied increase instead of a constant increase.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 36

Exercise 35: Sorting And Searching

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

In this exercise I'm going to cover four sorting algorithms and one search algorithm. The sorting algorithms are going to be quick sort, heap sort, merge sort, and radix sort. I'm then going to show you how to binary search after you've done a radix sort.

However, I'm a lazy guy, and in most standard C libraries you have existing implementations of the heapsort, quicksort, and mergesort algorithms. Here's how you use them:

Source 105: src/lcthw/darray_algos.c

```

1  #include <lcthw/darray_algos.h>
2  #include <stdlib.h>
3
4  int DArray_qsort(DArray *array, DArray_compare cmp)
5  {
6      qsort(array->contents, DArray_count(array), sizeof(void *), cmp);
7      return 0;
8  }
9
10 int DArray_heapsort(DArray *array, DArray_compare cmp)
11 {
12     return heapsort(array->contents, DArray_count(array), sizeof(void *), cmp);
13 }
14
15 int DArray_mergesort(DArray *array, DArray_compare cmp)
16 {
17     return mergesort(array->contents, DArray_count(array), sizeof(void *), cmp);
18 }
```

That's the whole implementation of the darray_algos.c file, and it should work on most modern Unix systems. What each of these does is sort the contents store of void pointers using the DArray_compare you give it. I'll show you the header file for this too:

Source 106: src/lcthw/darray_algos.h

```

1  #ifndef darray_algos_h
2  #define darray_algos_h
3
4  #include <lcthw/darray.h>
5
6  typedef int (*DArray_compare)(const void *a, const void *b);
7
8  int DArray_qsort(DArray *array, DArray_compare cmp);
9
10 int DArray_heapsort(DArray *array, DArray_compare cmp);
11
12 int DArray_mergesort(DArray *array, DArray_compare cmp);
13
14 #endif
```

About the same size and should be what you expect. Next you can see how these functions are used in the unit test for these three:

Source 107: tests/darray_algos_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/darray_algos.h>
3
4  int testcmp(char **a, char **b)
5  {
6      return strcmp(*a, *b);
7  }
8
9  DArray *create_words()
10 {
11     DArray *result = DArray_create(0, 5);
12     char *words[] = {"asdfasfd", "werwar", "13234", "asdfasfd", "oioj"};
13     int i = 0;
14
15     for(i = 0; i < 5; i++) {
16         DArray_push(result, words[i]);
17     }
18
19     return result;
20 }
21
22 int is_sorted(DArray *array)
23 {
24     int i = 0;
25
26     for(i = 0; i < DArray_count(array) - 1; i++) {
27         if(strcmp(DArray_get(array, i), DArray_get(array, i+1)) > 0) {
28             return 0;
29         }
30     }
31
32     return 1;
33 }
34
35 char *run_sort_test(int (*func)(DArray *, DArray_compare), const char *name)
36 {
37     DArray *words = create_words();
38     mu_assert(!is_sorted(words), "Words should start not sorted.");
39
40     debug("--- Testing %s sorting algorithm", name);
41     int rc = func(words, (DArray_compare)testcmp);
42     mu_assert(rc == 0, "sort failed");
43     mu_assert(is_sorted(words), "didn't sort it");
44
45     DArray_destroy(words);
46
47     return NULL;
48 }
49
50 char *test_qsort()
51 {
52     return run_sort_test(DArray_qsort, "qsort");
53 }
54
55 char *test_heapsort()
56 {
57     return run_sort_test(DArray_heapsort, "heapsort");
58 }
59
60 char *test_mergesort()
61 {
62     return run_sort_test(DArray_mergesort, "mergesort");
63 }
64
65
66 char *all_tests()
67 {
68     mu_suite_start();
69
70     mu_run_test(test_qsort);
71     mu_run_test(test_heapsort);
72     mu_run_test(test_mergesort);
73

```

```

74     return NULL;
75 }
76
77 RUN_TESTS(all_tests);

```

The thing to notice, and actually what tripped me up for a whole day, is the definition of `testcmp` on line 4. You have to use a `char **` and not a `char *` because `qsort` is going to give you a pointer to the pointers in the contents array. The reason is `qsort` and friends are scanning the array, and handing pointers to each element in the array to your comparison function. Since what I have in the contents array is pointers, that means you get a pointer to a pointer.

With that out of the way you have to just implemented three difficult sorting algorithms in about 20 lines of code. You could stop there, but part of this book is learning how these algorithms work so the extra credit is going to involve implementing each of these.

36.1 Radix Sort And Binary Search

Since you're going to implement quicksort, heapsort, and mergesort on your own, I'm going to show you a funky algorithm called Radix Sort. It has a slightly narrow usefulness in sorting arrays of integers, and seems to work like magic. In this case I'm going to create a special data structure called a `RadixMap` that is used to map one integer to another.

Here's the header file for the new algorithm that is both algorithm and data structure in one:

Source 108: `src/lcthw/radixmap.h`

```

1  #ifndef _radixmap_h
2  #include <stdint.h>
3
4  typedef union RMElement {
5      uint64_t raw;
6      struct {
7          uint32_t key;
8          uint32_t value;
9      } data;
10 } RMElement;
11
12 typedef struct RadixMap {
13     size_t max;
14     size_t end;
15     uint32_t counter;
16     RMElement *contents;
17     RMElement *temp;
18 } RadixMap;
19
20
21 RadixMap *RadixMap_create(size_t max);
22
23 void RadixMap_destroy(RadixMap *map);
24
25 void RadixMap_sort(RadixMap *map);
26
27 RMElement *RadixMap_find(RadixMap *map, uint32_t key);
28
29 int RadixMap_add(RadixMap *map, uint32_t key, uint32_t value);
30
31 int RadixMap_delete(RadixMap *map, RMElement *el);
32
33 #endif

```

You see I have a lot of the same operations as in a Dynamic Array or a List data structure, the difference is I'm working only with fixed size 32 bit `uint32_t` integers. I'm also introducing you to a new C concept called the union here.

36.1.1 C Unions

A union is a way to refer to the same piece of memory in a number of different ways. How they work is you define them like a struct except every element is sharing the same space with all of the others. You can think of a union as a picture of the memory, and the elements in the union as different colored lenses to view the picture.

What they are used for is to either save memory, or to convert chunks of memory between formats. The first usage is typically done with "variant types", where you create a struct that has "tag" for the type, and then a union inside it for each type. When used for converting between formats of memory, you simply define the two structures, and then access the right one.

First let me show you how to make a variant type with C unions:

Source 109: ex35.c

```

1  #include <stdio.h>
2
3  typedef enum {
4      TYPE_INT,
5      TYPE_FLOAT,
6      TYPE_STRING,
7  } VariantType;
8
9  struct Variant {
10     VariantType type;
11     union {
12         int as_integer;
13         float as_float;
14         char *as_string;
15     } data;
16 };
17
18 typedef struct Variant Variant;
19
20 void Variant_print(Variant *var)
21 {
22     switch(var->type) {
23         case TYPE_INT:
24             printf("INT: %d\n", var->data.as_integer);
25             break;
26         case TYPE_FLOAT:
27             printf("FLOAT: %f\n", var->data.as_float);
28             break;
29         case TYPE_STRING:
30             printf("STRING: %s\n", var->data.as_string);
31             break;
32         default:
33             printf("UNKNOWN TYPE: %d", var->type);
34     }
35 }
36
37 int main(int argc, char *argv[])
38 {
39     Variant a_int = {.type = TYPE_INT, .data.as_integer = 100};
40     Variant a_float = {.type = TYPE_FLOAT, .data.as_float = 100.34};
41     Variant a_string = {.type = TYPE_STRING, .data.as_string = "YO DUDE!"};
42
43     Variant_print(&a_int);
44     Variant_print(&a_float);
45     Variant_print(&a_string);
46
47     // here's how you access them
48     a_int.data.as_integer = 200;
49     a_float.data.as_float = 2.345;
50     a_string.data.as_string = "Hi there.";
51
52     Variant_print(&a_int);
53     Variant_print(&a_float);
54     Variant_print(&a_string);
55
56     return 0;
57 }

```

You find this in many implementations of dynamic languages. The language will define some base variant type with tags

for all the base types of the language, and then usually there's a generic "object" tag for the types you create. The advantage of doing this is that the Variant only takes up as much space as the VariantType type tag and the largest member of the union. This is because C is "layering" each element of the Variant.data union together so they overlap, and to do that it sizes it big enough to hold the largest element.

In the radixmap.h file I have the RMElement union which demonstrates using a union to convert blocks of memory between types. In this case, I want to store a uint64_t sized integer for sorting purposes, but I want a two uint32_t integers for the data to represent a key and value pair. By using a union I'm able to access the same block of memory in the two different ways I need cleanly.

36.1.2 The Implementation

I next have the actual RadixMap implementation for each of these operations:

Source 110: src/lcthw/radixmap.c

```

1  /*
2  * Based on code by Andre Reinald then heavily modified by Zed A. Shaw.
3  */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <assert.h>
8  #include <lcthw/radixmap.h>
9  #include <lcthw/dbg.h>
10
11 RadixMap *RadixMap_create(size_t max)
12 {
13     RadixMap *map = calloc(sizeof(RadixMap), 1);
14     check_mem(map);
15
16     map->contents = calloc(sizeof(RMElement), max + 1);
17     check_mem(map->contents);
18
19     map->temp = calloc(sizeof(RMElement), max + 1);
20     check_mem(map->temp);
21
22     map->max = max;
23     map->end = 0;
24
25     return map;
26 error:
27     return NULL;
28 }
29
30 void RadixMap_destroy(RadixMap *map)
31 {
32     if(map) {
33         free(map->contents);
34         free(map->temp);
35         free(map);
36     }
37 }
38
39
40 #define ByteOf(x,y) (((uint8_t *)x)[(y)])
41
42 static inline void radix_sort(short offset, uint64_t max, uint64_t *source, uint64_t *dest)
43 {
44     uint64_t count[256] = {0};
45     uint64_t *cp = NULL;
46     uint64_t *sp = NULL;
47     uint64_t *end = NULL;
48     uint64_t s = 0;
49     uint64_t c = 0;
50
51     // count occurrences of every byte value
52     for (sp = source, end = source + max; sp < end; sp++) {
53         count[ByteOf(sp, offset)]++;
54     }
55
56     // transform count into index by summing elements and storing into same array
57     for (s = 0, cp = count, end = count + 256; cp < end; cp++) {

```

```

58     c = *cp;
59     *cp = s;
60     s += c;
61 }
62
63 // fill dest with the right values in the right place
64 for (sp = source, end = source + max; sp < end; sp++) {
65     cp = count + ByteOf(sp, offset);
66     dest[*cp] = *sp;
67     ++(*cp);
68 }
69 }
70
71 void RadixMap_sort(RadixMap *map)
72 {
73     uint64_t *source = &map->contents[0].raw;
74     uint64_t *temp = &map->temp[0].raw;
75
76     radix_sort(0, map->end, source, temp);
77     radix_sort(1, map->end, temp, source);
78     radix_sort(2, map->end, source, temp);
79     radix_sort(3, map->end, temp, source);
80 }
81
82 RMElement *RadixMap_find(RadixMap *map, uint32_t to_find)
83 {
84     int low = 0;
85     int high = map->end - 1;
86     RMElement *data = map->contents;
87
88     while (low <= high) {
89         int middle = low + (high - low)/2;
90         uint32_t key = data[middle].data.key;
91
92         if (to_find < key) {
93             high = middle - 1;
94         } else if (to_find > key) {
95             low = middle + 1;
96         } else {
97             return &data[middle];
98         }
99     }
100
101     return NULL;
102 }
103
104 int RadixMap_add(RadixMap *map, uint32_t key, uint32_t value)
105 {
106     check(key < UINT32_MAX, "Key can't be equal to UINT32_MAX.");
107
108     RMElement element = {.data = {.key = key, .value = value}};
109     check(map->end + 1 < map->max, "RadixMap is full.");
110
111     map->contents[map->end++] = element;
112
113     RadixMap_sort(map);
114
115     return 0;
116
117 error:
118     return -1;
119 }
120
121 int RadixMap_delete(RadixMap *map, RMElement *el)
122 {
123     check(map->end > 0, "There is nothing to delete.");
124     check(el != NULL, "Can't delete a NULL element.");
125
126     el->data.key = UINT32_MAX;
127
128     if (map->end > 1) {
129         // don't bother resorting a map of 1 length
130         RadixMap_sort(map);
131     }
132
133     map->end--;
134
135     return 0;
136 error:
137     return -1;
138 }

```

As usual enter this in and get it working along with the unit test then I'll explain what's happening. Take special care with the radix_sort function as it's very particular in how it's implemented.

Source 111: tests/radixmap_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/radixmap.h>
3  #include <time.h>
4
5  static int make_random(RadixMap *map)
6  {
7      size_t i = 0;
8
9      for (i = 0; i < map->max - 1; i++) {
10         uint32_t key = (uint32_t)(rand() | (rand() << 16));
11         check(RadixMap_add(map, key, i) == 0, "Failed to add key %u.", key);
12     }
13
14     return i;
15
16 error:
17     return 0;
18 }
19
20 static int check_order(RadixMap *map)
21 {
22     RMElement d1, d2;
23     unsigned int i = 0;
24
25     // only signal errors if any (should not be)
26     for (i = 0; map->end > 0 && i < map->end - 1; i++) {
27         d1 = map->contents[i];
28         d2 = map->contents[i + 1];
29
30         if(d1.data.key > d2.data.key) {
31             debug("FAIL:i=%u, key: %u, value: %u, equals max? %d\n", i, d1.data.key, d1.data.value,
32                 d2.data.key == UINT32_MAX);
33             return 0;
34         }
35     }
36
37     return 1;
38 }
39
40 static int test_search(RadixMap *map)
41 {
42     unsigned i = 0;
43     RMElement *d = NULL;
44     RMElement *found = NULL;
45
46     for(i = map->end / 2; i < map->end; i++) {
47         d = &map->contents[i];
48         found = RadixMap_find(map, d->data.key);
49         check(found != NULL, "Didn't find %u at %u.", d->data.key, i);
50         check(found->data.key == d->data.key, "Got the wrong result: %p:%u looking for %u at %u",
51             found, found->data.key, d->data.key, i);
52     }
53
54     return 1;
55 error:
56     return 0;
57 }
58
59 // test for big number of elements
60 static char *test_operations()
61 {
62     size_t N = 200;
63
64     RadixMap *map = RadixMap_create(N);
65     mu_assert(map != NULL, "Failed to make the map.");
66     mu_assert(make_random(map), "Didn't make a random fake radix map.");
67
68     RadixMap_sort(map);
69     mu_assert(check_order(map), "Failed to properly sort the RadixMap.");
70
71     mu_assert(test_search(map), "Failed the search test.");
72     mu_assert(check_order(map), "RadixMap didn't stay sorted after search.");
73

```

```

74 while(map->end > 0) {
75     RMElement *el = RadixMap_find(map, map->contents[map->end / 2].data.key);
76     mu_assert(el != NULL, "Should get a result.");
77
78     size_t old_end = map->end;
79
80     mu_assert(RadixMap_delete(map, el) == 0, "Didn't delete it.");
81     mu_assert(old_end - 1 == map->end, "Wrong size after delete.");
82
83     // test that the end is now the old value, but uint32 max so it trails off
84     mu_assert(check_order(map), "RadixMap didn't stay sorted after delete.");
85 }
86
87 RadixMap_destroy(map);
88
89 return NULL;
90 }
91
92
93 char *all_tests()
94 {
95     mu_suite_start();
96     srand(time(NULL));
97     mu_run_test(test_operations);
98
99     return NULL;
100 }
101
102
103 RUN_TESTS(all_tests);

```

I shouldn't have to explain too much about the test. It's simply simulating placing random integers into the RadixMap and then making sure it can get them out reliably. Not too interesting.

In the radixmap.c file most of the operations are easy to understand if you read the code. Here's a description of what the basic functions are doing and how they work:

RadixMap_create

As usual I'm allocating all the memory needed for the structures defined in radixmap.h. I'll be using the temp and contents later when I talk about radix_sort.

RadixMap_destroy

Again, just destroying what was created.

radix_sort

The meat of the data structure, but I'll explain what it's doing in the next section.

RadixMap_sort

This uses the radix_sort function to actually sort the contents. It does this by sorting between the contents and temp until finally contents is sorted. You'll see how this works when I describe radix_sort later.

RadixMap_find

This is using a binary search algorithm to find a key you give it. I'll explain how this works shortly.

RadixMap_add

Using the RadixMap_sort function, this will add the key and value you request at the end, then simply sort it again so that everything is in the right place. Once everything is sorted, the RadixMap_find will work properly because it's a binary search.

RadixMap_delete

Works the same as RadixMap_add except "deletes" elements of the structure by setting their values to the max for a unsigned 32 bit integer, UINT32_MAX. This means you can't use that value as an key value, but it makes deleting elements easy. Simply set it to that and then sort and it'll get moved to the end. Now it's deleted.

Study the code for the ones I described, and then that just leaves RadixMap_sort, radix_sort, and RadixMap_find to understand.

36.1.3 RadixMap_find And Binary Search

I'll start with how the binary search is implemented. Binary search is simple algorithm that most people can understand intuitively. In fact, you could take a deck of playing cards (or cards with numbers) and do this manually. Here's how this function works, and how a binary search works:

1. Set a high and low mark based on the size of the array.
2. Get the middle element between the low and high marks.
3. If the key is less-than, then the key must be below the middle. Set high to one less than middle.
4. If the key is greater-than, then the key must be above the middle. Set the low mark one greater than the middle.
5. If it's equal then you found it, stop.
6. Keep looping until low and high pass each other. You don't find it if you exit the loop.

What you are effectively doing is guessing where the key might be by picking the middle and comparing it. Since the data is sorted, you know that the the key has to be above or below this. If it's below, then you just divided the search space in half. You keep going until you either find it or you overlap the boundaries and exhaust the search space.

36.1.4 RadixMap_sort And radix_sort

A radix sort is easy to understand if you try to do it manually first. What this algorithm does is exploit the fact that numbers are stored with a sequence of digits that go from "least significant" to "most significant". It then takes the numbers and buckets them by the digit, and when it has processed all the digits the numbers come out sorted. At first it seems like magic, and honestly looking at the code sure seems like it is, but try doing it manually once.

To do this algorithm write out a bunch of three digit numbers, in a random order, let's say we do 223, 912, 275, 100, 633, 120, and 380.

1. Place the number in buckets by their 1's digit: [380, 100, 120], [912], [633, 223], [275].
2. I now have to go through each of these buckets in order, and then sort it into 10's buckets: [100], [912], [120, 223], [633], [275], [380].
3. Now each bucket contains numbers that are sorted by the 1's then 10's digit. I need to then go through these in order and fill the final 100's buckets: [100, 120], [223, 275], [380], [633], [912].
4. At this point each bucket is sorted by 100's, 10's, then 1's and if I take each bucket in order I get the final sorted list: 100, 120, 223, 275, 380, 633, 912.

Make sure you do this a few times so you understand how it works. It really is a slick little algorithm and most importantly it will work on numbers of arbitrary size, so you can sort really huge numbers because you are just doing them one byte at a time.

In my situation the "digits" are individual 8 bit bytes, so I need 256 buckets to store the distribution of the numbers by their digits. I also need a way to store them such that I don't use too much space. If you look at radix_sort first thing I do is build a count histogram so I know how many occurrences of each digit there are for the given offset.

Once I know the counts for each digit (all 256 of them) I can then use that as distribution points into a target array. For example, if I have 10 bytes that are 0x00, then I know I can place them in the first 10 slots of the target array. This gives me an index for where they go in the target array, which is the second for-loop in radix_sort.

Finally, once I know where they can go in the target array, I simply go through all the digits in the source array, for this offset and place the numbers in their slots in order. Using the ByteOf macro helps keep the code clean since there's a

bit of pointer hackery to make it work, but the end result is all of the integers will be placed in the bucket for their digit when the final for-loop is done.

What becomes interesting is then how I use this in `RadixMap_sort` to sort these 64 bit integers by just the first 32 bits. Remember how I have the key and value in a union for the `RMElement` type? That means to sort this array by the key I only need to sort the first 4 bytes (32 bits / 8 bits per byte) of every integer.

If you look at the `RadixMap_sort` you see that I grab a quick pointer to the contents and temp to for source and target arrays, and then I call `radix_sort` four times. Each time I call it, I alternate source and target and do the next byte. When I'm done, the `radix_sort` has done its job and the final copy has been done into the contents.

36.2 How To Improve It

There is a big disadvantage to this implementation because it has to process the entire array four times on every insertion. It does do it fast, but it'd be better if you could limit the amount of sorting by the size of what needs to be sorted.

There's two ways you can improve this implementation:

1. Use a binary search to find the minimum position for the new element, then only sort from there to the end. You find the minimum, put the new element on the end, then just sort from the minimum on. This will cut your sort space down considerably most of the time.
2. Keep track of the biggest key currently being used, and then only sort enough digits to handle that key. You can also keep track of the smallest number, and then only sort the digits necessary for the range. To do this you'll have to start caring about CPU integer ordering (endianess).

Try these optimizations, but after you augment the unit test with some timing information so you can see if you're actually improving the speed of the implementation.

36.3 Extra Credit

1. Implement quicksort, heapsort, and mergesort and provide a `#define` that lets you pick between the two, or create a second set of functions you can call. Use the technique I taught you to read the Wikipedia page for the algorithm and then implement it with the psuedo-code.
2. Compare the performance of your implementations to the original ones.
3. Use these sorting functions to create a `DArray_sort_add` that adds elements to the `DArray` but sorts the array after.
4. Write a `DArray_find` that uses the binary search algorithm from `RadixMap_find` and the `DArray_compare` to find elements in a sorted `DArray`.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 37

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 36: Safer Strings

I've already introduced you to the [Better String](#) library in Exercise 26 when we made devpkg. This exercise is designed to get you into using bstring from now on, why C's strings are an incredibly bad idea, and then have you change the liblcthw code to use bstring.

