

OBJECTIVE-C PROGRAMMING

THE BIG NERD RANCH GUIDE

AARON HILLEGASS



Big
nerd
ranch

Objective-C Programming: The Big Nerd Ranch Guide

by Aaron Hillegass

Copyright © 2011 Big Nerd Ranch, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, contact

Big Nerd Ranch, Inc.
154 Krog Street
Suite 100
Atlanta, GA 30307
(404) 478-9005
<http://www.bignerdranch.com/>
book-comments@bignerdranch.com

The 10-gallon hat with propeller logo is a trademark of Big Nerd Ranch, Inc.

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
<http://www.informit.com>

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

App Store, Apple, Cocoa, Cocoa Touch, Instruments, Interface Builder, iOS, iPad, iPhone, iTunes, iTunes Store, Mac, Mac OS, Objective-C, and Xcode are trademarks of Apple, Inc., registered in the U.S. and other countries.

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

ISBN-10 0321706285
ISBN-13 978-0321706287

Library of Congress Control Number: 2011931707

First printing, November 2011

Table of Contents

I. Getting Started	1
1. You and This Book	3
C and Objective-C	3
How this book works	4
How the life of a programmer works	4
2. Your First Program	7
Installing Apple's developer tools	7
Getting started with Xcode	7
Where do I start writing code?	9
How do I run my program?	13
So what is a program?	14
Don't stop	15
II. How Programming Works	17
3. Variables and Types	19
Types	19
A program with variables	20
Challenge	22
4. if/else	23
Boolean variables	24
else if	25
For the More Curious: Conditional (ternary) operator	25
Challenge	26
5. Functions	27
When should I use a function?	27
How do I write and use a function?	27
How functions work together	30
Local variables, frames, and the stack	32
Recursion	34
Looking at the frames in the debugger	35
return	37
Global and static variables	39
Challenge	40
6. Numbers	41
printf()	41
Integers	42
Tokens for displaying integers	42
Integer operations	44
Floating-point numbers	46
Tokens for displaying floating-point numbers	46
Functions for floating-point numbers	47
Challenge	47
7. Loops	49
The while loop	49
The for loop	50
break	51

continue	52
The do-while loop	53
Challenge	54
8. Addresses and Pointers	55
Getting addresses	55
Storing addresses in pointers	56
Getting the data at an address	57
How many bytes?	57
NULL	58
Stylish pointer declarations	59
Challenges	59
9. Pass By Reference	61
Writing pass-by-reference functions	62
Avoid dereferencing NULL	64
10. Structs	65
Challenge	66
11. The Heap	69
III. Objective-C and Foundation	73
12. Objects	75
Creating and using your first object	75
Message anatomy	77
Objects in memory	79
id	79
Challenge	80
13. More Messages	81
Nesting message sends	81
Multiple arguments	82
Sending messages to nil	82
Challenge	83
14. NSString	85
Challenge	86
15. NSArray	87
NSMutableArray	89
Challenges	90
16. Developer Documentation	93
Reference pages	94
Quick Help	96
Other options and resources	98
17. Your First Class	101
Accessor methods	103
Dot notation	104
Properties	105
self	106
Multiple files	106
Challenge	106
18. Inheritance	109
Overriding methods	112
super	113

Challenge	113
19. Object Instance Variables	115
Object ownership and ARC	117
Creating the Asset class	118
Adding a to-many relationship to Employee	119
Challenge	123
20. Preventing Memory Leaks	125
Retain cycles	127
Weak references	130
Zeroing of weak references	131
For the More Curious: Manual reference counting and ARC History	132
Retain count rules	134
21. Collection Classes	135
NSArray/NSMutableArray	135
Immutable objects	135
Sorting	136
Filtering	137
NSSet/NSMutableSet	138
NSDictionary/NSMutableDictionary	140
C primitive types	142
Collections and nil	142
Challenge	143
22. Constants	145
Preprocessor directives	145
#include and #import	146
#define	146
Global variables	147
enum	148
#define vs global variables	149
23. Writing Files with NSString and NSData	151
Writing an NSString to a file	151
NSError	152
Reading files with NSString	153
Writing an NSData object to a file	154
Reading an NSData from a file	155
24. Callbacks	157
Target-action	157
Helper objects	160
Notifications	163
Which to use?	164
Callbacks and object ownership	164
25. Protocols	167
26. Property Lists	171