37.1 Why C Strings Were A Horrible Idea

When people talk about problems with C, it's concept of a "string" is one of the top flaws. You've been using these extensively, and I've talked about the kinds of flaws they have, but there's not much that explains exactly why C strings are flawed and always will be. I'll try to explain that right now, but part of my explanation will just be that after decades of using C's strings there's enough evidence that they are just a bad idea.

It is impossible to confirm that any given C string is valid:

1. A C string is invalid if it does not end in `'\0'`.
2. Any loop that processes an invalid C string will loop infinitely (or, just buffer overflow).
3. C strings do not have a known length, so the only way to check if it's terminated correctly is to loop through it.
4. Therefore, it is not possible to validate a C string without possibly looping infinitely.

This is simple logic. You can't write a loop that checks if a C string is valid because invalid C strings cause loops to never terminate. That's it, and the only solution is to include the size. Once you know the size you can avoid the infinite loop problem. If you look at the two functions I showed you from Exercise 27 you can see this:

Source 112: ex36.c

```

1 void copy(char to[], char from[])
2 {
3     int i = 0;
4
5     // while loop will not end if from isn't '\0' terminated
6     while((to[i] = from[i]) != '\0') {
7         ++i;
8     }
9 }
10
11 int safercopy(int from_len, char *from, int to_len, char *to)
12 {
13     int i = 0;
14     int max = from_len > to_len - 1 ? to_len - 1 : from_len;
15
16     // to_len must have at least 1 byte
17     if(from_len < 0 || to_len <= 0) return -1;
18
19     for(i = 0; i < max; i++) {
20         to[i] = from[i];

```



```

21 }
22
23 to[to_len - 1] = '\0';
24
25 return i;
26 }

```

Imagine you want to add a check to the copy function to confirm that the from string is valid. How would you do that? Why you'd write a loop that checked that the string ended in '\0'. Oh wait, if the string doesn't end in '\0' then how does the checking loop end? It doesn't. Checkmate.

No matter what you do, you can't check that a C string is valid without knowing the length of the underlying storage, and in this case the `safercopy` includes those lengths. This function doesn't have the same problem as its loops will always terminate, even if you lie to it about the size, you still have to give it a finite size.

What the Better String library does is create a struct that always includes the length of the string's storage. Because the length is always available to a `bstring` then all of its operations can be safer. The loops will terminate, the contents can be validated, and it will not have this major flaw. The `bstring` library also comes with a ton of operations you need with strings, like splitting, formatting, searching, and they are most likely done right and safer.

There could be flaws in `bstring`, but it's been around a long time so those are probably minimal. They still find flaws in `glibc` so what's a programmer to do right?

37.2 Using `bstrlib`

There's quite a few improved string libraries, but I like `bstrlib` because it fits in one file for the basics and has most of the stuff you need to deal with strings. You've already used it a bit, so in this exercise you'll go get the two files `bstrlib.c` and `bstrlib.h` from the [Better String](#) project.

Here's me doing this in the `liblcthw` project directory:

Source 113: Adding `bstrlib.c`

```

1  $ mkdir bstrlib
2  $ cd bstrlib/
3  $ unzip ~/Downloads/bstrlib-05122010.zip
4  Archive:  /Users/zedshaw/Downloads/bstrlib-05122010.zip
5  ...
6  $ ls
7  bsafe.c bstraux.c bstrlib.h bstrwrap.h license.txt test.cpp
8  bsafe.h bstraux.h bstrlib.txt cpptest.cpp porting.txt testaux.c
9  btest.c bstrlib.c bstrwrap.cpp gpl.txt security.txt
10 $ mv bstrlib.h bstrlib.c ../src/lcthw/
11 $ cd ../
12 $ rm -rf bstrlib
13 # make the edits
14 $ vim src/lcthw/bstrlib.c
15 $ make clean all
16 ...
17 $

```

On line 14 you seem me edit the `bstrlib.c` file to move it to a new location and to fix a bug on OSX. Here's the diff:

Source 114: `bstrlib.diff`

```

25c25
< #include "bstrlib.h"
---
> #include <lcthw/bstrlib.h>
2759c2759
< #ifdef __GNUC__
---

```

```
> #if defined(__GNUC__) && !defined(__APPLE__)
```

That is, change the include to be `<lcthw/bstrlib.h>`, and then fix one of the `ifdef` at line 2759.

37.3 Learning The Library

This exercise is short and simply getting you ready for the remaining exercises that use the library. In the next two exercises I'll use `bstrlib.c` to create a Hashmap data structure.

You should now get familiar with this library by reading the header file, the implementations, and then write a `tests/bstr_tests.c` that tests out the following functions:

`bfromcstr`

Create a `bstring` from a C style constant.

`blk2bstr`

Same but give the length of the buffer.

`bstrcpy`

Copy a `bstring`.

`bassign`

Set one `bstring` to another.

`bassigncstr`

Set a `bstring` to a C string's contents.

`bassignblk`

Set a `bstring` to a C string but give the length.

`bdestroy`

Destroy a `bstring`.

`bconcat`

Concatenate one `bstring` onto another.

`bstricmp`

Compare two `bstrings` returning the same result as `strcmp`.

`biseq`

Tests if two `bstrings` are equal.

`binstr`

Tells if one `bstring` is in another.

`bfindreplace`

Find one `bstring` in another then replace it with a third.

`bsplit`

How to split a `bstring` into a `bstrList`.

`bformat`

Doing a format string, super handy.

`blength`

Getting the length of a `bstring`.

`bdata`

Getting the data from a `bstring`.

`bchar`

Getting a char from a `bstring`.

Your test should try out all of these operations, and a few more that you find interesting from the header file. Make sure

to run the test under valgrind to make sure you use the memory correctly.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 38

Exercise 37: Hashmaps

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Hash Maps (Hashmaps, Hashes, or sometimes Dictionaries) are used frequently in many dynamic programming for storing key/value data. A Hashmap works by performing a "hashing" calculation on the keys to produce an integer, then uses that integer to find a bucket to get or set the value. It is a very fast practical data structure since it works on nearly any data and they are easy to implement.

Here's an example of using a Hashmap (aka dict) in Python:

Source 115: ex37.py

```
1 fruit_weights = {'Apples': 10, 'Oranges': 100, 'Grapes': 1.0}
2
3 for key, value in fruit_weights.items():
4     print key, "=", value
```

Almost every modern language has something like this, so many people end up writing code and never understand how this actually works. By creating the Hashmap data structure in C I'll show you how this works. I'll start with the header file so I can talk about the data structure.

Source 116: src/lcthw/hashmap.h

```
1 #ifndef _lcthw_Hashmap_h
2 #define _lcthw_Hashmap_h
3
4 #include <stdint.h>
5 #include <lcthw/darray.h>
6
7 #define DEFAULT_NUMBER_OF_BUCKETS 100
8
9 typedef int (*Hashmap_compare)(void *a, void *b);
10 typedef uint32_t (*Hashmap_hash)(void *key);
11
12 typedef struct Hashmap {
13     DArray *buckets;
14     Hashmap_compare compare;
15     Hashmap_hash hash;
16 } Hashmap;
17
18 typedef struct HashmapNode {
19     void *key;
20     void *data;
21     uint32_t hash;
22 } HashmapNode;
23
24 typedef int (*Hashmap_traverse_cb)(HashmapNode *node);
25
26 Hashmap *Hashmap_create(Hashmap_compare compare, Hashmap_hash);
27 void Hashmap_destroy(Hashmap *map);
28
29 int Hashmap_set(Hashmap *map, void *key, void *data);
```

```

30 void *HashMap_get(HashMap *map, void *key);
31
32 int HashMap_traverse(HashMap *map, HashMap_traverse_cb traverse_cb);
33
34 void *HashMap_delete(HashMap *map, void *key);
35
36 #endif

```

The structure consists of a HashMap that contains any number of HashMapNode structs. Looking at HashMap you can see that it is structured like this:

DArray *buckets

A dynamic array that will be set to a fixed size of 100 buckets. Each bucket will in turn contain a DArray that will actually hold HashMapNode pairs.

HashMap_compare compare

This is a comparison function that the HashMap uses to actually find elements by their key. It should work like all of the other compare functions, and defaults to using bstrcmp so that keys are just bstrings.

HashMap_hash hash

This is the hashing function and it's responsible for taking a key, processing its contents, and producing a single uint32_t index number. You'll see the default one soon.

This almost tells you how the data is stored, but the bucketsDArray isn't created yet. Just remember that it's kind of a two level mapping:

1. There are 100 buckets that make up the first level, and things are in these buckets based on their hash.
2. Each bucket is a DArray that then contains HashMapNode structs simply appended to the end as they're added.

The HashMapNode is then composed of these three elements:

void *key

The key for this key=value pair.

void *value

The value.

uint32_t hash

The calculated hash, which makes finding this node quicker since we can just check the hash and skip any that don't match, only checking they key if it's equal.

The rest of the header file is nothing new, so now I can show you the implementation hashmap.c file:

Source 117: src/lcthw/hashmap.c

```

1  #undef NDEBUG
2  #include <stdint.h>
3  #include <lcthw/hashmap.h>
4  #include <lcthw/dbg.h>
5  #include <lcthw/bstrlib.h>
6
7  static int default_compare(void *a, void *b)
8  {
9      return bstrcmp((bstring)a, (bstring)b);
10 }
11
12 /**
13  * Simple Bob Jenkins's hash algorithm taken from the
14  * wikipedia description.
15  */
16 static uint32_t default_hash(void *a)
17 {
18     size_t len = blength((bstring)a);
19

```

```

    char *key = bdata((bstring)a);
20  uint32_t hash = 0;
21  uint32_t i = 0;
22
23  for(hash = i = 0; i < len; ++i)
24  {
25      hash += key[i];
26      hash += (hash << 10);
27      hash ^= (hash >> 6);
28  }
29
30  hash += (hash << 3);
31  hash ^= (hash >> 11);
32  hash += (hash << 15);
33
34  return hash;
35 }
36
37
38 Hashmap *Hashmap_create(Hashmap_compare compare, Hashmap_hash hash)
39 {
40     Hashmap *map = calloc(1, sizeof(Hashmap));
41     check_mem(map);
42
43     map->compare = compare == NULL ? default_compare : compare;
44     map->hash = hash == NULL ? default_hash : hash;
45     map->buckets = DArray_create(sizeof(DArray *), DEFAULT_NUMBER_OF_BUCKETS);
46     map->buckets->end = map->buckets->max; // fake out expanding it
47     check_mem(map->buckets);
48
49     return map;
50
51 error:
52     if(map) {
53         Hashmap_destroy(map);
54     }
55
56     return NULL;
57 }
58
59
60 void Hashmap_destroy(Hashmap *map)
61 {
62     int i = 0;
63     int j = 0;
64
65     if(map) {
66         if(map->buckets) {
67             for(i = 0; i < DArray_count(map->buckets); i++) {
68                 DArray *bucket = DArray_get(map->buckets, i);
69                 if(bucket) {
70                     for(j = 0; j < DArray_count(bucket); j++) {
71                         free(DArray_get(bucket, j));
72                     }
73                     DArray_destroy(bucket);
74                 }
75             }
76             DArray_destroy(map->buckets);
77         }
78
79         free(map);
80     }
81 }
82
83 static inline HashmapNode *Hashmap_node_create(int hash, void *key, void *data)
84 {
85     HashmapNode *node = calloc(1, sizeof(HashmapNode));
86     check_mem(node);
87
88     node->key = key;
89     node->data = data;
90     node->hash = hash;
91
92     return node;
93
94 error:
95     return NULL;
96 }
97
98
99 static inline DArray *Hashmap_find_bucket(Hashmap *map, void *key,
100     int create, uint32_t *hash_out)

```

```

101 {
102     uint32_t hash = map->hash(key);
103     int bucket_n = hash % DEFAULT_NUMBER_OF_BUCKETS;
104     check(bucket_n >= 0, "Invalid bucket found: %d", bucket_n);
105     *hash_out = hash; // store it for the return so the caller can use it
106
107
108     DArray *bucket = DArray_get(map->buckets, bucket_n);
109
110     if(!bucket && create) {
111         // new bucket, set it up
112         bucket = DArray_create(sizeof(void *), DEFAULT_NUMBER_OF_BUCKETS);
113         check_mem(bucket);
114         DArray_set(map->buckets, bucket_n, bucket);
115     }
116
117     return bucket;
118
119 error:
120     return NULL;
121 }
122
123
124 int Hashmap_set(Hashmap *map, void *key, void *data)
125 {
126     uint32_t hash = 0;
127     DArray *bucket = Hashmap_find_bucket(map, key, 1, &hash);
128     check(bucket, "Error can't create bucket.");
129
130     HashmapNode *node = Hashmap_node_create(hash, key, data);
131     check_mem(node);
132
133     DArray_push(bucket, node);
134
135     return 0;
136
137 error:
138     return -1;
139 }
140
141 static inline int Hashmap_get_node(Hashmap *map, uint32_t hash, DArray *bucket, void *key)
142 {
143     int i = 0;
144
145     for(i = 0; i < DArray_end(bucket); i++) {
146         debug("TRY: %d", i);
147         HashmapNode *node = DArray_get(bucket, i);
148         if(node->hash == hash && map->compare(node->key, key) == 0) {
149             return i;
150         }
151     }
152
153     return -1;
154 }
155
156 void *Hashmap_get(Hashmap *map, void *key)
157 {
158     uint32_t hash = 0;
159     DArray *bucket = Hashmap_find_bucket(map, key, 0, &hash);
160     if(!bucket) return NULL;
161
162     int i = Hashmap_get_node(map, hash, bucket, key);
163     if(i == -1) return NULL;
164
165     HashmapNode *node = DArray_get(bucket, i);
166     check(node != NULL, "Failed to get node from bucket when it should exist.");
167
168     return node->data;
169
170 error: // fallthrough
171     return NULL;
172 }
173
174
175 int Hashmap_traverse(Hashmap *map, Hashmap_traverse_cb traverse_cb)
176 {
177     int i = 0;
178     int j = 0;
179     int rc = 0;
180
181     for(i = 0; i < DArray_count(map->buckets); i++) {
182         DArray *bucket = DArray_get(map->buckets, i);

```

```

183     if(bucket) {
184         for(j = 0; j < DArray_count(bucket); j++) {
185             HashmapNode *node = DArray_get(bucket, j);
186             rc = traverse_cb(node);
187             if(rc != 0) return rc;
188         }
189     }
190 }
191
192 return 0;
193 }
194
195 void *Hashmap_delete(Hashmap *map, void *key)
196 {
197     uint32_t hash = 0;
198     DArray *bucket = Hashmap_find_bucket(map, key, 0, &hash);
199     if(!bucket) return NULL;
200
201     int i = Hashmap_get_node(map, hash, bucket, key);
202     if(i == -1) return NULL;
203
204     HashmapNode *node = DArray_get(bucket, i);
205     void *data = node->data;
206     free(node);
207
208     HashmapNode *ending = DArray_pop(bucket);
209
210     if(ending != node) {
211         // alright looks like it's not the last one, swap it
212         DArray_set(bucket, i, ending);
213     }
214
215     return data;
216 }

```

There's nothing very complicated in the implementation, but the `default_hash` and `Hashmap_find_bucket` functions will need some explanation. When you use `Hashmap_create` you can pass in any compare and hash functions you want, but if you don't it uses the `default_compare` and `default_hash` functions.

The first thing to look at is how `default_hash` does its thing. This is a simple hash function called a "Jenkins hash" after Bob Jenkins. I got it from the [Wikipedia page](#) for the algorithm. It simply goes through each byte of the key to hash (a `bstring`) and works the bits so that the end result is a single `uint32_t`. It does this with some adding and xor operations.

There are many different hash functions, all with different properties, but once you have one you need a way to use it to find the right buckets. The `Hashmap_find_bucket` does it like this:

1. First it calls `map->hash(key)` to get the hash for the key.
2. It then finds the bucket using `hash % DEFAULT_NUMBER_OF_BUCKETS`, that way every hash will always find some bucket no matter how big it is.
3. It then gets the bucket, which is also a `DArray`, and if it's not there it will create it. That depends on if the create variable says too.
4. Once it has found the `DArray` bucket for the right hash, it returns it, and also the `hash_out` variable is used to give the caller the hash that was found.

All of the other functions then use `Hashmap_find_bucket` to do their work:

1. Setting a key/value involves finding the bucket, then making a `HashmapNode`, and then adding it to the bucket.
2. Getting a key involves finding the bucket, then finding the `HashmapNode` that matches the hash and key you want.
3. Deleting an item again finds the bucket, finds where the requested node is, and then removes it by swapping the last node into its place.

The only other function that you should study is the `Hashmap_travers`. This simply walks every bucket, and for any bucket that has possible values, it calls the `traverse_cb` on each value. This is how you scan a whole `Hashmap` for its values.

38.0.1 The Unit Test

Finally you have the unit test that is testing all of these operations:

Source 118: tests/hashmap_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/hashmap.h>
3  #include <assert.h>
4  #include <lcthw/bstring.h>
5
6  Hashmap *map = NULL;
7  static int traverse_called = 0;
8  struct tagbstring test1 = bsStatic("test data 1");
9  struct tagbstring test2 = bsStatic("test data 2");
10 struct tagbstring test3 = bsStatic("test data 3");
11 struct tagbstring expect1 = bsStatic("THE VALUE 1");
12 struct tagbstring expect2 = bsStatic("THE VALUE 2");
13 struct tagbstring expect3 = bsStatic("THE VALUE 3");
14
15 static int traverse_good_cb(HashmapNode *node)
16 {
17     debug("KEY: %s", bdata((bstring)node->key));
18     traverse_called++;
19     return 0;
20 }
21
22
23 static int traverse_fail_cb(HashmapNode *node)
24 {
25     debug("KEY: %s", bdata((bstring)node->key));
26     traverse_called++;
27
28     if(traverse_called == 2) {
29         return 1;
30     } else {
31         return 0;
32     }
33 }
34
35
36 char *test_create()
37 {
38     map = Hashmap_create(NULL, NULL);
39     mu_assert(map != NULL, "Failed to create map.");
40
41     return NULL;
42 }
43
44 char *test_destroy()
45 {
46     Hashmap_destroy(map);
47
48     return NULL;
49 }
50
51
52 char *test_get_set()
53 {
54     int rc = Hashmap_set(map, &test1, &expect1);
55     mu_assert(rc == 0, "Failed to set &test1");
56     bstring result = Hashmap_get(map, &test1);
57     mu_assert(result == &expect1, "Wrong value for test1.");
58
59     rc = Hashmap_set(map, &test2, &expect2);
60     mu_assert(rc == 0, "Failed to set test2");
61     result = Hashmap_get(map, &test2);
62     mu_assert(result == &expect2, "Wrong value for test2.");
63
64     rc = Hashmap_set(map, &test3, &expect3);
65     mu_assert(rc == 0, "Failed to set test3");
66     result = Hashmap_get(map, &test3);
67     mu_assert(result == &expect3, "Wrong value for test3.");
68
69     return NULL;
70 }
71
72 char *test_traverse()
```

```

73 {
74     int rc = Hashmap_traverse(map, traverse_good_cb);
75     mu_assert(rc == 0, "Failed to traverse.");
76     mu_assert(traverse_called == 3, "Wrong count traverse.");
77
78     traverse_called = 0;
79     rc = Hashmap_traverse(map, traverse_fail_cb);
80     mu_assert(rc == 1, "Failed to traverse.");
81     mu_assert(traverse_called == 2, "Wrong count traverse for fail.");
82
83     return NULL;
84 }
85
86 char *test_delete()
87 {
88     bstring deleted = (bstring)Hashmap_delete(map, &test1);
89     mu_assert(deleted != NULL, "Got NULL on delete.");
90     mu_assert(deleted == &expect1, "Should get test1");
91     bstring result = Hashmap_get(map, &test1);
92     mu_assert(result == NULL, "Should delete.");
93
94     deleted = (bstring)Hashmap_delete(map, &test2);
95     mu_assert(deleted != NULL, "Got NULL on delete.");
96     mu_assert(deleted == &expect2, "Should get test2");
97     result = Hashmap_get(map, &test2);
98     mu_assert(result == NULL, "Should delete.");
99
100    deleted = (bstring)Hashmap_delete(map, &test3);
101    mu_assert(deleted != NULL, "Got NULL on delete.");
102    mu_assert(deleted == &expect3, "Should get test3");
103    result = Hashmap_get(map, &test3);
104    mu_assert(result == NULL, "Should delete.");
105
106    return NULL;
107 }
108
109 char *all_tests()
110 {
111     mu_suite_start();
112
113     mu_run_test(test_create);
114     mu_run_test(test_get_set);
115     mu_run_test(test_traverse);
116     mu_run_test(test_delete);
117     mu_run_test(test_destroy);
118
119     return NULL;
120 }
121
122 RUN_TESTS(all_tests);

```

The only thing to learn about this unit test is that at the top I use a feature of bstring to create static strings to work with in the tests. I use the tagbstring and bsStatic to create them on lines 7–13.

38.1 How To Improve It

This is a very simple implementation of Hashmap as are most of the other data structures in this book. My goal isn't to give you insanely great hyper speed well tuned data structures. Usually those are much too complicated to discuss and only distract you from the real basic data structure at work. My goal is to give you an understandable starting point to then improve it or understand how they are implemented.

In this case, there's some things you can do with this implementation:

1. You can use a sort on each bucket so that they are always sorted. This increases your insert time, but decreases your find time because you can then use a binary search to find each node. Right now it's looping through all of the nodes in a bucket just to find one.
2. You can dynamically size the number of buckets, or let the caller specify the number for each Hashmap created.
3. You can use a better default_hash. There are tons of them.

4. This (and nearly every Hashmap is vulnerable to someone picking keys that will fill only one bucket, and then tricking your program into processing them. This then makes your program run slower because it changes from processing a Hashmap to effectively processing a single DArray. If you sort the nodes in the bucket this helps, but you can also use better hashing functions, and for the really paranoid add a random salt so that keys can't be predicted.
5. You could have it delete buckets that are empty of nodes to save space, or put empty buckets into a cache so you save on creating and destroying them.
6. Right now it just adds elements even if they already exist. Write an alternative set method that only adds it if it isn't set already.

As usual you should go through each function and make it bullet proof. The Hashmap could also use a debug setting for doing an invariant check.

38.2 Extra Credit

1. Research the Hashmap implementation of your favorite programming language to see what features they have.
2. Find out what the major disadvantages of a Hashmap are and how to avoid them. For example, they do not preserve order without special changes and they don't work when you need to find things based on parts of keys.
3. Write a unit test that demonstrates the defect of filling a Hashmap with keys that land in the same bucket, then test how this impact performance. A good way to do this is to just reduce the number of buckets to something stupid like 5.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 39

Exercise 38: Hashmap Algorithms

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

There are three hash functions that you'll implement in this exercise:

FNV-1a

Named after the creators Glenn Fowler, Phong Vo, and Landon Curt Noll. This hash produces good numbers and is reasonably fast.

Adler-32

Named after Mark Adler, is a horrible hash algorithm, but it's been around a long time and it's good for studying.

DJB Hash

This hash algorithm is attributed to Dan J. Bernstein (DJB) but it's difficult to find his discussion of the algorithm. It's shown to be fast, but possibly not great numbers.

You've already seen the Jenkins hash as the default hash for the Hashmap data structure, so this exercise will be looking at these three new ones. The code for them is usually small, and it's not optimized at all. As usual I'm going for understanding and not blinding latest speed.

The header file is very simple, so I'll start with that:

Source 119: src/lcthw/hashmap_algos.h

```
1 #ifndef hashmap_algos_h
2 #define hashmap_algos_h
3
4 #include <stdint.h>
5
6 uint32_t Hashmap_fnv1a_hash(void *data);
7
8 uint32_t Hashmap_adler32_hash(void *data);
9
10 uint32_t Hashmap_djb_hash(void *data);
11
12 #endif
```

I'm just declaring the three functions I'll implement in the hashmap_algos.c file:

Source 120: src/lcthw/hashmap_algos.c

```
1 #include <lcthw/hashmap_algos.h>
2 #include <lcthw/bstrlib.h>
3
4 // settings taken from
5 // http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-param
6 const uint32_t FNV_PRIME = 16777619;
7 const uint32_t FNV_OFFSET_BASIS = 2166136261;
8
```

```

9  uint32_t Hashmap_fnv1a_hash(void *data)
10 {
11     bstring s = (bstring)data;
12     uint32_t hash = FNV_OFFSET_BASIS;
13     int i = 0;
14
15     for(i = 0; i < blength(s); i++) {
16         hash ^= bchare(s, i, 0);
17         hash *= FNV_PRIME;
18     }
19
20     return hash;
21 }
22
23 const int MOD_ADLER = 65521;
24
25 uint32_t Hashmap_adler32_hash(void *data)
26 {
27     bstring s = (bstring)data;
28     uint32_t a = 1, b = 0;
29     int i = 0;
30
31     for (i = 0; i < blength(s); i++)
32     {
33         a = (a + bchare(s, i, 0)) % MOD_ADLER;
34         b = (b + a) % MOD_ADLER;
35     }
36
37     return (b << 16) | a;
38 }
39
40 uint32_t Hashmap_djb_hash(void *data)
41 {
42     bstring s = (bstring)data;
43     uint32_t hash = 5381;
44     int i = 0;
45
46     for(i = 0; i < blength(s); i++) {
47         hash = ((hash << 5) + hash) + bchare(s, i, 0); /* hash * 33 + c */
48     }
49
50     return hash;
51 }

```

This file then has the three hash algorithms. You should notice that I'm defaulting to just using a bstring for the key, and I'm using the bchare function to get a character from the bstring, but return 0 if that character is outside the string's length.

Each of these algorithms are found online so go search for them and read about them. Again I used Wikipedia primarily and then followed it to other sources.

I then have a unit test that tests out each algorithm, but also tests that it will distribute well across a number of buckets:

Source 121: tests/hashmap_algos_tests.c

```

1  #include <lcthw/bstrlib.h>
2  #include <lcthw/hashmap.h>
3  #include <lcthw/hashmap_algos.h>
4  #include <lcthw/darray.h>
5  #include "minunit.h"
6
7  struct tagbstring test1 = bsStatic("test data 1");
8  struct tagbstring test2 = bsStatic("test data 2");
9  struct tagbstring test3 = bsStatic("xest data 3");
10
11 char *test_fnv1a()
12 {
13     uint32_t hash = Hashmap_fnv1a_hash(&test1);
14     mu_assert(hash != 0, "Bad hash.");
15
16     hash = Hashmap_fnv1a_hash(&test2);
17     mu_assert(hash != 0, "Bad hash.");
18

```

```

19  hash = Hashmap_fnv1a_hash(&test3);
20  mu_assert(hash != 0, "Bad hash.");
21
22  return NULL;
23 }
24
25 char *test_adler32()
26 {
27     uint32_t hash = Hashmap_adler32_hash(&test1);
28     mu_assert(hash != 0, "Bad hash.");
29
30     hash = Hashmap_adler32_hash(&test2);
31     mu_assert(hash != 0, "Bad hash.");
32
33     hash = Hashmap_adler32_hash(&test3);
34     mu_assert(hash != 0, "Bad hash.");
35
36     return NULL;
37 }
38
39 char *test_djb()
40 {
41     uint32_t hash = Hashmap_djb_hash(&test1);
42     mu_assert(hash != 0, "Bad hash.");
43
44     hash = Hashmap_djb_hash(&test2);
45     mu_assert(hash != 0, "Bad hash.");
46
47     hash = Hashmap_djb_hash(&test3);
48     mu_assert(hash != 0, "Bad hash.");
49
50     return NULL;
51 }
52
53 #define BUCKETS 100
54 #define BUFFER_LEN 20
55 #define NUM_KEYS BUCKETS * 1000
56 enum { ALGO_FNV1A, ALGO_ADLER32, ALGO_DJB};
57
58 int gen_keys(DArray *keys, int num_keys)
59 {
60     int i = 0;
61     FILE *urand = fopen("/dev/urandom", "r");
62     check(urand != NULL, "Failed to open /dev/urandom");
63
64     struct bStream *stream = bsopen((bNread)fread, urand);
65     check(stream != NULL, "Failed to open /dev/urandom");
66
67     bstring key = bfromcstr("");
68     int rc = 0;
69
70     // FNV1a histogram
71     for(i = 0; i < num_keys; i++) {
72         rc = bsread(key, stream, BUFFER_LEN);
73         check(rc >= 0, "Failed to read from /dev/urandom.");
74
75         DArray_push(keys, bstrcpy(key));
76     }
77
78     bsclose(stream);
79     fclose(urand);
80     return 0;
81
82 error:
83     return -1;
84 }
85
86 void destroy_keys(DArray *keys)
87 {
88     int i = 0;
89     for(i = 0; i < NUM_KEYS; i++) {
90         bdestroy(DArray_get(keys, i));
91     }
92
93     DArray_destroy(keys);
94 }
95
96 void fill_distribution(int *stats, DArray *keys, Hashmap_hash hash_func)
97 {
98     int i = 0;
99     uint32_t hash = 0;
100

```

```

101 for(i = 0; i < DArray_count(keys); i++) {
102     hash = hash_func(DArray_get(keys, i));
103     stats[hash % BUCKETS] += 1;
104 }
105
106 }
107
108 char *test_distribution()
109 {
110     int i = 0;
111     int stats[3][BUCKETS] = {{0}};
112     DArray *keys = DArray_create(0, NUM_KEYS);
113
114     mu_assert(gen_keys(keys, NUM_KEYS) == 0, "Failed to generate random keys.");
115
116     fill_distribution(stats[ALGO_FNV1A], keys, Hashmap_fnv1a_hash);
117     fill_distribution(stats[ALGO_ADLER32], keys, Hashmap_adler32_hash);
118     fill_distribution(stats[ALGO_DJB], keys, Hashmap_djb_hash);
119
120     fprintf(stderr, "FNV\tA32\tDJB\n");
121
122     for(i = 0; i < BUCKETS; i++) {
123         fprintf(stderr, "%d\t%d\t%d\n",
124             stats[ALGO_FNV1A][i],
125             stats[ALGO_ADLER32][i],
126             stats[ALGO_DJB][i]);
127     }
128
129     destroy_keys(keys);
130
131     return NULL;
132 }
133
134 char *all_tests()
135 {
136     mu_suite_start();
137
138     mu_run_test(test_fnv1a);
139     mu_run_test(test_adler32);
140     mu_run_test(test_djb);
141     mu_run_test(test_distribution);
142
143     return NULL;
144 }
145
146 RUN_TESTS(all_tests);

```

I have the number of BUCKETS in this code set fairly high, since I have a fast enough computer, but if it runs slow just lower them, and also lower NUM_KEYS. What this test lets me do is run the test and then look at the distribution of keys for each hash function using a bit of analysis with a language called R.

How I do this is I craft a big list of keys using the gen_keys function. These keys are taken out of the /dev/urandom device they are random byte keys. I then use these keys to have the fill_distribution function fill up the stats array with where those keys would hash in a theoretical set of buckets. All this function does is go through all the keys, do the hash, then do what the Hashmap would do to find its bucket.

Finally I'm simply printing out a three column table of the final count for each bucket, showing how many keys managed to get into each bucket randomly. I can then look at these numbers to see if the hash functions are distributing keys mostly evenly.

39.1 What You Should See

Teaching you R is outside the scope of this book, but if you want to get it and try this then it can be found at r-project.org.

Here is an abbreviated shell session showing me run the tests/hashmap_algos_test to get the table produced by test_distribution (not shown here), and then use R to see what the summary statistics are:

Source 122: R Analysis of hashmap_algos_tests.c

```

1 $ tests/hashmap_algos_tests
2 # copy-paste the table it prints out
3 $ vim hash.txt
4 $ R
5 > hash <- read.table("hash.txt", header=T)
6 > summary(hash)
7   FNV      A32      DJB
8   Min. : 945   Min. : 908.0   Min. : 927
9   1st Qu.: 980   1st Qu.: 980.8   1st Qu.: 979
10  Median : 998   Median :1000.0   Median : 998
11  Mean   :1000   Mean   :1000.0   Mean   :1000
12  3rd Qu.:1016   3rd Qu.:1019.2   3rd Qu.:1021
13  Max.   :1072   Max.   :1075.0   Max.   :1082
14 >

```

First I just run the test, which on your screen will print the table. Then I just copy-paste it out of my terminal and use vim hash.txt to save the data. If you look at the data it has the header FNV A32 DJB for each of the three algorithms.

Second I run R and load the data using the read.table command. This is a smart function that works with this kind of tab-delimited data and I only have to tell it header=T so it knows the data has a header.

Finally, I have the data loaded and I can use summary to print out its summary statistics for each column. Here you can see that each function actually does alright with this random data. I'll explain what each of these rows means:

Min.

This is the minimum value found for the data in that column. FNV seems to win this on this run since it has the largest number, meaning it has a tighter range at the low end.

1st Qu.

The point where the first quarter of the data ends.

Median

This is the number that is in the middle if you sorted them. Median is most useful when compared to mean.

Mean

Mean is the "average" most people think of, and is the sum/count of the data. If you look, all of them are 1000, which is great. If you compare this to the median you see that all three have really close medians to the mean. What this means is the data isn't "skewed" in one direction, so you can trust the mean.

3rd Qu.

The point where the last quarter of the data starts and represents the tail end of the numbers.

Max.

This is the maximum number of the data, and presents the upper bound on all of them.

Looking at this data, you see that all of these hashes seem to do good on random keys, and that the means match the NUM_KEYS setting I made. What I'm looking for is that if I make 1000 keys per buckets (BUCKETS * 1000), then on average each bucket should have 1000 keys in it. If the hash function isn't working then you'll see these summary statistics show a Mean that's not 1000, and really high ranges at the 1st quarter and 3rd quarter. A good hash function should have a dead on 1000 mean, and as tight as possible range.

You should also know that you will get mostly different numbers from mine, and even between different runs of this unit test.

39.2 How To Break It

I'm finally going to have you do some breaking in this exercise. I want you to write the worst hash function you can, and then use the data to prove that it's really bad. You can use R to do the stats, just like I did, but maybe you have another

tool you can use to give you the same summary statistics.

The goal is to make a hash function that seems normal to an untrained eye, but when actually run has a bad mean and is all over the place. That means you can't just have it return 1, but have to give a stream of numbers that seem alright, but really are all over the place and loading up some buckets too much.

Extra points if you can make a minimal change to one of the four hash algorithms I gave you to do this.

The purpose of this exercise is to imagine that some "friendly" coder comes to you and offers to improve your hash function, but actually just makes a nice little backdoor that screws up your Hashmap.

As the Royal Society says, "Nullius in verba."

39.3 Extra Credit

1. Take the default_hash out of the hashmap.c, make it one of the algorithms in hashmap_algos.c and then make all the tests work again.
2. Add the default_hash to the hashmap_algos_tests.c test and compare its statistics to the other hash functions.
3. Find a few more hash functions and add them too. You can never have too many hash functions!

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 40

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 39: String Algorithms

In this exercise I'm going to show you one of the supposedly faster string search algorithms, and compare it to the one that exists in `bstrlib.c` call `binstr`. The documentation for `binstr` says that it uses a simple "brute force" string search to find the first instance. The one I'll implement will use the Boyer–Moore–Horspool (BMH) algorithm, which is supposed to be faster if you analyze the theoretical time. You'll see that, assuming my implementation isn't flawed, that the practical time for BMH is much worse than the simple brute force of `binstr`.

The point of this exercise isn't really to explain the algorithm because it's simple enough for you to go to [the Boyer–Moore–Horspool Wikipedia page](#) and read it. The gist of this algorithm is that it calculates a "skip characters list" as a first operation, then it uses this list to quickly scan through the string. It is supposed to be faster than brute force, so let's get the code into the right files and see.

First, I have the header:

Source 123: `src/lcthw/string_algos.h`

```

1  #ifndef string_algos_h
2  #define string_algos_h
3
4  #include <lcthw/bstrlib.h>
5  #include <lcthw/darray.h>
6
7  typedef struct StringScanner {
8      bstring in;
9      const unsigned char *haystack;
10     ssize_t hlen;
11     const unsigned char *needle;
12     ssize_t nlen;
13     size_t skip_chars[UCHAR_MAX + 1];
14 } StringScanner;
15
16 int String_find(bstring in, bstring what);
17
18 StringScanner *StringScanner_create(bstring in);
19
20 int StringScanner_scan(StringScanner *scan, bstring tofind);
21
22 void StringScanner_destroy(StringScanner *scan);
23
24 #endif

```

In order to see the effects of this "skip characters list" I'm going to make two versions of the BMH algorithm:

String_find

Simply find the first instance of one string in another, doing the entire algorithm in one shot.

StringScanner_scan

Uses a `StringScanner` state structure to separate the skip list build from the actual find. This will let me see what

impact that has on performance. This model also has the advantage that I can incrementally scan for one string in another and find all instances quickly.

Once you have that, here's the implementation:

Source 124: src/lcthw/string_algos.c

```

1  #include <lcthw/string_algos.h>
2  #include <limits.h>
3
4  static inline void String_setup_skip_chars(
5      size_t *skip_chars,
6      const unsigned char *needle, ssize_t nlen)
7  {
8      size_t i = 0;
9      size_t last = nlen - 1;
10
11     for(i = 0; i < UCHAR_MAX + 1; i++) {
12         skip_chars[i] = nlen;
13     }
14
15     for (i = 0; i < last; i++) {
16         skip_chars[needle[i]] = last - i;
17     }
18 }
19
20
21 static inline const unsigned char *String_base_search(
22     const unsigned char *haystack, ssize_t hlen,
23     const unsigned char *needle, ssize_t nlen,
24     size_t *skip_chars)
25 {
26     size_t i = 0;
27     size_t last = nlen - 1;
28
29     assert(haystack != NULL && "Given bad haystack to search.");
30     assert(needle != NULL && "Given bad needle to search for.");
31
32     check(nlen > 0, "nlen can't be <= 0");
33     check(hlen > 0, "hlen can't be <= 0");
34
35     while (hlen >= nlen)
36     {
37         for (i = last; haystack[i] == needle[i]; i--) {
38             if (i == 0) {
39                 return haystack;
40             }
41         }
42
43         hlen -= skip_chars[haystack[last]];
44         haystack += skip_chars[haystack[last]];
45     }
46
47     error: // fallthrough
48     return NULL;
49 }
50
51 int String_find(bstring in, bstring what)
52 {
53     const unsigned char *found = NULL;
54
55     const unsigned char *haystack = (const unsigned char *)bdata(in);
56     ssize_t hlen = blength(in);
57     const unsigned char *needle = (const unsigned char *)bdata(what);
58     ssize_t nlen = blength(what);
59     size_t skip_chars[UCHAR_MAX + 1] = {0};
60
61     String_setup_skip_chars(skip_chars, needle, nlen);
62
63     found = String_base_search(haystack, hlen, needle, nlen, skip_chars);
64
65     return found != NULL ? found - haystack : -1;
66 }
67
68 StringScanner *StringScanner_create(bstring in)
69 {
70     StringScanner *scan = calloc(1, sizeof(StringScanner));
71     check_mem(scan);

```

```

72
73     scan->in = in;
74     scan->haystack = (const unsigned char *)bdata(in);
75     scan->hlen = blength(in);
76
77     assert(scan != NULL && "fuck");
78     return scan;
79
80 error:
81     free(scan);
82     return NULL;
83 }
84
85 static inline void StringScanner_set_needle(StringScanner *scan, bstring tofind)
86 {
87     scan->needle = (const unsigned char *)bdata(tofind);
88     scan->nlen = blength(tofind);
89
90     String_setup_skip_chars(scan->skip_chars, scan->needle, scan->nlen);
91 }
92
93 static inline void StringScanner_reset(StringScanner *scan)
94 {
95     scan->haystack = (const unsigned char *)bdata(scan->in);
96     scan->hlen = blength(scan->in);
97 }
98
99 int StringScanner_scan(StringScanner *scan, bstring tofind)
100 {
101     const unsigned char *found = NULL;
102     ssize_t found_at = 0;
103
104     if(scan->hlen <= 0) {
105         StringScanner_reset(scan);
106         return -1;
107     }
108
109     if((const unsigned char *)bdata(tofind) != scan->needle) {
110         StringScanner_set_needle(scan, tofind);
111     }
112
113     found = String_base_search(
114         scan->haystack, scan->hlen,
115         scan->needle, scan->nlen,
116         scan->skip_chars);
117
118     if(found) {
119         found_at = found - (const unsigned char *)bdata(scan->in);
120         scan->haystack = found + scan->nlen;
121         scan->hlen -= found_at - scan->nlen;
122     } else {
123         // done, reset the setup
124         StringScanner_reset(scan);
125         found_at = -1;
126     }
127
128     return found_at;
129 }
130
131
132 void StringScanner_destroy(StringScanner *scan)
133 {
134     if(scan) {
135         free(scan);
136     }
137 }

```

The entire algorithm is in two static inline functions called `String_setup_skip_chars` and `String_base_search`. These are then used in the other functions to actually implement the searching styles I want. Study these first two functions and compare them to the Wikipedia description so you know what's going on.

The `String_find` then just uses these two functions to do a find and return the position found. It's very simple and I'll use it to see how this "build skip chars" phase impacts real practical performance. Keep in mind that you could maybe make this faster, but I'm teaching you how to confirm theoretical speed after you implement an algorithm.

The `StringScanner_scan` function is then following the common pattern I use of "create, scan, destroy" and is used to incrementally scan a string for another string. You'll see how this is used when I show you the unit test that will test this out.

Finally, I have the unit test that first confirms this is all working, then runs simple performance tests for all three finding algorithms in a commented out section.

Source 125: tests/string_algos_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/string_algos.h>
3  #include <lcthw/bstrlib.h>
4  #include <time.h>
5
6  struct tagbstring IN_STR = bsStatic("I have ALPHA beta ALPHA and oranges ALPHA");
7  struct tagbstring ALPHA = bsStatic("ALPHA");
8  const int TEST_TIME = 1;
9
10 char *test_find_and_scan()
11 {
12     StringScanner *scan = StringScanner_create(&IN_STR);
13     mu_assert(scan != NULL, "Failed to make the scanner.");
14
15     int find_i = String_find(&IN_STR, &ALPHA);
16     mu_assert(find_i > 0, "Failed to find 'ALPHA' in test string.");
17
18     int scan_i = StringScanner_scan(scan, &ALPHA);
19     mu_assert(scan_i > 0, "Failed to find 'ALPHA' with scan.");
20     mu_assert(scan_i == find_i, "find and scan don't match");
21
22     scan_i = StringScanner_scan(scan, &ALPHA);
23     mu_assert(scan_i > find_i, "should find another ALPHA after the first");
24
25     scan_i = StringScanner_scan(scan, &ALPHA);
26     mu_assert(scan_i > find_i, "should find another ALPHA after the first");
27
28     mu_assert(StringScanner_scan(scan, &ALPHA) == -1, "shouldn't find it");
29
30     StringScanner_destroy(scan);
31
32     return NULL;
33 }
34
35 char *test_binstr_performance()
36 {
37     int i = 0;
38     int found_at = 0;
39     unsigned long find_count = 0;
40     time_t elapsed = 0;
41     time_t start = time(NULL);
42
43     do {
44         for(i = 0; i < 1000; i++) {
45             found_at = binstr(&IN_STR, 0, &ALPHA);
46             mu_assert(found_at != BSTR_ERR, "Failed to find!");
47             find_count++;
48         }
49
50         elapsed = time(NULL) - start;
51     } while(elapsed <= TEST_TIME);
52
53     debug("BINSTR COUNT: %lu, END TIME: %d, OPS: %f",
54         find_count, (int)elapsed, (double)find_count / elapsed);
55     return NULL;
56 }
57
58 char *test_find_performance()
59 {
60     int i = 0;
61     int found_at = 0;
62     unsigned long find_count = 0;
63     time_t elapsed = 0;
64     time_t start = time(NULL);
65
66     do {
67         for(i = 0; i < 1000; i++) {
68             found_at = String_find(&IN_STR, &ALPHA);

```

```

69     find_count++;
70 }
71
72     elapsed = time(NULL) - start;
73 } while(elapsed <= TEST_TIME);
74
75     debug("FIND COUNT: %lu, END TIME: %d, OPS: %f",
76         find_count, (int)elapsed, (double)find_count / elapsed);
77
78     return NULL;
79 }
80
81 char *test_scan_performance()
82 {
83     int i = 0;
84     int found_at = 0;
85     unsigned long find_count = 0;
86     time_t elapsed = 0;
87     StringScanner *scan = StringScanner_create(&IN_STR);
88
89     time_t start = time(NULL);
90
91     do {
92         for(i = 0; i < 1000; i++) {
93             found_at = 0;
94
95             do {
96                 found_at = StringScanner_scan(scan, &ALPHA);
97                 find_count++;
98             } while(found_at != -1);
99         }
100
101         elapsed = time(NULL) - start;
102     } while(elapsed <= TEST_TIME);
103
104     debug("SCAN COUNT: %lu, END TIME: %d, OPS: %f",
105         find_count, (int)elapsed, (double)find_count / elapsed);
106
107     StringScanner_destroy(scan);
108
109     return NULL;
110 }
111
112
113 char *all_tests()
114 {
115     mu_suite_start();
116
117     mu_run_test(test_find_and_scan);
118
119     // this is an idiom for commenting out sections of code
120     #if 0
121     mu_run_test(test_scan_performance);
122     mu_run_test(test_find_performance);
123     mu_run_test(test_binstr_performance);
124     #endif
125
126     return NULL;
127 }
128
129 RUN_TESTS(all_tests);

```

I have it written here with `#if 0` which is a way to use the C++ to comment out a section of code. Type it in like this, and then remove that and the `#endif` so you can see these performance tests run. When you continue with the book, simply comment these out so that the test doesn't waste development time.

There's nothing amazing in this unit test, it just runs each of the different functions in loops that last long enough to get a few seconds of sampling. The first test (`test_find_and_scan`) just confirms that what I've written works, because there's no point in testing the speed of something that doesn't work. Then the next three functions run a large number of searches using each of the three functions.

The trick to notice is that I grab the starting time in `start`, and then I loop until at least `TEST_TIME` seconds have passed. This makes sure that I get enough samples to work with in comparing the three. I'll then run this test with

different TEST_TIME settings and analyze the results.

40.1 What You Should See

When I run this test on my laptop, I get number that look like this:

Source 126: 2 Second Test Run

```
1  $ ./tests/string_algos_tests
2  DEBUG tests/string_algos_tests.c:124: ----- RUNNING: ./tests/string_algos_tests
3  -----
4  RUNNING: ./tests/string_algos_tests
5  DEBUG tests/string_algos_tests.c:116:
6  ----- test_find_and_scan
7  DEBUG tests/string_algos_tests.c:117:
8  ----- test_scan_performance
9  DEBUG tests/string_algos_tests.c:105: SCAN COUNT: 110272000, END TIME: 2, OPS: 55136000.000000
10 DEBUG tests/string_algos_tests.c:118:
11 ----- test_find_performance
12 DEBUG tests/string_algos_tests.c:76: FIND COUNT: 12710000, END TIME: 2, OPS: 6355000.000000
13 DEBUG tests/string_algos_tests.c:119:
14 ----- test_binstr_performance
15 DEBUG tests/string_algos_tests.c:54: BINSTR COUNT: 72736000, END TIME: 2, OPS: 36368000.000000
16 ALL TESTS PASSED
17 Tests run: 4
18 $
```

I look at this and I sort of want to do more than 2 seconds of each run, and I want to run this many times then use R to check it out like I did before. Here's what I get for 10 samples of about 10 seconds each:

Source 127: 10 Runs At 10 Seconds, Operations / Second

```
scan find binstr
71195200 6353700 37110200
75098000 6358400 37420800
74910000 6351300 37263600
74859600 6586100 37133200
73345600 6365200 37549700
74754400 6358000 37162400
75343600 6630400 37075000
73804800 6439900 36858700
74995200 6384300 36811700
74781200 6449500 37383000
```

The way I got this is with a little bit of shell help and then editing the output:

Source 128: Getting Timing Logs

```
1  $ for i in 1 2 3 4 5 6 7 8 9 10; do echo "RUN --
- $i" >> times.log; ./tests/string_algos_tests 2>&1 | grep COUNT >> times.log ; done
2  $ less times.log
3  $ vim times.log
```

Right away you can see that the scanning system beats the pants off both of the others, but I'll open this in R and confirm the results:

Source 129: R Summary Of Operations/Second

```
1  > times <- read.table("times.log", header=T)
2  > summary(times)
3  scan      find      binstr
4  Min. :71195200 Min. :6351300 Min. :36811700
5  1st Qu.:74042200 1st Qu.:6358100 1st Qu.:37083800
6  Median :74820400 Median :6374750  Median :37147800
7
```



```

      Mean :74308760 Mean :6427680 Mean :37176830
8  3rd Qu.:74973900 3rd Qu.:6447100 3rd Qu.:37353150
9  Max. :75343600 Max. :6630400 Max. :37549700
10 >

```

To understand why I'm getting the summary statistics I have to explain some statistics for you. What I'm looking for in these numbers can be said simply to be, "Are these three functions (scan, find, bsintr) actually different?" I know that each time I run my tester function I get slightly different numbers, and that those numbers can cover a certain range. You see here that the 1st and 3rd quarters do that for each sample.

What I look at first is the mean and I want to see if each sample's mean is different from the others. I can see that, and clearly the scan beats binstr which also beats find. However, I have a problem, if I use just the mean, there's a chance that the ranges of each sample might overlap.

What if I have means that are different, but the 1st and 3rd quarters overlap? In that case I could say that there's a chance that if I ran the samples again the means might not be different. The more overlap I have in the ranges the higher probability that my two samples (and my two functions) are not actually different. Any difference I'm seeing in the two (in this case three) is just random chance.

Statistics has many tools to solve this problem, but in our case I can just look at the 1st and 3rd quarters as well as the mean for all three samples. If the means are different and the quarters are way off never possibly overlapping, then it's alright to say they are different.

In my three samples I can say that scan, find and binstr are different, don't overlap in range, and that I can trust the sample (for the most part).

40.2 Analyzing The Results

Looking at the results I can see that String_find is much slower than the other two. In fact, so slow I'd think there's something wrong with how I implemented it. However when I compare it with StringScanner_scan I can see that it's the part that builds the skip list that is most likely costing the time. Not only is find slower, it's also doing less than scan because it's just finding the first string while scan finds all of them.

I can also see that scan beats binstr as well by quite a large margin. Again I can say that not only does scan do more than both of these, but it's also much faster.

There's a few caveats with this analysis:

1. I may have messed up this implementation or the test. At this point I would go research all the possible ways to do a BMH algorithm and try to improve it. I would also confirm that I'm doing the test right.
2. If you alter the time the test runs, you get different results. There is a "warm up" period I'm not investigating.
3. The test_scan_performance unit test isn't quite the same as the others, but it is doing more than the other tests so it's probably alright.
4. I'm only doing the test by searching for one string in another. I could randomize the strings to find to remove their position and length as a confounding factor.
5. Maybe binstr is implemented better than "simple" brute force.
6. I could be running these in an unfortunate order and maybe randomizing which test runs first will give better results.

One thing to gather from this is you need to confirm real performance even if you implement an algorithm "correctly". In this case the claim is that the BMH algorithm should have beaten the binstr algorithm, but a simple test proved it didn't. Had I not done this I would have been using an inferior algorithm implementation without knowing it. With these metrics I can start to tune my implementation, or simply scrap it and find another one.

40.3 Extra Credit

1. See if you can make the `Scan_find` faster. Why is my implementation here slow?
2. Try some different scan times and see if you get different numbers. What impact does the length of time that you run the test have on the scan times? What can you say about that result?
3. Alter the unit test so that it runs each function for a short burst in the beginning to clear out any "warm up" period, then start the timing portion. Does that change the dependence on the length of time the test runs and how many operations / second are possible?
4. Make the unit test randomize the strings to find and then measure the performance you get. One way to do this is use the `bsplit` function from `bstrlib.h` to split the `IN_STR` on spaces. Then use the `bstrList` struct you get to access each string it returns. This will also teach you how to use `bstrList` operations for string processing.
5. Try some runs with the tests in different orders and see if you get different results.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 41

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 40: Binary Search Trees

The binary tree is the simplest tree based data structure and while it has been replaced by Hash Maps in many languages is still useful for many applications. Variants on the binary tree exist for very useful things like database indexes, search algorithm structures, and even graphics processing.

I'm calling my binary tree a BSTree for "binary search tree" and the best way to describe it is that it's another way to do a Hashmap style key/value store. The difference is that instead of hashing the key to find a location, the BSTree compares the key to nodes in a tree, and then walks the tree to find the best place to store it based on how it compares to other nodes.

Before I really explain how this works, let me show you the bstree.h header file so you can see the data structures, then I can use that to explain how it's built.

Source 130: src/lcthw/bstree.h

```

1  #ifndef _lcthw_BSTree_h
2  #define _lcthw_BSTree_h
3
4
5  typedef int (*BSTree_compare)(void *a, void *b);
6
7  typedef struct BSTreeNode {
8      void *key;
9      void *data;
10
11      struct BSTreeNode *left;
12      struct BSTreeNode *right;
13      struct BSTreeNode *parent;
14  } BSTreeNode;
15
16  typedef struct BSTree {
17      int count;
18      BSTree_compare compare;
19      BSTreeNode *root;
20  } BSTree;
21
22  typedef int (*BSTree_traverse_cb)(BSTreeNode *node);
23
24  BSTree *BSTree_create(BSTree_compare compare);
25  void BSTree_destroy(BSTree *map);
26
27  int BSTree_set(BSTree *map, void *key, void *data);
28  void *BSTree_get(BSTree *map, void *key);
29
30  int BSTree_traverse(BSTree *map, BSTree_traverse_cb traverse_cb);
31
32  void *BSTree_delete(BSTree *map, void *key);
33
34  #endif

```

This follows the same pattern I've been using this whole time where I have a base "container" named `BSTree` and that then has nodes names `BSTreeNode` that make up the actual contents. Bored yet? Good, there's no reason to be clever with this kind of structure.

The important part is how the `BSTreeNode` is configured and how it gets used to do each operation: set, get, and delete. I'll cover get first since it's the easiest operation and I'll pretend I'm doing it manually against the data structure:

1. I take the key you're looking for and I start at the root. First thing I do is compare your key with that node's key.
2. If your key is less-than the node.key, then I traverse down the tree using the left pointer.
3. If your key is greater-than the node.key, then I go down with right.
4. I repeat step 2 and 3 until either I find a matching node.key, or I get to a node that has no left and right. In the first case I return the node.data, in the second I return NULL.

That's all there is to get, so now to do set it's nearly the same thing, except you're looking for where to put a new node:

1. If there is no `BSTree.root` then I just make that and we're done. That's the first node.
2. After that I compare your key to node.key, starting at the root.
3. If your key is less-than or equal to the node.key then I want to go left. If your key is greater-than (not equal) then I want to go right.
4. I keep repeating 3 until I reach a node where the left or right doesn't exist, but that's the direction I need to go.
5. Once there I set that direction (left or right) to a new node for the key and data I want, and set this new node's parent to the previous node I came from. I'll use the parent node when I do delete.

This also makes sense given how get works. If finding a node involves going left or right depending on how they key compares, well then setting a node involves the same thing until I can set the left or right for a new node.

Take some time to draw out a few trees on paper and go through some setting and getting nodes so you understand how it work. After that you are ready to look at the implementation so that I can explain delete. Deleting in trees is a major pain, and so it's best explained by doing a line-by-line code breakdown.

Source 131: `src/lcthw/bstree.c`

```

1  #include <lcthw/dbg.h>
2  #include <lcthw/bstree.h>
3  #include <stdlib.h>
4  #include <lcthw/bstrlib.h>
5
6  static int default_compare(void *a, void *b)
7  {
8      return bstrcmp((bstring)a, (bstring)b);
9  }
10
11
12 BSTree *BSTree_create(BSTree_compare compare)
13 {
14     BSTree *map = calloc(1, sizeof(BSTree));
15     check_mem(map);
16
17     map->compare = compare == NULL ? default_compare : compare;
18
19     return map;
20
21 error:
22     if(map) {
23         BSTree_destroy(map);
24     }
25     return NULL;
26 }
27
28 static int BSTree_destroy_cb(BSTreeNode *node)
29 {
30     free(node);
31     return 0;

```

```

32 }
33
34 void BSTree_destroy(BSTree *map)
35 {
36     if(map) {
37         BSTree_traverse(map, BSTree_destroy_cb);
38         free(map);
39     }
40 }
41
42
43 static inline BSTreeNode *BSTreeNode_create(BSTreeNode *parent, void *key, void *data)
44 {
45     BSTreeNode *node = calloc(1, sizeof(BSTreeNode));
46     check_mem(node);
47
48     node->key = key;
49     node->data = data;
50     node->parent = parent;
51     return node;
52
53 error:
54     return NULL;
55 }
56
57
58 static inline void BSTree_setnode(BSTree *map, BSTreeNode *node, void *key, void *data)
59 {
60     int cmp = map->compare(node->key, key);
61
62     if(cmp <= 0) {
63         if(node->left) {
64             BSTree_setnode(map, node->left, key, data);
65         } else {
66             node->left = BSTreeNode_create(node, key, data);
67         }
68     } else {
69         if(node->right) {
70             BSTree_setnode(map, node->right, key, data);
71         } else {
72             node->right = BSTreeNode_create(node, key, data);
73         }
74     }
75 }
76
77
78 int BSTree_set(BSTree *map, void *key, void *data)
79 {
80     if(map->root == NULL) {
81         // first so just make it and get out
82         map->root = BSTreeNode_create(NULL, key, data);
83         check_mem(map->root);
84     } else {
85         BSTree_setnode(map, map->root, key, data);
86     }
87
88     return 0;
89 error:
90     return -1;
91 }
92
93 static inline BSTreeNode *BSTree_getnode(BSTree *map, BSTreeNode *node, void *key)
94 {
95     int cmp = map->compare(node->key, key);
96
97     if(cmp == 0) {
98         return node;
99     } else if(cmp < 0) {
100         if(node->left) {
101             return BSTree_getnode(map, node->left, key);
102         } else {
103             return NULL;
104         }
105     } else {
106         if(node->right) {
107             return BSTree_getnode(map, node->right, key);
108         } else {
109             return NULL;
110         }
111     }
112 }
113

```

```

114 void *BSTree_get(BSTree *map, void *key)
115 {
116     if(map->root == NULL) {
117         return NULL;
118     } else {
119         BSTreeNode *node = BSTree_getnode(map, map->root, key);
120         return node == NULL ? NULL : node->data;
121     }
122 }
123
124
125 static inline int BSTree_traverse_nodes(BSTreeNode *node, BSTree_traverse_cb traverse_cb)
126 {
127     int rc = 0;
128
129     if(node->left) {
130         rc = BSTree_traverse_nodes(node->left, traverse_cb);
131         if(rc != 0) return rc;
132     }
133
134     if(node->right) {
135         rc = BSTree_traverse_nodes(node->right, traverse_cb);
136         if(rc != 0) return rc;
137     }
138
139     return traverse_cb(node);
140 }
141
142 int BSTree_traverse(BSTree *map, BSTree_traverse_cb traverse_cb)
143 {
144     if(map->root) {
145         return BSTree_traverse_nodes(map->root, traverse_cb);
146     }
147
148     return 0;
149 }
150
151 static inline BSTreeNode *BSTree_find_min(BSTreeNode *node)
152 {
153     while(node->left) {
154         node = node->left;
155     }
156
157     return node;
158 }
159
160 static inline void BSTree_replace_node_in_parent(BSTree *map, BSTreeNode *node, BSTreeNode *new_value)
161 {
162     if(node->parent) {
163         if(node == node->parent->left) {
164             node->parent->left = new_value;
165         } else {
166             node->parent->right = new_value;
167         }
168     } else {
169         // this is the root so gotta change it
170         map->root = new_value;
171     }
172
173     if(new_value) {
174         new_value->parent = node->parent;
175     }
176 }
177
178 static inline void BSTree_swap(BSTreeNode *a, BSTreeNode *b)
179 {
180     void *temp = NULL;
181     temp = b->key; b->key = a->key; a->key = temp;
182     temp = b->data; b->data = a->data; a->data = temp;
183 }
184
185 static inline BSTreeNode *BSTree_node_delete(BSTree *map, BSTreeNode *node, void *key)
186 {
187     int cmp = map->compare(node->key, key);
188
189     if(cmp < 0) {
190         if(node->left) {
191             return BSTree_node_delete(map, node->left, key);
192         } else {
193             // not found
194             return NULL;
195         }
196     }

```

```

196 } else if(cmp > 0) {
197     if(node->right) {
198         return BSTree_node_delete(map, node->right, key);
199     } else {
200         // not found
201         return NULL;
202     }
203 } else {
204     if(node->left && node->right) {
205         // swap this node for the smallest node that is bigger than us
206         BSTreeNode *successor = BSTree_find_min(node->right);
207         BSTree_swap(successor, node);
208
209         // this leaves the old successor with possibly a right child
210         // so replace it with that right child
211         BSTree_replace_node_in_parent(map, successor, successor->right);
212
213         // finally it's swapped, so return successor instead of node
214         return successor;
215     } else if(node->left) {
216         BSTree_replace_node_in_parent(map, node, node->left);
217     } else if(node->right) {
218         BSTree_replace_node_in_parent(map, node, node->right);
219     } else {
220         BSTree_replace_node_in_parent(map, node, NULL);
221     }
222
223     return node;
224 }
225 }
226
227 void *BSTree_delete(BSTree *map, void *key)
228 {
229     void *data = NULL;
230
231     if(map->root) {
232         BSTreeNode *node = BSTree_node_delete(map, map->root, key);
233
234         if(node) {
235             data = node->data;
236             free(node);
237         }
238     }
239
240     return data;
241 }

```

Before getting into how `BSTree_delete` works I want to explain a pattern I'm using for doing recursive function calls in a sane way. You'll find that many tree based data structures are easy to write if you use recursion, but that formulating a single recursive function is difficult. Part of the problem is that you need to setup some initial data for the first operation, then recurse into the data structure, which is hard to do with one function.

The solution is to use two functions. One function "sets up" the data structure and initial recursion conditions so that a second function can do the real work. Take a look at `BSTree_get` first to see what I mean:

1. I have an initial condition to handle that if `map->root` is `NULL` then return `NULL` and don't recurse.
2. I then setup a call to the real recursion, which is in `BSTree_getnode`. I create the initial condition of the root node to start with, the key, and the map.
3. In the `BSTree_getnode` then I do the actual recursive logic. I compare the keys with `map->compare(node->key, key)` and go left, right, or equal depending on that.
4. Since this function is "self-similar" and doesn't have to handle any initial conditions (because `BSTree_get` did) then I can structure it very simply. When it's done it returns to the caller, and that return then comes back to `BSTree_get` the result.
5. At the end, the `BSTree_get` then handles getting the node.data element but only if the result isn't `NULL`.

This way of structuring a recursive algorithm matches the way I structure my recursive data structures. I have an initial "base function" that handles initial conditions and some edge cases, then it calls a clean recursive function that does the work. Compare that with how I have a "base struct" in `BStree` combined with recursive `BSTreeNode` structures that all

reference each other in a tree. Using this pattern makes it easy to deal with recursion and keep it straight.

Next, go look at `BSTree_set` and `BSTree_setnode` to see the exact same pattern going on. I use `BSTree_set` to configure the initial conditions and edge cases. A common edge case is that there's no root node, so I have to make one to get things started.

This pattern will work with nearly any recursive algorithm you have to figure out. The way I do this is I follow this pattern:

1. Figure out the initial variables, how they change, and what the stopping conditions are for each recursive step.
2. Write a recursive function that calls itself, with arguments for each stopping condition and initial variable.
3. Write a setup function to set initial starting conditions for the algorithm and handle edge cases, then it calls the recursive function.
4. Finally, the setup function returns the final result and possibly alters it if the recursive function can't handle final edge cases.

Which leads me finally to `BSTree_delete` and `BSTree_node_delete`. First you can just look at `BSTree_delete` and see that it's the setup function, and what it is doing is grabbing the resulting node data and freeing the node that's found. In `BSTree_node_delete` is where things get complex because to delete a node at any point in the tree, I have to rotate that node's children up to the parent. I'll break this function down and the ones it uses:

`bstree.c:190`

I run the compare function to figure out which direction I'm going.

`bstree.c:192-198`

This is the usual "less-than" branch where I want to go left. I'm handling the case that left doesn't exist here and returning NULL to say "not found". This handles deleting something that isn't in the BSTree.

`bstree.c:199-205`

The same thing but for the right branch of the tree. Just keep recursing down into the tree just like in the other functions, and return NULL if it doesn't exist.

`bstree.c:206`

This is where I have found the node since the key is equal (compare return 0).

`bstree.c:207`

This node has both a left and right branch, so it's deeply embedded in the tree.

`bstree.c:209`

To remove this node I need to first find the smallest node that's greater than this node, which means I call `BSTree_find_min` on the right child.

`bstree.c:210`

Once I have this node, I will do a swap on its key and data with the current node's. This will effectively take this node that was down at the bottom of the tree, and put it's contents here so that I don't have to try and shuffle this node out by its pointers.

`bstree.c:214`

The successor is now this dead branch that has the current node's values. It could just be removed, but there's a chance that it has a right node value, which means I need to do a single rotate so that the successor's right node gets moved up to completely detach it.

`bstree.c:217`

At this point, the successor is removed from the tree, its values replaced the current node's values, and any children it had are moved up into the parent. I can return the successor as if it were the node.

`bstree.c:218`

At this branch I know that the node has a left but no right, so I want to replace this node with its left child.

bstree.c:219

I again use `BSTree_replace_node_in_parent` to do the replace, rotating the left child up.

bstree.c:220

This branch of the if-statement means I have a right child but no left child, so I want to rotate the right child up.

bstree.c:221

Again, use the function to do the rotate, but this time of the right node.

bstree.c:222

Finally, the only thing that's left is the condition that I've found the node, and it has no children (no left or right). In this case, I simply replace this node with NULL using the same function I did with all the others.

bstree.c:210

After all that, I have the current node rotated out of the tree and replaced with some child element that will fit in the tree. I just return this to the caller so it can be freed and managed.

This operation is very complex, and to be honest, in some tree data structures I just don't bother doing deletes and treat them like constant data in my software. If I need to do heavy insert and delete, I use a Hashmap instead.

Finally, you can look at the unit test to see how I'm testing it:

Source 132: tests/bstree_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/bstree.h>
3  #include <assert.h>
4  #include <lcthw/bstrlib.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  BSTree *map = NULL;
9  static int traverse_called = 0;
10 struct tagbstring test1 = bsStatic("test data 1");
11 struct tagbstring test2 = bsStatic("test data 2");
12 struct tagbstring test3 = bsStatic("test data 3");
13 struct tagbstring expect1 = bsStatic("THE VALUE 1");
14 struct tagbstring expect2 = bsStatic("THE VALUE 2");
15 struct tagbstring expect3 = bsStatic("THE VALUE 3");
16
17 static int traverse_good_cb(BSTreeNode *node)
18 {
19     debug("KEY: %s", bdata((bstring)node->key));
20     traverse_called++;
21     return 0;
22 }
23
24
25 static int traverse_fail_cb(BSTreeNode *node)
26 {
27     debug("KEY: %s", bdata((bstring)node->key));
28     traverse_called++;
29
30     if(traverse_called == 2) {
31         return 1;
32     } else {
33         return 0;
34     }
35 }
36
37
38 char *test_create()
39 {
40     map = BSTree_create(NULL);
41     mu_assert(map != NULL, "Failed to create map.");
42
43     return NULL;
44 }
45
46 char *test_destroy()
47 {
48     BSTree_destroy(map);
49

```

```

50     return NULL;
51 }
52
53
54 char *test_get_set()
55 {
56     int rc = BSTree_set(map, &test1, &expect1);
57     mu_assert(rc == 0, "Failed to set &test1");
58     bstring result = BSTree_get(map, &test1);
59     mu_assert(result == &expect1, "Wrong value for test1.");
60
61     rc = BSTree_set(map, &test2, &expect2);
62     mu_assert(rc == 0, "Failed to set test2");
63     result = BSTree_get(map, &test2);
64     mu_assert(result == &expect2, "Wrong value for test2.");
65
66     rc = BSTree_set(map, &test3, &expect3);
67     mu_assert(rc == 0, "Failed to set test3");
68     result = BSTree_get(map, &test3);
69     mu_assert(result == &expect3, "Wrong value for test3.");
70
71     return NULL;
72 }
73
74 char *test_traverse()
75 {
76     int rc = BSTree_traverse(map, traverse_good_cb);
77     mu_assert(rc == 0, "Failed to traverse.");
78     mu_assert(traverse_called == 3, "Wrong count traverse.");
79
80     traverse_called = 0;
81     rc = BSTree_traverse(map, traverse_fail_cb);
82     mu_assert(rc == 1, "Failed to traverse.");
83     mu_assert(traverse_called == 2, "Wrong count traverse for fail.");
84
85     return NULL;
86 }
87
88 char *test_delete()
89 {
90     bstring deleted = (bstring)BSTree_delete(map, &test1);
91     mu_assert(deleted != NULL, "Got NULL on delete.");
92     mu_assert(deleted == &expect1, "Should get test1");
93     bstring result = BSTree_get(map, &test1);
94     mu_assert(result == NULL, "Should delete.");
95
96     deleted = (bstring)BSTree_delete(map, &test1);
97     mu_assert(deleted == NULL, "Should get NULL on delete");
98
99     deleted = (bstring)BSTree_delete(map, &test2);
100    mu_assert(deleted != NULL, "Got NULL on delete.");
101    mu_assert(deleted == &expect2, "Should get test2");
102    result = BSTree_get(map, &test2);
103    mu_assert(result == NULL, "Should delete.");
104
105    deleted = (bstring)BSTree_delete(map, &test3);
106    mu_assert(deleted != NULL, "Got NULL on delete.");
107    mu_assert(deleted == &expect3, "Should get test3");
108    result = BSTree_get(map, &test3);
109    mu_assert(result == NULL, "Should delete.");
110
111    // test deleting non-existent stuff
112    deleted = (bstring)BSTree_delete(map, &test3);
113    mu_assert(deleted == NULL, "Should get NULL");
114
115    return NULL;
116 }
117
118 char *test_fuzzing()
119 {
120     BSTree *store = BSTree_create(NULL);
121     int i = 0;
122     int j = 0;
123     bstring numbers[100] = {NULL};
124     bstring data[100] = {NULL};
125     srand((unsigned int)time(NULL));
126
127     for(i = 0; i < 100; i++) {
128         int num = rand();
129         numbers[i] = bformat("%d", num);
130         data[i] = bformat("data %d", num);
131         BSTree_set(store, numbers[i], data[i]);

```

```

132 }
133
134 for(i = 0; i < 100; i++) {
135     bstring value = BSTree_delete(store, numbers[i]);
136     mu_assert(value == data[i], "Failed to delete the right number.");
137
138     mu_assert(BSTree_delete(store, numbers[i]) == NULL, "Should get nothing.");
139
140     for(j = i + 1; j < 99 - i; j++) {
141         bstring value = BSTree_get(store, numbers[j]);
142         mu_assert(value == data[j], "Failed to get the right number.");
143     }
144
145     bdestroy(value);
146     bdestroy(numbers[i]);
147 }
148
149 BSTree_destroy(store);
150
151 return NULL;
152 }
153
154 char *all_tests()
155 {
156     mu_suite_start();
157
158     mu_run_test(test_create);
159     mu_run_test(test_get_set);
160     mu_run_test(test_traverse);
161     mu_run_test(test_delete);
162     mu_run_test(test_destroy);
163     mu_run_test(test_fuzzing);
164
165     return NULL;
166 }
167
168 RUN_TESTS(all_tests);

```

I'll point you at the `test_fuzzing` function, which is an interesting technique for testing complex data structures. It is difficult to create a set of keys that cover all the branches in `BSTree_node_delete`, and chances are I would miss some edge case. A better way is to create a "fuzz" function that does all the operations, but does them in as horrible and random a way as possible. In this case I'm inserting a set of random string keys, then I'm deleting them and trying to get the rest after each delete.

Doing this prevents the situation where you test only what you know to work, which means you'll miss things you don't know. By throwing random junk at your data structures you'll hit things you didn't expect and work out any bugs you have.

41.1 How To Improve It

Do not do any of these yet since in the next exercise I'll be using this unit test to teach you some more performance tuning tricks. You'll come back and do these after you do Exercise 41.

1. As usual, you should go through all the defensive programming checks and add asserts for conditions that shouldn't happen. For example, you should not be getting NULL values for the recursion functions, so assert that.
2. The traverse function traverses the tree in order by traversing left, then right, then the current node. You can create traverse functions for reverse order as well.
3. It does a full string compare on every node, but I could use the Hashmap hashing functions to speed this up. I could hash the keys, then keep the hash in the `BSTreeNode`. Then in each of the set up functions I can hash the key ahead of time, and pass it down to the recursive function. Using this hash I can then compare each node much quicker similar to I do in Hashmap.

41.2 Extra Credit

Again, do not do these yet, wait until Exercise 41 when you can use performance tuning features of Valgrind to do them.

1. There's an alternative way to do this data structure without using recursion. The Wikipedia page shows alternatives that don't use recursion but do the same thing. Why would this be better or worse?
2. Read up on all the different similar trees you can find. There's AVL trees, Red-Black trees, and some non-tree structures like Skip Lists.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 42

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 41: Using Cachegrind And Callgrind For Performance Tuning

In this exercise I'm going to give you a quick course in using two tools for Valgrind called callgrind and cachegrind. These two tools will analyze your program's execution and tell you what parts are running slow. The results are accurate because of the way Valgrind works and help you spot problems such as lines of code that execute too much, hot spots, memory access problems, and even CPU cache misses.

To do this exercise I'm going to use the bstree_tests unit tests you just did to look for places to improve the algorithms used. Make sure your versions of these programs are running without any valgrind errors and that it is exactly like my code. I'll be using dumps of my code to talk about how cachegrind and callgrind work.

42.1 Running Callgrind

To run callgrind you pass the `--tool=callgrind` option to valgrind and it will produce a `callgrind.out.PID` file (where PID is replace with the process ID of the program that ran). Once you run it you can analyze this callgrind.out file with a tool called `callgrind_annotate` which will tell you which functions used the most instructions to run. Here's an example of me running callgrind on bstree_tests and then getting its information:

Source 133: Callgrind On bstree_tests

```

1  $ valgrind --dsymutil=yes --tool=callgrind tests/bstree_tests
2  ...
3  $ callgrind_annotate callgrind.out.1232
4  -----
5  Profile data file 'callgrind.out.1232' (creator: callgrind-3.7.0.SVN)
6  -----
7  I1 cache:
8  D1 cache:
9  LL cache:
10 Timerange: Basic block 0 - 1098689
11 Trigger: Program termination
12 Profiled target: tests/bstree_tests (PID 1232, part 1)
13 Events recorded: 1r
14 Events shown: 1r
15 Event sort order: 1r
16 Thresholds: 99
17 Include dirs:
18 User annotated:
19 Auto-annotation: off
20
21 -----
22 1r
23 -----
24 4,605,808 PROGRAM TOTALS
25
26 -----
27 1r file:function
28 -----

```

```
29 670,486 src/lcthw/bstlib.c:bstrcmp [tests/bstree_tests]
30 194,377 src/lcthw/bstree.c:BSTree_get [tests/bstree_tests]
31 65,580 src/lcthw/bstree.c:default_compare [tests/bstree_tests]
32 16,338 src/lcthw/bstree.c:BSTree_delete [tests/bstree_tests]
33 13,000 src/lcthw/bstlib.c:bformat [tests/bstree_tests]
34 11,000 src/lcthw/bstlib.c:bfromcstralloc [tests/bstree_tests]
35 7,774 src/lcthw/bstree.c:BSTree_set [tests/bstree_tests]
36 5,800 src/lcthw/bstlib.c:bdestroy [tests/bstree_tests]
37 2,323 src/lcthw/bstree.c:BSTreeNode_create [tests/bstree_tests]
38 1,183 /private/tmp/pkg-build/coregrind/vg_preloaded.c:vg_cleanup_env [/usr/local/lib/valgrind/vgpreload_core-amd64-darwin.so]
39
40 $
```

I've removed the unit test run and the valgrind output since it's not very useful for this exercise. What you should look at is the callgrind_annotate output. What this shows you is the number of instructions run (which valgrind calls Ir) for each function, and the functions sorted highest to lowest. You can usually ignore the header data and just jump to the list of functions.

Note 13: More OSX Annoyances

In if you get a ton of "???Image" lines and things that are not in your program then you're picking up junk from the OS. Just add | grep -v "???" at the end to filter those out, like this.

I can now do a quick breakdown of this output to figure out where to look next:

1. Each line lists the number of Ir and the file:function that executed them. The Ir is just the instruction count, and if you make that lower then you have made it faster. There's some complexity to that, but at first just focus on getting the Ir down.
2. The way to attack this is to look at your top functions, and then see which one you think you can improve first. In this case, I'd look at improving bstrcmp or BSTree_get. It's probably easier to start with BSTree_get.
3. Some of these functions are just called from the unit test, so I would just ignore those. Functions like bformat, bfromcstralloc, and bdestroy fit this description.
4. I would also look for functions I can simply avoid calling. For example, maybe I can just say BSTree only works with bstring keys, and then I can just not use the callback system and remove default_compare entirely.

At this point though, I only know that I want to look at BSTree_get to improve it, and not the reason BSTree_get is slow. That is phase two of the analysis.

42.2 Callgrind Annotating Source

I will next tell callgrind_annotate to dump out the bstree.c file and annotate each line with the number of Ir it took. You get the annotated source file by running:

Source 134: Callgrind Annotated BSTree_get

```
1 $ callgrind_annotate callgrind.out.1232 src/lcthw/bstree.c
2 ...
```

Your output will have a big dump of the file's source, but I've cut out the parts for BSTree_get and BSTree_getnode:

Source 135: Callgrind Annotated BSTree_get

```
-----
-- User-annotated source: src/lcthw/bstree.c
-----
Ir
```

```

2,453 static inline BSTreeNode *BSTree_getnode(BSTree *map, BSTreeNode *node, void *key)
{
61,853 int cmp = map->compare(node->key, key);
663,908 => src/lcthw/bstree.c:default_compare (14850x)
{
14,850 if(cmp == 0) {
return node;
24,794 } else if(cmp < 0) {
30,623 if(node->left) {
return BSTree_getnode(map, node->left, key);
} else {
return NULL;
}
} else {
13,146 if(node->right) {
return BSTree_getnode(map, node->right, key);
} else {
return NULL;
}
}
}
}

void *BSTree_get(BSTree *map, void *key)
4,912 {
24,557 if(map->root == NULL) {
14,736 return NULL;
} else {
BSTreeNode *node = BSTree_getnode(map, map->root, key);
2,453 return node == NULL ? NULL : node->data;
}
}

```

Each line is shown with either the number of Ir (instructions) it ran, or a period (.) to show that it's not important. What I'm looking for is hotspots, or lines that have huge numbers of Ir that I can possibly bring down. In this case, line 10 of the output above shows that what makes `BSTree_getnode` so expensive is that it calls `default_comapre` which calls `bstrcmp`. I already know that `bstrcmp` is the worst running function, so if I want to improve the speed of `BSTree_getnode` I should work on that first.

I'll then look at `bstrcmp` the same way:

Source 136: Callgrind Annotated `bstrcmp`

```

98,370 int bstrcmp(const_bstring b0, const_bstring b1) {
int i, v, n;
196,740 if (b0 == NULL || b1 == NULL || b0->data == NULL || b1->data == NULL ||
32,790 b0->slen < 0 || b1->slen < 0) return SHRT_MIN;
65,580 n = b0->slen; if (n > b1->slen) n = b1->slen;
89,449 if (b0->slen == b1->slen && (b0->data == b1->data || b0->slen == 0))
return BSTR_OK;
23,915 for (i = 0; i < n; i++) {
163,642 v = ((char) b0->data[i]) - ((char) b1->data[i]);
if (v != 0) return v;
if (b0->data[i] == (unsigned char) '\0') return BSTR_OK;
}
if (b0->slen > n) return 1;
if (b1->slen > n) return -1;
return BSTR_OK;
}

```

The Ir for this function shows two lines that take up most of the execution. First, `bstrcmp` seems to go through a lot of trouble to make sure that it is not given a NULL value. That's a good thing so I want to leave that alone, but I'd consider writing a different compare function that was more "risky" and assumed it was never given a NULL. The next one is the loop that does the actual comparison. It seems that there's some optimization that could be done in comparing the

characters of the two data buffers.

42.3 Analyzing Memory Access With Cachegrind

What I want to do next is see how many times this `bstrcmp` function access memory to either read it or write it. The tool for doing that (and other things) is `cachegrind` and you use it like this:

Source 137: Cachegrind On `bstree_tests`

```
1 $ valgrind --tool=cachegrind tests/bstree_tests
2 ...
3 $ cg_annotate --show=Dr,Dw cachegrind.out.1316 | grep -v "???"
4 -----
5 I1 cache:      32768 B, 64 B, 8-way associative
6 D1 cache:      32768 B, 64 B, 8-way associative
7 LL cache:     4194304 B, 64 B, 16-way associative
8 Command:      tests/bstree_tests
9 Data file:     cachegrind.out.1316
10 Events recorded: Ir I1mr I1Lmr Dr D1mr DLmr Dw D1mw DLmw
11 Events shown:  Dr Dw
12 Event sort order: Ir I1mr I1Lmr Dr D1mr DLmr Dw D1mw DLmw
13 Thresholds:    0.1 100 100 100 100 100 100 100 100
14 Include dirs:
15 User annotated:
16 Auto-annotation: off
17
18 -----
19      Dr    Dw
20 -----
21 997,124 349,058 PROGRAM TOTALS
22
23 -----
24      Dr    Dw file:function
25 -----
26 169,754 19,430 src/lcthw/bstrlib.c:bstrcmp
27 67,548 27,428 src/lcthw/bstree.c:BSTree_get
28 19,430 19,430 src/lcthw/bstree.c:default_compare
29 5,420 2,383 src/lcthw/bstree.c:BSTree_delete
30 2,000 4,200 src/lcthw/bstrlib.c:bformat
31 1,600 2,800 src/lcthw/bstrlib.c:bfromcstralloc
32 2,770 1,410 src/lcthw/bstree.c:BSTree_set
33 1,200 1,200 src/lcthw/bstrlib.c:bdestroy
34
35 $
```

I tell `valgrind` to use the `cachegrind` tool, which runs `bstree_tests` and then produces a `cachegrind.out.PID` file just like `callgrind` did. I then use the program `cg_annotate` to get a similar output, but notice that I'm telling it to do `--show=Dr,Dw`. This option says that I only want the memory read `Dr` and write `Dw` counts for each function.

After that you get your usual header and then the counts for `Dr` and `Dw` for each `file:function` combination. I've edited this down so it shows the files and also removed any OS junk with `| grep -v "???"` so your output may be a little different. What you see in my output is that `bstrcmp` is the worst function for memory usage too, which is to be expected since that's mostly the only thing it does. I'm going to now dump it's annotated source to see where.

Source 138: Cachegrind Annotated `bstrcmp`

```
-----
-- User-annotated source: src/lcthw/bstrlib.c
-----
Dr    Dw

0 19,430 int bstrcmp (const_bstring b0, const_bstring b1) {
.    . int i, v, n;
.
77,720 0 if (b0 == NULL || b1 == NULL || b0->data == NULL || b1->data == NULL ||
38,860 0 b0->slen < 0 || b1->slen < 0) return SHRT_MIN;
0 0 n = b0->slen; if (n > b1->slen) n = b1->slen;
0 0 if (b0->slen == b1->slen && (b0->data == b1->data || b0->slen == 0))
```



```

    . . return BSTR_OK;
    . .
53,174 0 0 for (i = 0; i < n; i++) {
    . . 0 v = ((char) b0->data[i]) - ((char) b1->data[i]);
    . . if (v != 0) return v;
    . . if (b0->data[i] == (unsigned char) '\0') return BSTR_OK;
    . . }
    . .
    . . if (b0->slen > n) return 1;
    . . if (b1->slen > n) return -1;
    . . return BSTR_OK;
    . . }

```

The surprising thing about this output is that the worst line of `bstrcmp` isn't the character comparison like I thought. For memory access it's that protective `if`-statement at the top that checks every possible bad variable it could receive. That one `if` statement does more than twice as many memory accesses compared to the line that's comparing the characters on line 17 of this output. If I were to make `bstrcmp` then I would definitely just ditch that or do it once somewhere else.

Another option is to turn this check into an `assert` that only exists when running in development, and then compile it out in production. I now have enough evidence to say that this line is bad for this data structure, so I can justify removing it.

What I don't want to do however is justify making this function less defensive to just gain a few more cycles. In a real performance improvement situation I would simply put this on a list and then dig for other gains I can make in the program.

42.4 Judo Tuning

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

(Donald Knuth)

In my opinion, this quote seems to miss a major point about performance tuning. In this Knuth is saying that when you performance tune matters, in that if you do it in the beginning, then you'll cause all sorts of problems. According to him optimization should happen "sometime later", or at least that's my guess. Who knows these days really.

I'm going to declare this quote not necessarily wrong, but missing the point, and instead I'm going to officially give my quote. You can quote me on this:

"Use evidence to find the biggest optimizations that take the least effort."

(Zed A. Shaw)

It doesn't matter when you try to optimize something, but instead it's how you figure out if your optimization actually improved the software, and how much effort you put into doing them. With evidence you can find the places in the code where just a little effort gets you big improvements. Usually these places are just dumb decisions, as in `bstrcmp` trying to check everything possible for a `NULL` value.

At a certain point you have tuned the code to where the only thing that remains is tiny little micro-optimizations such as reorganizing `if`-statements and special loops like Duff's Device. At this point, just stop because there's a good chance that you'd gain more by redesigning the software to just not do things.

This is something that programmers who are optimizing simply fail to see. Many times the best way to do something fast is to find out ways to not do them. In the above analysis, I wouldn't try to make `bstrcmp` faster, I'd try to find a way to not use `bstrcmp` so much. Maybe there's a hashing scheme I can use that let's me do a sortable hash instead of constantly

doing `bstrcmp`. Maybe I can optimize it by trying the first char first, and if it's comparable just don't call `bstrcmp`.

If after all that you can't do a redesign then start looking for little micro-optimizations, but as you do them constantly confirm they improve speed. Remember that the goal is to cause the biggest impact with the least effort possible.

42.5 Using KCachegrind

The final section of this exercise is going to point you at a tool called [KCachegrind](#). This is a fantastic GUI for analyzing callgrind and cachegrind output. I use it almost exclusively when I'm working on a Linux or BSD computer, and I've actually switched to just coding on Linux for projects because of KCachegrind.

Teaching you how to use it is outside the scope of this exercise, but you should be able to understand how to use it after this exercise. The output is nearly the same except KCachegrind lets you do the following:

1. Graphically browse the source and execution times doing various sorts to find things to improve.
2. Analyze different graphs to visually see what's taking up the most time and also what it is calling.
3. Look at the actual machine code assembler output so you can see possible instructions that are happening, giving you more clues.
4. Visualize the jump patterns for loops and branches in the source code, helping you find ways to optimize the code easier.

You should spend some time getting KCachegrind installed and play with it.

42.6 Extra Credit

1. Read the [callgrind manual](#) and try some advanced options.
2. Read the [cachegrind manual](#) and also try some advanced options.
3. Use callgrind and cachegrind on all the unit tests and see if you can find optimizations to make. Did you find some things that surprised you? If not you probably aren't looking hard enough.
4. Use KCachegrind and see how it compares to doing the terminal output like I'm doing here.
5. Now use these tools to do the Exercise 40 extra credits and improvements.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 43

Exercise 42: Stacks and Queues

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

At this point in the book you know most of the data structures that are used to build all the other data structures. If you have some kind of List, DArray, Hashmap, and Tree then you can build most anything else that's out there. Everything else you run into either uses these or is some variant on these. If it's not then it's most likely an exotic data structure that you probably will not need.

Stacks and Queues are very simple data structures that are really variants of the List data structure. All they are is using a List with a "discipline" or convention that says you'll always place elements on one end of the List. For a Stack, you always push and pop. For a Queue, you always shift to the front, but pop from the end.

I can implement both data structures using nothing but the CPP and two header files. My header files are 21 lines long and do all the Stack and Queue operations without any fancy defines.

To see if you've been paying attention, I'm going to show you the unit tests, and then you have to implement the header files needed to make them work. To pass this exercise you can't create any stack.c or queue.c implementation files. Use only the stack.h and queue.h files to make the tests runs.

Source 139: tests/stack_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/stack.h>
3  #include <assert.h>
4
5  static Stack *stack = NULL;
6  char *tests[] = {"test1 data", "test2 data", "test3 data"};
7  #define NUM_TESTS 3
8
9
10 char *test_create()
11 {
12     stack = Stack_create();
13     mu_assert(stack != NULL, "Failed to create stack.");
14
15     return NULL;
16 }
17
18 char *test_destroy()
19 {
20     mu_assert(stack != NULL, "Failed to make stack #2");
21     Stack_destroy(stack);
22
23     return NULL;
24 }
25
26 char *test_push_pop()
27 {
28     int i = 0;
29     for(i = 0; i < NUM_TESTS; i++) {
30         Stack_push(stack, tests[i]);
31         mu_assert(Stack_peek(stack) == tests[i], "Wrong next value.");
32     }

```

```

    }
33
34     mu_assert(Stack_count(stack) == NUM_TESTS, "Wrong count on push.");
35
36     STACK_FOREACH(stack, cur) {
37         debug("VAL: %s", (char *)cur->value);
38     }
39
40     for(i = NUM_TESTS - 1; i >= 0; i--) {
41         char *val = Stack_pop(stack);
42         mu_assert(val == tests[i], "Wrong value on pop.");
43     }
44
45     mu_assert(Stack_count(stack) == 0, "Wrong count after pop.");
46
47     return NULL;
48 }
49
50 char *all_tests() {
51     mu_suite_start();
52
53     mu_run_test(test_create);
54     mu_run_test(test_push_pop);
55     mu_run_test(test_destroy);
56
57     return NULL;
58 }
59
60 RUN_TESTS(all_tests);

```

Then the queue_tests.c is almost the same just using Queue:

Source 140: tests/queue_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/queue.h>
3  #include <assert.h>
4
5  static Queue *queue = NULL;
6  char *tests[] = {"test1 data", "test2 data", "test3 data"};
7  #define NUM_TESTS 3
8
9
10 char *test_create()
11 {
12     queue = Queue_create();
13     mu_assert(queue != NULL, "Failed to create queue.");
14
15     return NULL;
16 }
17
18 char *test_destroy()
19 {
20     mu_assert(queue != NULL, "Failed to make queue #2");
21     Queue_destroy(queue);
22
23     return NULL;
24 }
25
26 char *test_send_rcv()
27 {
28     int i = 0;
29     for(i = 0; i < NUM_TESTS; i++) {
30         Queue_send(queue, tests[i]);
31         mu_assert(Queue_peek(queue) == tests[0], "Wrong next value.");
32     }
33
34     mu_assert(Queue_count(queue) == NUM_TESTS, "Wrong count on send.");
35
36     QUEUE_FOREACH(queue, cur) {
37         debug("VAL: %s", (char *)cur->value);
38     }
39
40     for(i = 0; i < NUM_TESTS; i++) {
41         char *val = Queue_rcv(queue);
42         mu_assert(val == tests[i], "Wrong value on rcv.");
43     }
44

```

```

45     mu_assert(Queue_count(queue) == 0, "Wrong count after recv.");
46
47     return NULL;
48 }
49
50 char *all_tests() {
51     mu_suite_start();
52
53     mu_run_test(test_create);
54     mu_run_test(test_send_recv);
55     mu_run_test(test_destroy);
56
57     return NULL;
58 }
59
60 RUN_TESTS(all_tests);

```

43.1 What You Should See

Your unit test should run without you changing the tests, and it should pass valgrind with no memory errors. Here's what it looks like if I run `stack_tests` directly:

Source 141: `stack_tests` run

```

1  $ ./tests/stack_tests
2  DEBUG tests/stack_tests.c:60: ----- RUNNING: ./tests/stack_tests
3  -----
4  RUNNING: ./tests/stack_tests
5  DEBUG tests/stack_tests.c:53:
6  ----- test_create
7  DEBUG tests/stack_tests.c:54:
8  ----- test_push_pop
9  DEBUG tests/stack_tests.c:37: VAL: test3 data
10 DEBUG tests/stack_tests.c:37: VAL: test2 data
11 DEBUG tests/stack_tests.c:37: VAL: test1 data
12 DEBUG tests/stack_tests.c:55:
13 ----- test_destroy
14 ALL TESTS PASSED
15 Tests run: 3
16 $

```

The `queue_test` is basically the same kind of output so I shouldn't have to show it to you at this stage.

43.2 How To Improve It

The only real improvement you could make to this is to switch from using a List to using a DArray. The Queue data structure is more difficult to do with a DArray because it works at both ends of the list of nodes.

A disadvantage of doing this entirely in a header file is that you can't easily performance tune it. Mostly what you're doing with this technique is establishing a "protocol" for how to use a List in a certain style. When performance tuning, if you make List fast then these two should improve as well.

43.3 Extra Credit

1. Implement Stack using DArray instead of List without changing the unit test. That means you'll have to create your own `STACK_FOREACH`.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 44

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 43: A Simple Statistics Engine

This is a simple algorithm I use for collecting summary statistics "online", or without storing all of the samples. I use this in any software that needs to keep some statistics such as mean, standard deviation, and sum, but where I can't store all the samples needed. Instead I can just store the rolling results of the calculations which is only 5 numbers.

44.1 Rolling Standard Deviation And Mean

The first thing you need is a sequence of samples. This can be anything from time to complete a task, numbers of times someone accesses something, or even star ratings on a website. Doesn't really matter what, just so long as you have a stream of numbers and you want to know the following summary statistics about them:

sum

This is the total of all the numbers added together.

sum squared (sumsq)

This is the sum of the square of each number.

count (n)

This is the number samples you've taken.

min

This is the smallest sample you've seen.

max

This is the largest sample you've seen.

mean

This is the most likely middle number. It's not actually the middle, since that's the median, but it's an accepted approximation for it.

stddev

Calculated using $\sqrt{\text{sumsq} - (\text{sum} * \text{mean}) / (n - 1)}$ where sqrt is the square root function in the math.h header.

I will confirm this calculation works using R since I know R gets these right:

Source 142: Standard Deviation in R

```
1 > s <- runif(n=10, max=10)
2 > s
3 [1] 6.1061334 9.6783204 1.2747090 8.2395131 0.3333483 6.9755066 1.0626275
4 [8] 7.6587523 4.9382973 9.5788115
5 > summary(s)
6   Min. 1st Qu.  Median    Mean 3rd Qu.   Max.
7  0.3333  2.1910  6.5410  5.5850  8.0940  9.6780
8 > sd(s)
9 [1] 3.547868
```



```

10 > sum(s)
11 [1] 55.84602
12 > sum(s * s)
13 [1] 425.1641
14 > sum(s) * mean(s)
15 [1] 311.8778
16 > sum(s * s) - sum(s) * mean(s)
17 [1] 113.2863
18 > (sum(s * s) - sum(s) * mean(s)) / (length(s) - 1)
19 [1] 12.58737
20 > sqrt((sum(s * s) - sum(s) * mean(s)) / (length(s) - 1))
21 [1] 3.547868
22 >

```

You don't need to know R, just follow along while I explain how I'm breaking this down to check my math:

lines 1–4

I use the function `runif` to get a "random uniform" distribution of numbers, then print them out. I'll use these in the unit test later.

lines 5–7

Here's the summary, so you can see the values that R calculates for these.

lines 8–9

This is the stddev using the `sd` function.

lines 10–11

Now I begin to build this calculation manually, first by getting the sum.

lines 12–13

Next piece of the stdev formula is the `sumsq`, which I can get with `sum(s * s)` which tells R to multiple the whole `s` list by itself and then sum those.¹

lines 14–15

Looking at the formula, I then need the sum multiplied by mean, so I do `sum(s) * mean(s)`.

lines 16–17

I then combine the `sumsq` with this to get `sum(s * s) - sum(s) * mean(s)`.

lines 18–19

That needs to be divided by $n - 1$, so I do `(sum(s * s) - sum(s) * mean(s)) / (length(s) - 1)`.

lines 20–21

Finally, I `sqrt` that and I get 3.547868 which matches the number R gave me for `sd` above.

44.2 Implemention

That's how you calculate the stddev, so now I can make some simple code to implement this calculation.

Source 143: `src/lcthw/stats.h`

```

1  #ifndef lcthw_stats_h
2  #define lcthw_stats_h
3
4  typedef struct Stats {
5      double sum;
6      double sumsq;
7      unsigned long n;
8      double min;
9      double max;
10 } Stats;
11
12 Stats *Stats_recreate(double sum, double sumsq, unsigned long n, double min, double max);
13
14 Stats *Stats_create();
15
16 double Stats_mean(Stats *st);
17

```

```

18 double Stats_stddev(Stats *st);
19
20 void Stats_sample(Stats *st, double s);
21
22 void Stats_dump(Stats *st);
23
24 #endif

```

Here you can see I've put the calculations I need to store in a struct and then I have functions for sampling and getting the numbers. Implementing this is then just an exercise in converting the math:

Source 144: src/lcthw/stats.c

```

1  #include <math.h>
2  #include <lcthw/stats.h>
3  #include <stdlib.h>
4  #include <lcthw/dbg.h>
5
6  Stats *Stats_recreate(double sum, double sumsq, unsigned long n, double min, double max)
7  {
8      Stats *st = malloc(sizeof(Stats));
9      check_mem(st);
10
11      st->sum = sum;
12      st->sumsq = sumsq;
13      st->n = n;
14      st->min = min;
15      st->max = max;
16
17      return st;
18
19  error:
20      return NULL;
21  }
22
23  Stats *Stats_create()
24  {
25      return Stats_recreate(0.0, 0.0, 0L, 0.0, 0.0);
26  }
27
28  double Stats_mean(Stats *st)
29  {
30      return st->sum / st->n;
31  }
32
33  double Stats_stddev(Stats *st)
34  {
35      return sqrt( (st->sumsq - (st->sum * st->sum / st->n)) / (st->n - 1) );
36  }
37
38  void Stats_sample(Stats *st, double s)
39  {
40      st->sum += s;
41      st->sumsq += s * s;
42
43      if(st->n == 0) {
44          st->min = s;
45          st->max = s;
46      } else {
47          if(st->min > s) st->min = s;
48          if(st->max < s) st->max = s;
49      }
50
51      st->n += 1;
52  }
53
54  void Stats_dump(Stats *st)
55  {
56      fprintf(stderr, "sum: %f, sumsq: %f, n: %ld, min: %f, max: %f, mean: %f, stddev: %f",
57          st->sum, st->sumsq, st->n, st->min, st->max,
58          Stats_mean(st), Stats_stddev(st));
59  }

```

Here's what each function in stats.c does:

Stats_recreate

I'll want to load these numbers from some kind of database, and this function let's me recreate a Stats struct.

Stats_create

Simply called Stats_recreate with all 0 values.

Stats_mean

Using the sum and n it gives the mean.

Stats_stddev

Implements the formula I worked out, with the only difference being that I calculate the mean with $st \rightarrow \text{sum} / st \rightarrow n$ in this formula instead of calling Stats_mean.

Stats_sample

This does the work of maintaining the numbers in the Stats struct. When you give it the first value it sees that n is 0 and sets min and max accordingly. Every call after that keeps increasing sum, sumsq, and n. It then figures out if this new sample is a new min or max.

Stats_dump

Simple debug function that dumps the stats so you can view them.

The last thing I need to do is confirm that this math is correct. I'm going to use my numbers and calculations from my R session to create a unit test that confirms I'm getting the right results.

Source 145: tests/stats_tests.c

```

1  #include "minunit.h"
2  #include <lcthw/stats.h>
3  #include <math.h>
4
5  const int NUM_SAMPLES = 10;
6  double samples[] = {
7      6.1061334, 9.6783204, 1.2747090, 8.2395131, 0.3333483,
8      6.9755066, 1.0626275, 7.6587523, 4.9382973, 9.5788115
9  };
10
11 Stats expect = {
12     .sumsq = 425.1641,
13     .sum = 55.84602,
14     .min = 0.333,
15     .max = 9.678,
16     .n = 10,
17 };
18 double expect_mean = 5.584602;
19 double expect_stddev = 3.547868;
20
21 #define EQ(X,Y,N) (round((X) * pow(10, N)) == round((Y) * pow(10, N)))
22
23 char *test_operations()
24 {
25     int i = 0;
26     Stats *st = Stats_create();
27     mu_assert(st != NULL, "Failed to create stats.");
28
29     for(i = 0; i < NUM_SAMPLES; i++) {
30         Stats_sample(st, samples[i]);
31     }
32
33     Stats_dump(st);
34
35     mu_assert(EQ(st->sumsq, expect.sumsq, 3), "sumsq not valid");
36     mu_assert(EQ(st->sum, expect.sum, 3), "sum not valid");
37     mu_assert(EQ(st->min, expect.min, 3), "min not valid");
38     mu_assert(EQ(st->max, expect.max, 3), "max not valid");
39     mu_assert(EQ(st->n, expect.n, 3), "n not valid");
40     mu_assert(EQ(expect_mean, Stats_mean(st), 3), "mean not valid");
41     mu_assert(EQ(expect_stddev, Stats_stddev(st), 3), "stddev not valid");
42
43     return NULL;
44 }
45
46 char *test_recreate()
47
```

```

48     Stats *st = Stats_recreate(expect.sum, expect.sumsq, expect.n, expect.min, expect.max);
49
50     mu_assert(st->sum == expect.sum, "sum not equal");
51     mu_assert(st->sumsq == expect.sumsq, "sumsq not equal");
52     mu_assert(st->n == expect.n, "n not equal");
53     mu_assert(st->min == expect.min, "min not equal");
54     mu_assert(st->max == expect.max, "max not equal");
55     mu_assert(EQ(expect_mean, Stats_mean(st), 3), "mean not valid");
56     mu_assert(EQ(expect_stddev, Stats_stddev(st), 3), "stddev not valid");
57
58     return NULL;
59 }
60
61 char *all_tests()
62 {
63     mu_suite_start();
64
65     mu_run_test(test_operations);
66     mu_run_test(test_recreate);
67
68     return NULL;
69 }
70
71 RUN_TESTS(all_tests);

```

There's nothing new in this unit test, except maybe the EQ macro. I felt lazy and didn't want to look up the standard way to tell if two double values are close, so I made this macro. The problem with double is that equality assumes totally equal, but I'm using two different systems with slightly different rounding errors. The solution is to say I want the numbers to be "equal to X decimal places".

I do this with EQ by raising the number to a power of 10, then using the round function to get an integer. This is a simple way to round to N decimal places and compare the results as an integer. I'm sure there's a billion other ways to do the same thing, but this works for now.

The expected results are then in a Statsstruct and then I simply make sure that the number I get is close to the number R gave me.

44.3 How To Use It

You can use the standard deviation and mean to determine if a new sample is "interesting", or you can use this to collect statistics on statistics. The first one is easy for people to understand so I'll explain that quickly using an example for login times.

Imagine you're tracking how long users spend on a server and you're using stats to analyze it. Every time someone logs in, you keep track of how long they are there, then you call Stats_sample. I'm looking for people are a on "too long" and also people who seem to be on "too quickly".

Instead of setting specific levels, what I'd do is compare how long someone is on with the mean (plus or minus) 2 * stddev range. I get the mean and 2 * stddev, and consider login times to be "interesting" if they are outside these two ranges. Since I'm keeping these statistics using a rolling algorithm this is a very fast calculation and I can then have the software flag the users who are outside of this range.

This doesn't necessarily point out people who are behaving badly, but instead it flags potential problems that you can review to see what's going on. It's also doing it based on the behavior of all the users, which avoids the problem where you pick some arbitrary number that's not based on what's really happening.

The general rule you can get from this is that the mean (plus or minus) 2 * stddev is an estimate of where 90% of the values are expected to fall, and that anything outside those ranges is interesting.

The second way to use these statistics is to go meta and calculate them for other Stats calculations. You basically do

your Stats_sample like normal, but then you run Stats_sample on the min, max, n, mean, and stddev on that sample. This gives a two-level measurement, and let's you compare samples of samples.

Confusing right? I'll continue my example above and add that you have 100 servers that each hold a different application. You are already tracking user's login times for each application server, but you want to compare all 100 applications and flag any users that are logging in "too much" on all of them. Easiest way to do that is each time someone logs in, calculate the new login stats, then add that Stats structs elements to a second Stat.

What you end up with is a series of stats that can be named like this:

mean of means

This is a full Stats struct that gives you mean and stddev of the means of all the servers. Any server or user who is outside of this is work looking at on a global level.

mean of stddevs

Another Stats struct that produces the statistics of how all of the servers range. You can then analyze each server and see if any of them have unusually wide ranging numbers by comparing their stddev to this mean of stddevs statistic.

You could do them all, but these are the most useful. If you wanted to then monitor servers for erratic login times you'd do this:

1. User John logs into and out of server A. Grab server A's stats, update them.
2. Grab the mean of means stats, and take A's mean and add it as a sample. I'll call this m_of_m.
3. Grab the mean of stddevs stats, and add A's stddev to it as a sample. I'll call this m_of_s.
4. If A's mean is outside of $m_of_m.mean + 2 * m_of_m.stddev$ then flag it as possibly having a problem.
5. If A's stddev is outside of $m_of_s.mean + 2 * m_of_s.stddev$ then flag it as possible behaving too erratically.
6. Finally, if John's login time is outside of A's range, or A's m_of_m range, then flag it as interesting.

Using this "mean of means" and "mean of stddevs" calculation you can do efficient tracking of many metrics with a minimal amount of processing and storage.

44.4 Extra Credit

1. Convert the Stats_stddev and Stats_mean to static inline functions in the stats.h file instead of in the stats.c file.
2. Use this code to write a performance test of the string_algos_test.c. Make it optional and have it run the base test as a series of samples then report the results.
3. Write a version of this in another programming language you know. Confirm that this version is correct based on what I have here.
4. Write a little program that can take a file full of numbers and spit these statistics out for them.
5. Make the program accept a table of data that has headers on one line, then all the other numbers on lines after it separated by any number of spaces. Your program should then print out these stats for each column by the header name.

¹The power of R is being able to do math on entire data structures like this.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 45

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 44: Ring Buffer

Ring buffers are incredibly useful when processing asynchronous IO. They allow one side to receive data in random intervals of random sizes, but feed cohesive chunks to another side in set sizes or intervals. They are a variant on the Queue data structure but it focuses on blocks of bytes instead of a list of pointers. In this exercise I'm going to show you the RingBuffer code, and then you have to make a full unit test for it.

Source 146: src/lcthw/ringbuffer.h

```

1  #ifndef _lcthw_RingBuffer_h
2  #define _lcthw_RingBuffer_h
3
4  #include <lcthw/bstrlib.h>
5
6  typedef struct {
7      char *buffer;
8      int length;
9      int start;
10     int end;
11 } RingBuffer;
12
13 RingBuffer *RingBuffer_create(int length);
14
15 void RingBuffer_destroy(RingBuffer *buffer);
16
17 int RingBuffer_read(RingBuffer *buffer, char *target, int amount);
18
19 int RingBuffer_write(RingBuffer *buffer, char *data, int length);
20
21 int RingBuffer_empty(RingBuffer *buffer);
22
23 int RingBuffer_full(RingBuffer *buffer);
24
25 int RingBuffer_available_data(RingBuffer *buffer);
26
27 int RingBuffer_available_space(RingBuffer *buffer);
28
29 bstring RingBuffer_gets(RingBuffer *buffer, int amount);
30
31 #define RingBuffer_available_data(B) ((B)->end + 1) % (B)->length - (B)->start - 1)
32
33 #define RingBuffer_available_space(B) ((B)->length - (B)->end - 1)
34
35 #define RingBuffer_full(B) (RingBuffer_available_data((B)) - (B)->length == 0)
36
37 #define RingBuffer_empty(B) (RingBuffer_available_data((B)) == 0)
38
39 #define RingBuffer_puts(B, D) RingBuffer_write((B), bdata((D)), blength((D)))
40
41 #define RingBuffer_get_all(B) RingBuffer_gets((B), RingBuffer_available_data((B)))
42
43 #define RingBuffer_starts_at(B) ((B)->buffer + (B)->start)
44
45 #define RingBuffer_ends_at(B) ((B)->buffer + (B)->end)
46
47 #define RingBuffer_commit_read(B, A) ((B)->start = ((B)->start + (A)) % (B)->length)
48

```

```

49 #define RingBuffer_commit_write(B, A) ((B)->end = ((B)->end + (A)) % (B)->length)
50
51 #endif

```

Looking at the data structure you see I have a buffer, start and end. A RingBuffer does nothing more than move the start and end around the buffer so that it "loops" whenever it reaches the buffer's end. Doing this gives the illusion of an infinite read device in a small space. I then have a bunch of macros that do various calculations based on this.

Here's the implementation which is a much better explanation of how this works:

Source 147: src/lcthw/ringbuffer.c

```

1  #undef NDEBUG
2  #include <assert.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <lcthw/dbg.h>
7  #include <lcthw/ringbuffer.h>
8
9  RingBuffer *RingBuffer_create(int length)
10 {
11     RingBuffer *buffer = calloc(1, sizeof(RingBuffer));
12     buffer->length = length + 1;
13     buffer->start = 0;
14     buffer->end = 0;
15     buffer->buffer = calloc(buffer->length, 1);
16
17     return buffer;
18 }
19
20 void RingBuffer_destroy(RingBuffer *buffer)
21 {
22     if(buffer) {
23         free(buffer->buffer);
24         free(buffer);
25     }
26 }
27
28 int RingBuffer_write(RingBuffer *buffer, char *data, int length)
29 {
30     if(RingBuffer_available_data(buffer) == 0) {
31         buffer->start = buffer->end = 0;
32     }
33
34     check(length <= RingBuffer_available_space(buffer),
35           "Not enough space: %d request, %d available",
36           RingBuffer_available_data(buffer), length);
37
38     void *result = memcpy(RingBuffer_ends_at(buffer), data, length);
39     check(result != NULL, "Failed to write data into buffer.");
40
41     RingBuffer_commit_write(buffer, length);
42
43     return length;
44 error:
45     return -1;
46 }
47
48 int RingBuffer_read(RingBuffer *buffer, char *target, int amount)
49 {
50     check_debug(amount <= RingBuffer_available_data(buffer),
51               "Not enough in the buffer: has %d, needs %d",
52               RingBuffer_available_data(buffer), amount);
53
54     void *result = memcpy(target, RingBuffer_starts_at(buffer), amount);
55     check(result != NULL, "Failed to write buffer into data.");
56
57     RingBuffer_commit_read(buffer, amount);
58
59     if(buffer->end == buffer->start) {
60         buffer->start = buffer->end = 0;
61     }
62
63     return amount;
64 error:

```



```

65     return -1;
66 }
67
68 bstring RingBuffer_gets(RingBuffer *buffer, int amount)
69 {
70     check(amount > 0, "Need more than 0 for gets, you gave: %d ", amount);
71     check_debug(amount <= RingBuffer_available_data(buffer),
72                 "Not enough in the buffer.");
73
74     bstring result = blk2bstr(RingBuffer_starts_at(buffer), amount);
75     check(result != NULL, "Failed to create gets result.");
76     check(bllength(result) == amount, "Wrong result length.");
77
78     RingBuffer_commit_read(buffer, amount);
79     assert(RingBuffer_available_data(buffer) >= 0 && "Error in read commit.");
80
81     return result;
82 error:
83     return NULL;
84 }

```

This is all there is to a basic RingBuffer implementation. You can read and write blocks of data to it. You can ask how much is in it and how much space it has. There are some fancier ring buffers that use tricks in the OS to create an imaginary infinite store, but those aren't portable.

Since my RingBuffer deals with reading and writing blocks of memory, I'm making sure that any time `end == start` then I reset them to 0 (zero) so that they go to the beginning of the buffer. In the Wikipedia version it wasn't writing blocks of data, so it only had to move `end` and `start` around in a circle. To better handle blocks you have to drop to the beginning of the internal buffer whenever the data is empty.

45.1 The Unit Test

For your unit test, you'll want to test as many possible conditions as you can. Easiest way to do that is to preconstruct different RingBuffer structs and then manually check that the functions and math work right. For example, you could make one where `end` is right at the end of the buffer and `start` is right before it, then see how it fails.

45.2 What You Should See

Here's my `ringbuffer_tests` run:

Source 148: `ringbuffer_tests`

```

1  $ ./tests/ringbuffer_tests
2  DEBUG tests/ringbuffer_tests.c:60: ----- RUNNING: ./tests/ringbuffer_tests
3  -----
4  RUNNING: ./tests/ringbuffer_tests
5  DEBUG tests/ringbuffer_tests.c:53:
6  ----- test_create
7  DEBUG tests/ringbuffer_tests.c:54:
8  ----- test_read_write
9  DEBUG tests/ringbuffer_tests.c:55:
10 ----- test_destroy
11 ALL TESTS PASSED
12 Tests run: 3
13 $

```

You should have at least three tests that confirm all the basic operations, and then see how much more you can test beyond what I've done.

45.3 How To Improve It

As usual you should go back and add the defensive programming checks to this exercise. Hopefully you've been doing this because the base code in most of liblcthw doesn't check for common defensive programming that I'm teaching you. I leave this to you so that you get used to improving code with these extra checks.

For example, in this ring buffer there's not a lot of checking that an access will actually be inside the buffer.

If you read the [Ring Buffer Wikipedia page](#) you'll see the "Optimized POSIX implementation" that uses POSIX specific calls to create an infinite space. Study that as I'll have you try it in the extra credit.

45.4 Extra Credit

1. Create an alternative implementation of RingBuffer that uses the POSIX trick and a unit test for it.
2. Add a performance comparison test to this unit test that compares the two versions by fuzzing them with random data and random read/write operations. Make sure that you setup this fuzzing so that the same operations are done to each so you can compare them between runs.
3. Use callgrind and cachegrind to compare the performance of these two.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 46

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 45: A Simple TCP/IP Client

I'm going to use the RingBuffer to create a very simplistic little network testing tool called netclient. To do this I have to add some stuff to the Makefile to handle little programs in the bin/ directory.

46.1 Augment The Makefile

First, add a variable for the programs just like the unit tests TESTS and TEST_SRC variables:

```
PROGRAMS_SRC=$(wildcard bin/*.c)
PROGRAMS=$(patsubst %.c,%, $(PROGRAMS_SRC))
```

Then you want to add the PROGRAMS to the all target:

```
all: $(TARGET) $(SO_TARGET) tests $(PROGRAMS)
```

Then add PROGRAMS to the rm line in the clean target:

```
rm -rf build $(OBJECTS) $(TESTS) $(PROGRAMS)
```

Finally you just need a target at the end to build them all:

```
$(PROGRAMS): CFLAGS += $(TARGET)
```

With these changes you can drop simple .c files into bin and make will build them and link them to the library just like the tests are done.

46.2 The netclient Code

The code for the little netclient looks like this:

Source 149: bin/netclient.c

```
1 #undef NDEBUG
2 #include <stdlib.h>
3 #include <sys/select.h>
4 #include <stdio.h>
5 #include <lcthw/ringbuffer.h>
6 #include <lcthw/dbg.h>
7 #include <sys/socket.h>
8 #include <sys/types.h>
9 #include <sys/uio.h>
10 #include <arpa/inet.h>
11 #include <netdb.h>
12 #include <unistd.h>
```

```

13 #include <fcntl.h>
14
15 struct tagbstring NL = bsStatic("\n");
16 struct tagbstring CRLF = bsStatic("\r\n");
17
18 int nonblock(int fd) {
19     int flags = fcntl(fd, F_GETFL, 0);
20     check(flags >= 0, "Invalid flags on nonblock.");
21
22     int rc = fcntl(fd, F_SETFL, flags | O_NONBLOCK);
23     check(rc == 0, "Can't set nonblocking.");
24
25     return 0;
26 error:
27     return -1;
28 }
29
30 int client_connect(char *host, char *port)
31 {
32     int rc = 0;
33     struct addrinfo *addr = NULL;
34
35     rc = getaddrinfo(host, port, NULL, &addr);
36     check(rc == 0, "Failed to lookup %s:%s", host, port);
37
38     int sock = socket(AF_INET, SOCK_STREAM, 0);
39     check(sock >= 0, "Cannot create a socket.");
40
41     rc = connect(sock, addr->ai_addr, addr->ai_addrlen);
42     check(rc == 0, "Connect failed.");
43
44     rc = nonblock(sock);
45     check(rc == 0, "Can't set nonblocking.");
46
47     freeaddrinfo(addr);
48     return sock;
49
50 error:
51     freeaddrinfo(addr);
52     return -1;
53 }
54
55 int read_some(RingBuffer *buffer, int fd, int is_socket)
56 {
57     int rc = 0;
58
59     if(RingBuffer_available_data(buffer) == 0) {
60         buffer->start = buffer->end = 0;
61     }
62
63     if(is_socket) {
64         rc = recv(fd, RingBuffer_starts_at(buffer), RingBuffer_available_space(buffer), 0);
65     } else {
66         rc = read(fd, RingBuffer_starts_at(buffer), RingBuffer_available_space(buffer));
67     }
68
69     check(rc >= 0, "Failed to read from fd: %d", fd);
70
71     RingBuffer_commit_write(buffer, rc);
72
73     return rc;
74
75 error:
76     return -1;
77 }
78
79
80 int write_some(RingBuffer *buffer, int fd, int is_socket)
81 {
82     int rc = 0;
83     bstring data = RingBuffer_get_all(buffer);
84
85     check(data != NULL, "Failed to get from the buffer.");
86     check(bfindreplace(data, &NL, &CRLF, 0) == BSTR_OK, "Failed to replace NL.");
87
88     if(is_socket) {
89         rc = send(fd, bdata(data), blength(data), 0);
90     } else {
91         rc = write(fd, bdata(data), blength(data));
92     }
93
94     check(rc == blength(data), "Failed to write everything to fd: %d.", fd);

```

```

95  bdestroy(data);
96
97  return rc;
98
99  error:
100  return -1;
101 }
102
103
104 int main(int argc, char *argv[])
105 {
106     fd_set allreads;
107     fd_set readmask;
108
109     int socket = 0;
110     int rc = 0;
111     RingBuffer *in_rb = RingBuffer_create(1024 * 10);
112     RingBuffer *sock_rb = RingBuffer_create(1024 * 10);
113
114     check(argc == 3, "USAGE: netclient host port");
115
116     socket = client_connect(argv[1], argv[2]);
117     check(socket >= 0, "connect to %s:%s failed.", argv[1], argv[2]);
118
119     FD_ZERO(&allreads);
120     FD_SET(socket, &allreads);
121     FD_SET(0, &allreads);
122
123     while(1) {
124         readmask = allreads;
125         rc = select(socket + 1, &readmask, NULL, NULL, NULL);
126         check(rc >= 0, "select failed.");
127
128         if(FD_ISSET(0, &readmask)) {
129             rc = read_some(in_rb, 0, 0);
130             check_debug(rc != -1, "Failed to read from stdin.");
131         }
132
133         if(FD_ISSET(socket, &readmask)) {
134             rc = read_some(sock_rb, socket, 0);
135             check_debug(rc != -1, "Failed to read from socket.");
136         }
137
138         while(!RingBuffer_empty(sock_rb)) {
139             rc = write_some(sock_rb, 1, 0);
140             check_debug(rc != -1, "Failed to write to stdout.");
141         }
142
143         while(!RingBuffer_empty(in_rb)) {
144             rc = write_some(in_rb, socket, 1);
145             check_debug(rc != -1, "Failed to write to socket.");
146         }
147     }
148
149     return 0;
150
151 error:
152     return -1;
153 }

```

This code uses `select` to handle events from both `stdin` (file descriptor 0) and from the socket it uses to talk to a server. It uses `RingBuffers` to store the data and copy it around, and you can consider the functions `read_some` and `write_some` early prototypes for similar functions in the `RingBuffer` library.

In this little bit of code are quite a few networking functions you may not know. As you hit a function you don't know, look it up in the man pages and make sure you understand it. This one little file could actually get you to research all the APIs required to write a little server in C.

46.3 What You Should See

If you have everything building then the quickest way to test it is see if you can get a special file off learncodethehardway.org:

Source 150: Testing netclient

```

1  $
2  $ ./bin/netclient learncodethehardway.org 80
3  GET /ex45.txt HTTP/1.1
4  Host: learncodethehardway.org
5
6  HTTP/1.1 200 OK
7  Date: Fri, 27 Apr 2012 00:41:25 GMT
8  Content-Type: text/plain
9  Content-Length: 41
10 Last-Modified: Fri, 27 Apr 2012 00:42:11 GMT
11 ETag: 4f99eb63-29
12 Server: Mongrel2/1.7.5
13
14 Learn C The Hard Way, Exercise 45 works.
15 ^C
16 $

```

What I did there is I type in the syntax needed to make the HTTP request for the file `/ex45.txt`, then the `Host:` header line, then hit ENTER to get an empty line. I then get the response, with headers and the content. After that I just hit CTRL-c to exit.

46.4 How To Break It

This code definitely could have bugs, and currently in the draft of the book I'm going to have to keep working on this. In the meantime, try analyzing the code I have here and thrashing it against other servers. There's a tool called netcat that is great for setting up these kinds of servers. Another thing to do is use a language like Python or Ruby to create a simple "junk server" that spews out junk and bad data, closes connections randomly, and other nasty things.

If you find bugs, report them in the comments and I'll fix them up.

46.5 Extra Credit

1. As I mentioned, there's quite a few functions you may not know, so look them up. In fact, look them all up even if you think you know them.
2. Run this under valgrind and look for errors.
3. Go back through and add various defensive programming checks to the functions to improve them.
4. Use the `getopt` function to allow the user to give this the option to not translate `\n` to `\r\n`. This is only needed on protocols that require it for line endings, like HTTP. Sometimes you don't want the translation, so give the user an option.

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 47

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 46: Ternary Search Tree

My favorite algorithm, a TST creates a binary tree of the characters in a set of strings, then sets the terminating character of each string to a piece of data. This effectively creates a kind of "suffix array hash map" or "character binary tree". It's very useful in quickly matching a given string to a set of strings, such as in web application routing.

I show a simplified implementation used in Mongrel and Mongrel2 and how it works.

47.1 What You Should See

47.2 How To Break It

47.3 Extra Credit

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)

4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 48

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 47: Ternary Search Tree Algorithms

Test driven exercise to implement prefix matching and suffix matching algorithms.

Maybe also approximate matching.

48.1 What You Should See

48.2 How To Break It

48.3 Extra Credit

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 49

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 48: A Tiny Virtual Machine Part 1

The rest of the book will be implementing a version of the DCPU16 virtual machine using the algorithms created so far. This will be done in 5 parts so it's broken down and understandable. It will apply nearly everything taught so far.

49.1 What You Should See

49.2 How To Break It

49.3 Extra Credit

Take An Online Video Course

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 50

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 48: A Tiny Virtual Machine Part 2

50.1 What You Should See

50.2 How To Break It

50.3 Extra Credit

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 51

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 50: A Tiny Virtual Machine Part 3

51.1 What You Should See

51.2 How To Break It

51.3 Extra Credit

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 52

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 51: A Tiny Virtual Machine Part 4

52.1 What You Should See

52.2 How To Break It

52.3 Extra Credit

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 53

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

Exercise 52: A Tiny Virtual Machine Part 5

53.1 What You Should See

53.2 How To Break It

53.3 Extra Credit

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 54

Next Steps

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

After you read this book you should...

Take An Online Video Course

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Part III

Reviewing And Critiquing Code

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[tail\]](#) [\[up\]](#)

55 [Deconstructing "K&R C"](#)

55.1 [An Overall Critique Of Correctness](#)

55.2 [Chapter 1 Examples](#)

[\[next\]](#) [\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Chapter 55 Deconstructing "K&R C"

[\[prev\]](#)
[\[prev-tail\]](#)
[\[tail\]](#)
[\[up\]](#)

When I was a kid I read this awesome book called "The C Programming Language" by the language's creators, Brian Kernighan and Dennis Ritchie. This book taught me and many people of my generation, and a generation before, how to write C code. You talk to anyone, whether they know C or not, and they'll say, "You can't beat "K&R C" . It's the best C book." It is an established piece of programmer lore that is not soon to die.

I myself believed that until I started writing this book. You see, "K&R C" is actually riddled with bugs and bad style. Its age is no excuse. These were bugs when they wrote the first printing, and the 42nd printing. I hadn't actually realized just how bad most of the code was in this book and recommended it to many people. After reading through it for just an hour I decided that it needs to be taken down from its pedestal and relegated to history rather than vaunted as state of the art.

I believe it is time to lay this book to rest, but I want to use it as an exercise for you in finding hacks, attacks, defects, and bugs by going through "K&R C" to break all the code. That's right, you are going to destroy this sacred cow for me, and you're going to have no problem doing it. When you are done doing this, you will have a finely honed eye for defect. You will also have an informed opinion of the book's actual quality, and will be able to make your own decisions on how to use the knowledge it contains.

In this chapter we will use all the knowledge you've gained from this book, and spend it reviewing the code in "K&R C" . What we will do is take many pieces of code from the book, find all the bugs in it, and write a unit test that exercises the bugs. We'll then run this test under Valgrind to get statistics and data, and then we'll fix the bugs with a redesign.

This will obviously be a long chapter so I'm going to only do a handful of these and then I'm going have you do the rest. I'll provide a guide that is each page, with the code on it, and hints to the bugs that it has. Your job is to then tear that piece of code apart and try to think like an attacker trying to break the code.

Note 14: Warning For The Fanboys

As you read this, if you feel that I am being disrespectful to the authors, then that's not my intent. I respect the authors more than anything you know and owe them a debt of gratitude for writing their book. My criticisms here are both for educational purposes of teaching people modern C code, and to destroy the belief in their work as a item of worship that cannot be questioned.

However, if when you read this you have feelings of me insulting you then just stop reading. You will gain nothing from this chapter but personal grief because you've attached your identity to "K&R C" and my criticisms will only be taken personally.

55.1 An Overall Critique Of Correctness

The primary problem "K&R C" has is its view of "correctness" comes from the first system it was used on: Unix. In the world of Unix software programs have a particular set of properties:

1. Programs are started and then exit, making resource allocation easier.
2. Most functions are only called by other parts of the same program in set ways.
3. The inputs to the program are limited to "expert" restricted users.

In the context of this 1970's computing style, "K&R C" is actually correct. As long as only trusted people run complete cohesive programs that exit and clean up all their resources then their code is fine.

Where "K&R C" runs into problems is when the functions or code snippets are taken out of the book and used in other programs. Once you take many of these code snippets and try use them in some other program they fall apart. They then have blatant buffer overflows, bugs, and problems that a beginner will trip over.

Another problem is that software these days doesn't exit right away, but instead it stays running for long periods of time because they're servers, desktop applications and mobile applications. The old style of "leaving the cleanup to the OS" doesn't work in the modern world the way it did back in the day.

The final problem though is that no software lives in a vacuum anymore. Software is now frequently attacked by people over network connections in an attempt to gain special privilege or simple street cred. The idea that "nobody will ever do that" is dead, and actually that's probably the first thing somebody will do.

The best way to summarize the problem of "K&R C" "correctness" is with an example from English. Imagine if you have the pair of sentences, "Jack and Jill went up the hill. He fell down." Well, from context clues you know that "He" means Jack. However, if you have that sentence on its own it's not clear who "He" is. Now, if you put that sentence at the end of another sentence you can get an unclear pronoun reference: "Jack and Frank went up the hill. He fell down." Which "He" are we talking about in that sentence?

This is how the code in "K&R C" works. As long as that code is not used in other programs without serious analysis of the entire software then it works. The second you take many of the functions out and try to use them in other systems they fall apart. And, what's the point of a book full of code you can't actually use in your own programs?

55.1.1 A First Demonstration Defect

The following copy function is found in the very first chapter and is an example of copying two strings. Here's a new source file to demonstrate the defects in this function.

Source 151: exercise-1.9-1.c

```

1  #include <stdio.h>
2  #include <assert.h>
3  #include <stdlib.h>
4
5  #define MAXLINE 10 // in the book this is 1000
6
7  void copy(char to[], char from[])
8  {
9      int i;
10
11     i = 0;
12     while((to[i] = from[i]) != '\0')
13         ++i;
14 }
15
16 int main(int argc, char *argv[])
17 {
18     int i;
19
20     // use heap memory as many modern systems do
21     char *line = malloc(MAXLINE);
22     char *longest = malloc(MAXLINE);
23
24     assert(line != NULL && longest != NULL && "memory error");
25

```

```

26 // initialize it but make a classic "off by one" error
27 for(i = 0; i < MAXLINE; i++) {
28     line[i] = 'a';
29 }
30
31 // cause the defect
32 copy(longest, line);
33
34 free(line);
35 free(longest);
36
37 return 0;
38 }

```

In the above example, I'm doing something that is fairly common: switching from using stack allocation to heap allocation with malloc. What happens is, typically malloc returns memory from the heap, and so the bytes after it are not initialized. Then you see me use a loop to accidentally initialize it wrong. This is a common defect, and one of the reasons we avoided classic style C strings in this book. You could also have this bug in programs that read from files, sockets, or other external resources. It is a very common bug, probably the most common in the world.

Before the switch to heap memory, this program probably ran just fine because the stack allocated memory will probably have a '\0' character at the end on accident. In fact, it would appear to run fine almost always since it just runs and exits quickly.

What's the effect of running this new program with copy used wrong?

Source 152: exercise-1.9-1.c Valgrind Failures

```

1 $ make 1.9-1
2 cc 1.9-1.c -o 1.9-1
3 $ ./1.9-1
4 $
5 $ valgrind ./1.9-1
6 ==2162== Memcheck, a memory error detector
7 ==2162== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
8 ==2162== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with -h for copyright info
9 ==2162== Command: ./1.9-1
10 ==2162==
11 ==2162== Invalid read of size 1
12 ==2162== at 0x4005C0: copy (in /home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
13 ==2162== by 0x400651: main (in /home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
14 ==2162== Address 0x51b104a is 0 bytes after a block of size 10 alloc'd
15 ==2162== at 0x4C2815C: malloc (vg_replace_malloc.c:236)
16 ==2162== by 0x4005E6: main (in /home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
17 ==2162==
18 ==2162== Invalid write of size 1
19 ==2162== at 0x4005C3: copy (in /home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
20 ==2162== by 0x400651: main (in /home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
21 ==2162== Address 0x51b109a is 0 bytes after a block of size 10 alloc'd
22 ==2162== at 0x4C2815C: malloc (vg_replace_malloc.c:236)
23 ==2162== by 0x4005F4: main (in /home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
24 ==2162==
25 ==2162== Invalid read of size 1
26 ==2162== at 0x4005C5: copy (in /home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
27 ==2162== by 0x400651: main (in /home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
28 ==2162== Address 0x51b109a is 0 bytes after a block of size 10 alloc'd
29 ==2162== at 0x4C2815C: malloc (vg_replace_malloc.c:236)
30 ==2162== by 0x4005F4: main (in /home/zedshaw/projects/books/learn-c-the-hard-way/code/krc/1.9-1)
31 ==2162==
32 ==2162==
33 ==2162== HEAP SUMMARY:
34 ==2162== in use at exit: 0 bytes in 0 blocks
35 ==2162== total heap usage: 2 allocs, 2 frees, 20 bytes allocated
36 ==2162==
37 ==2162== All heap blocks were freed -- no leaks are possible
38 ==2162==
39 ==2162== For counts of detected and suppressed errors, rerun with: -v
40 ==2162== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 4 from 4)
41 $

```

As you've already learned, Valgrind will show you all of your sins in full color. In this case, a perfectly harmless seeming program has a ton of "Invalid read of size 1". If you kept running it you'd find other errors pop up at random.

Now, in the context of the entire program in the original "K&R C" example, this function will work correctly. However, the second this function is called with longest and line uninitialized, initialized wrong, without a trailing '\0' character, then you'll hit difficult to debug errors.

This is the failing of the book. While the code works in the book, it does not work in many other situations leading to difficult to spot defects, and those are the worst kind of defects for a beginner (or expert). Instead of code that only works in this delicate balance, we will strive to create code that has a higher probability of working in any situation.

55.1.2 Why copy() Fails

Many people have looked at this copy function and thought that it is not defective. They claim that, as long as it's used correctly, it is correct. One person even went so far as to say, "It's not defective, it's just unsafe." Odd, since I'm sure this person wouldn't get into a car if the manufacturer said, "Our car is not defective, it's just unsafe."

However, there is a way to formally prove that this function is defective by enumerating the possible inputs and then seeing if any of them cause the while loop to never terminate.

What we'll do is have two strings, A and B, and figure out what copy() does with them:

1. A = {'a','b','\0'}; B = {'a', 'b', '\0'}; copy(A,B);
2. A = {'a','b'}; B = {'a', 'b', '\0'}; copy(A,B);
3. A = {'a','b','\0'}; B = {'a', 'b'}; copy(A,B);
4. A = {'a','b'}; B = {'a', 'b'}; copy(A,B);

This is all the basic permutations of strings that can be passed to the function based on whether they are terminated with a '\0' or not. To be complete I'm covering all possible permutations, even if they seem irrelevant. You may think there's no need to include permutations on A, but as you'll see in the analysis, not including A fails to find buffer overflows that are possible.

We can then go through each case and determine if the while loop in copy() terminates:

1. while-loop finds '\0' in B, copy fits in A, terminates.
2. while-loop finds '\0' in B, overflows A, terminates.
3. while-loop does not find '\0' in B, overflows A, does not terminate.
4. while-loop does not find '\0' in B, overflows A, does not terminate.

This provides a formal proof that the function is defective because there are possible inputs that causes the while-loop to run forever or overflow the target. If you were to try and use this function safely, you would need to follow all paths to its usage, and confirm that the data is correct along every path. That gives every path to this function a 50% to 75% chance it will fail with just the inputs above. You could find some more permutations of failure but these are the most basic ones.

Let's now compare this to a copy function that knows the lengths of all the inputs to see what it's probability of failure is:

1. A = {'a','b','\0'}; B = {'a', 'b', '\0'}; safercopy(2, A, 2, B);
2. A = {'a','b'}; B = {'a', 'b', '\0'}; safercopy(2, A, 2, B);
3. A = {'a','b','\0'}; B = {'a', 'b'}; safercopy(2, A, 2, B);
4. A = {'a','b'}; B = {'a', 'b'}; safercopy(2, A, 2, B);

Also assume that the `safercopy()` function uses a `for`-loop that does not test for a `'\0'` only, but instead uses the given lengths to determine the amount to copy. With that we can then do the same analysis:

1. `for`-loop processes 2 characters of A, terminates.
2. `for`-loop processes 2 characters of A, terminates.
3. `for`-loop processes 2 characters of A, terminates.
4. `for`-loop processes 2 characters of A, terminates.

In every case the `for`-loop variant with string length given as arguments will terminate no matter what. To really test the `for`-loop variant we'd need to add some permutations for differing lengths of strings A and B, but in every case the `for`-loop will always stop because it will only go through a fixed previously known finite number of characters.

That means the `for`-loop will never loop forever, and as long as it handles all the possible differing lengths of A and B, never overflow either side. The only way to break `safercopy()` is to lie about the lengths of the strings, but even then it will still always terminate. The worst possible scenario for the `safercopy()` function is that you are given an erroneous length for one of the strings and that string does not have a `'\0'` properly, so the function buffer overflows.

This shows exactly why the `copy()` function is defective, because it does not terminate cleanly for most possible inputs, and is only reliable for one of the conditions: B terminated and A the right size. It also shows why a `for`-loop variant with a given fixed length for each input is superior.

Finally, the significance of this is that I've effectively done a formal proof (well, mostly formal) that shows what you should be doing to analyze code. Each function has to stand on its own and not have any defects such as `while`-loops that do not terminate. In the above discussion I've shown that the original "K&R C" is defective, and fatally so since there is no way to fix it given the inputs. There's no way from just a pointer to ask if a string is properly formed since the only way to test that is to scan it, and scanning it runs into this same problem.

55.1.3 But, That's Not A C String

Some folks then defend this function (despite the proof above) by claiming that the strings in the proof aren't C strings. They want to apply an artful dodge that says "the function is not defective because you aren't giving it the right inputs", but I'm saying the function is defective because most of the possible inputs cause it to crash the software.

The problem with this mindset is there's no way to confirm that a C string is valid. Imagine you wanted to write a little `assert_good_string` function that checks if a C string is correctly terminated before using it. This function needs to go to the end of the string and see if there's a `'\0'` terminator. How does it do this? This function would also have to scan the target function to confirm that it ended in `'\0'`, which means it has the same problem as `copy()` because the input may not be terminated.

This may seem silly, but people actually do this with `strlen()`. They take an input and think that they just have to run `strlen()` on the input to confirm that it's the right length, but `strlen()` itself has the same fatal flaw because it has to scan and if the string isn't terminated it will also overflow.

This means any attempt to fix the problem using just C strings also has this problem. The only way to solve it is to include the length of every string and use that to scan it.

If you can't validate a C string in your function, then your only choice is to do full code reviews manually. This introduces human error and no matter what you do the error will happen.

55.1.4 Just Don't Do That

Another argument in favor of this `copy()` function is when the proponents of "K&R C" state that you are "just supposed to not use bad strings". Despite the mountains of empirical evidence that this is impossible in C code, they are basically correct and that's what I'm teaching in this exercise. But, instead of saying "just don't do that by checking all possible inputs", I'm advocating "just don't do that by not using this kind of function". I'll explain further.

In order to confirm that all inputs to this function are valid I have to go through a code review process that involves this:

1. Find all the places the `copy()` function is called.
2. Trace backwards from that call point to where the inputs are created.
3. Confirm that the data is created correctly.
4. Follow the path from the creation point of the data to where it's used and confirm that no line of code alters the data.
5. Repeat this for all paths and all branches, including all loops and if-statements involving the data.

In my experience this is only possible in small programs like the little ones that "K&R C" has. In real software the number of possible branches you'd need to check is much too high for most people to validate, especially in a team environment where individuals have varying degrees of capability. A way to quantify this difficulty is that each branch in the code leading to a function like `copy()` has a 50–70% chance of causing the defect.

However, if you can use a different function and avoid all of these checks then doesn't that mean the `copy()` function is defective by comparison? These people are right, the solution is to "just not do that" by just not using the `copy()` function. You can change the function to one that includes the sizes of the two strings and the problem is solved. If that's the case then the people who think "just don't do that" have just proved that the function is defective, because the simpler way to "not do that" is to use a better function.

If you think `copy()` is valid as long as you avoid the errors I outline, and if `safercopy()` avoids the errors, then `safercopy()` is superior and `copy()` is defective by comparison.

55.1.5 Stylistic Issues

A more minor critique of the book is that the style is not only old, but just error prone and annoyingly "clever". Take the code you just saw again and look at the while-loop in `copy`. There's no reason to write this loop this way, as the compiler can just as easily work with a for-loop and without the clever triple-equality trick. The original code also has a while-loop without braces, but an if-statement with braces, which leads to even more confusion:

Source 153: Braces Are Free, Use Them

```

1  /* bad use of while loop with compound if-statement */
2  while ((len = getline(line, MAXLINE)) > 0)
3      if (len > max) {
4          max = len;
5          copy(longest, line);
6      }
7  if (max > 0) /* there was a line */
8      printf("%s", longest);

```

This code is incredibly error prone because you can't easily tell where the pair of if-statements and the while-loop are paired. A quick glance makes it seem like this while-loop will loop both if-statements, but it doesn't. In modern C code you would instead just use braces all the time and avoid the confusion completely.

While the book could be forgiven for this because of its age, it has been republished in this form 42 times, and it was updated for the ANSI standard. At some point in its history you'd think the authors or some publisher ghostwriter could

have been bothered to update the book's style. However, this is the problem with sacred cows. Once they become idols of worship people are reluctant to question them or modify them.

In the rest of this chapter though we will be modernizing the code in "K&R C" to fit the style you've been learning throughout this book. It will be more verbose, but it will be clearer and less error prone because of this slight increase in verbosity.

55.2 Chapter 1 Examples

Now we begin...

[\[prev\]](#) [\[prev-tail\]](#) [\[front\]](#) [\[up\]](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.

Learn C The Hard Way

A Learn Code The Hard Way Book

- [Book](#)
- [Comments](#)
- [Video Courses](#)
- [Related Books](#)

Learn C The Hard Way

A Clear & Direct Introduction To Modern C Programming

Zed A. Shaw

July 2011

[Contents](#)

[Preface](#)

[Introduction: The Cartesian Dream Of C](#)

[What You Will Learn](#)

[How To Read This Book](#)

[The Core Competencies](#)

[License](#)

I [Basic Skills](#)

1 [Exercise 0: The Setup](#)

1.1 [Linux](#)

1.2 [Mac OSX](#)

1.3 [Windows](#)

1.4 [Text Editor](#)

2 [Exercise 1: Dust Off That Compiler](#)

2.1 [What You Should See](#)

2.2 [How To Break It](#)

2.3 [Extra Credit](#)

3 [Exercise 2: Make Is Your Python Now](#)

3.1 [Using Make](#)

3.2 [What You Should See](#)

3.3 [How To Break It](#)

3.4 [Extra Credit](#)

4 [Exercise 3: Formatted Printing](#)

4.1 [What You Should See](#)

4.2 [External Research](#)

4.3 [How To Break It](#)

4.4 [Extra Credit](#)

5 [Exercise 4: Introducing Valgrind](#)

5.1 [Installing Valgrind](#)

5.2 [Using Valgrind](#)

- 5.3 [What You Should See](#)
- 5.4 [Extra Credit](#)
- 6 [Exercise 5: The Structure Of A C Program](#)
 - 6.1 [What You Should See](#)
 - 6.2 [Breaking It Down](#)
 - 6.3 [Extra Credit](#)
- 7 [Exercise 6: Types Of Variables](#)
 - 7.1 [What You Should See](#)
 - 7.2 [How To Break It](#)
 - 7.3 [Extra Credit](#)
- 8 [Exercise 7: More Variables, Some Math](#)
 - 8.1 [What You Should See](#)
 - 8.2 [How To Break It](#)
 - 8.3 [Extra Credit](#)
- 9 [Exercise 8: Sizes And Arrays](#)
 - 9.1 [What You Should See](#)
 - 9.2 [How To Break It](#)
 - 9.3 [Extra Credit](#)
- 10 [Exercise 9: Arrays And Strings](#)
 - 10.1 [What You Should See](#)
 - 10.2 [How To Break It](#)
 - 10.3 [Extra Credit](#)
- 11 [Exercise 10: Arrays Of Strings, Looping](#)
 - 11.1 [What You Should See](#)
 - 11.2 [How To Break It](#)
 - 11.3 [Extra Credit](#)
- 12 [Exercise 11: While-Loop And Boolean Expressions](#)
 - 12.1 [What You Should See](#)
 - 12.2 [How To Break It](#)
 - 12.3 [Extra Credit](#)
- 13 [Exercise 12: If, Else-If, Else](#)
 - 13.1 [What You Should See](#)
 - 13.2 [How To Break It](#)
 - 13.3 [Extra Credit](#)
- 14 [Exercise 13: Switch Statement](#)
 - 14.1 [What You Should See](#)
 - 14.2 [How To Break It](#)
 - 14.3 [Extra Credit](#)
- 15 [Exercise 14: Writing And Using Functions](#)
 - 15.1 [What You Should See](#)
 - 15.2 [How To Break It](#)
 - 15.3 [Extra Credit](#)
- 16 [Exercise 15: Pointers Dreaded Pointers](#)
 - 16.1 [What You Should See](#)
 - 16.2 [Explaining Pointers](#)
 - 16.3 [Practical Pointer Usage](#)
 - 16.4 [The Pointer Lexicon](#)

- 16.5 [How To Break It](#)
- 16.6 [Extra Credit](#)
- 17 [Exercise 16: Structs And Pointers To Them](#)
 - 17.1 [What You Should See](#)
 - 17.2 [Explaining Structures](#)
 - 17.3 [How To Break It](#)
 - 17.4 [Extra Credit](#)
- 18 [Exercise 17: Heap And Stack Memory Allocation](#)
 - 18.1 [What You Should See](#)
 - 18.2 [Heap vs. Stack Allocation](#)
 - 18.3 [How To Break It](#)
 - 18.4 [Extra Credit](#)
- 19 [Exercise 18: Pointers To Functions](#)
 - 19.1 [What You Should See](#)
 - 19.2 [How To Break It](#)
 - 19.3 [Extra Credit](#)
- 20 [Exercise 19: A Simple Object System](#)
 - 20.1 [How The CPP Works](#)
 - 20.2 [The Prototype Object System](#)
 - 20.3 [The Game Implementation](#)
 - 20.4 [What You Should See](#)
 - 20.5 [Auditing The Game](#)
 - 20.6 [Extra Credit](#)
- 21 [Exercise 20: Zed's Awesome Debug Macros](#)
 - 21.1 [The C Error Handling Problem](#)
 - 21.2 [The Debug Macros](#)
 - 21.3 [Using dbg.h](#)
 - 21.4 [What You Should See](#)
 - 21.5 [How The CPP Expands Macros](#)
 - 21.6 [Extra Credit](#)
- 22 [Exercise 21: Advanced Data Types And Flow Control](#)
 - 22.1 [Available Data Types](#)
 - 22.2 [Available Operators](#)
 - 22.3 [Available Control Structures](#)
- 23 [Exercise 22: The Stack, Scope, And Globals](#)
 - 23.1 [What You Should See](#)
 - 23.2 [Scope, Stack, And Bugs](#)
 - 23.3 [How To Break It](#)
 - 23.4 [Extra Credit](#)
- 24 [Exercise 23: Meet Duff's Device](#)
 - 24.1 [What You Should See](#)
 - 24.2 [Solving The Puzzle](#)
 - 24.3 [Extra Credit](#)
- 25 [Exercise 24: Input, Output, Files](#)
 - 25.1 [What You Should See](#)
 - 25.2 [How To Break It](#)
 - 25.3 [The I/O Functions](#)

25.4 [Extra Credit](#)

26 [Exercise 25: Variable Argument Functions](#)

26.1 [What You Should See](#)

26.2 [How To Break It](#)

26.3 [Extra Credit](#)

27 [Exercise 26: Write A First Real Program](#)

27.1 [What Is devpkg?](#)

27.2 [Project Layout](#)

27.3 [The Makefile](#)

27.4 [The Source Files](#)

27.5 [The Mid-Term Exam](#)

II [Data Structures And Algorithms](#)

28 [Exercise 27: Creative And Defensive Programming](#)

28.1 [The Creative Programmer Mindset](#)

28.2 [The Defensive Programmer Mindset](#)

28.3 [The Eight Defensive Programmer Strategies](#)

28.4 [Applying The Eight Strategies](#)

28.5 [Order Is Not Important](#)

28.6 [Extra Credit](#)

29 [Exercise 28: Intermediate Makefiles](#)

29.1 [The Basic Project Structure](#)

29.2 [Makefile](#)

29.3 [What You Should See](#)

29.4 [Extra Credit](#)

30 [Exercise 29: Libraries And Linking](#)

30.1 [What You Should See](#)

30.2 [How To Break It](#)

30.3 [Extra Credit](#)

31 [Exercise 30: Automated Testing](#)

31.1 [Wiring Up The Test Framework](#)

31.2 [Extra Credit](#)

32 [Exercise 31: Debugging Code](#)

32.1 [Debug Printing Vs. GDB Vs. Valgrind](#)

32.2 [A Debugging Strategy](#)

32.3 [Using GDB](#)

32.4 [Process Attaching](#)

32.5 [GDB Tricks](#)

32.6 [Extra Credit](#)

33 [Exercise 32: Double Linked Lists](#)

33.1 [What Are Data Structures](#)

33.2 [Making The Library](#)

33.3 [Double Linked Lists](#)

33.4 [Tests](#)

33.5 [What You Should See](#)

33.6 [How To Improve It](#)

33.7 [Extra Credit](#)

34 [Exercise 33: Linked List Algorithms](#)

- 34.1 [What You Should See](#)
- 34.2 [How To Improve It](#)
- 34.3 [Extra Credit](#)
- 35 [Exercise 34: Dynamic Array](#)
- 35.1 [Advantages And Disadvantages](#)
- 35.2 [How To Improve It](#)
- 35.3 [Extra Credit](#)
- 36 [Exercise 35: Sorting And Searching](#)
- 36.1 [Radix Sort And Binary Search](#)
- 36.2 [How To Improve It](#)
- 36.3 [Extra Credit](#)
- 37 [Exercise 36: Safer Strings](#)
- 37.1 [Why C Strings Were A Horrible Idea](#)
- 37.2 [Using bstrlib](#)
- 37.3 [Learning The Library](#)
- 38 [Exercise 37: Hashmaps](#)
- 38.1 [How To Improve It](#)
- 38.2 [Extra Credit](#)
- 39 [Exercise 38: Hashmap Algorithms](#)
- 39.1 [What You Should See](#)
- 39.2 [How To Break It](#)
- 39.3 [Extra Credit](#)
- 40 [Exercise 39: String Algorithms](#)
- 40.1 [What You Should See](#)
- 40.2 [Analyzing The Results](#)
- 40.3 [Extra Credit](#)
- 41 [Exercise 40: Binary Search Trees](#)
- 41.1 [How To Improve It](#)
- 41.2 [Extra Credit](#)
- 42 [Exercise 41: Using Cachegrind And Callgrind For Performance Tuning](#)
- 42.1 [Running Callgrind](#)
- 42.2 [Callgrind Annotating Source](#)
- 42.3 [Analyzing Memory Access With Cachegrind](#)
- 42.4 [Judo Tuning](#)
- 42.5 [Using KCachegrind](#)
- 42.6 [Extra Credit](#)
- 43 [Exercise 42: Stacks and Queues](#)
- 43.1 [What You Should See](#)
- 43.2 [How To Improve It](#)
- 43.3 [Extra Credit](#)
- 44 [Exercise 43: A Simple Statistics Engine](#)
- 44.1 [Rolling Standard Deviation And Mean](#)
- 44.2 [Implementation](#)
- 44.3 [How To Use It](#)
- 44.4 [Extra Credit](#)
- 45 [Exercise 44: Ring Buffer](#)
- 45.1 [The Unit Test](#)

- 45.2 [What You Should See](#)
- 45.3 [How To Improve It](#)
- 45.4 [Extra Credit](#)
- 46 [Exercise 45: A Simple TCP/IP Client](#)
- 46.1 [Augment The Makefile](#)
- 46.2 [The netclient Code](#)
- 46.3 [What You Should See](#)
- 46.4 [How To Break It](#)
- 46.5 [Extra Credit](#)
- 47 [Exercise 46: Ternary Search Tree](#)
- 47.1 [What You Should See](#)
- 47.2 [How To Break It](#)
- 47.3 [Extra Credit](#)
- 48 [Exercise 47: Ternary Search Tree Algorithms](#)
- 48.1 [What You Should See](#)
- 48.2 [How To Break It](#)
- 48.3 [Extra Credit](#)
- 49 [Exercise 48: A Tiny Virtual Machine Part 1](#)
- 49.1 [What You Should See](#)
- 49.2 [How To Break It](#)
- 49.3 [Extra Credit](#)
- 50 [Exercise 48: A Tiny Virtual Machine Part 2](#)
- 50.1 [What You Should See](#)
- 50.2 [How To Break It](#)
- 50.3 [Extra Credit](#)
- 51 [Exercise 50: A Tiny Virtual Machine Part 3](#)
- 51.1 [What You Should See](#)
- 51.2 [How To Break It](#)
- 51.3 [Extra Credit](#)
- 52 [Exercise 51: A Tiny Virtual Machine Part 4](#)
- 52.1 [What You Should See](#)
- 52.2 [How To Break It](#)
- 52.3 [Extra Credit](#)
- 53 [Exercise 52: A Tiny Virtual Machine Part 5](#)
- 53.1 [What You Should See](#)
- 53.2 [How To Break It](#)
- 53.3 [Extra Credit](#)
- 54 [Next Steps](#)
- III [Reviewing And Critiquing Code](#)
- 55 [Deconstructing "K&R C"](#)
- 55.1 [An Overall Critique Of Correctness](#)
- 55.2 [Chapter 1 Examples](#)

Take An Online Video Course

You can sign up for a video course at:

<http://www.udemy.com/learn-c-the-hard-way/>

This course is currently being built at the same time that the book is being built, but if you sign up now then you get early access to both the *videos* and *PDF* of the book.

Related Books

You might want to check out these other books in the series:

1. [Learn Ruby The Hard Way](#)
2. [Learn Regex The Hard Way](#)
3. [Learn SQL The Hard Way](#)
4. [Learn C The Hard Way](#)
5. [Learn Python The Hard Way](#)

I'll be referencing other books shortly.

Copyright 2011 Zed A. Shaw. All Rights Reserved.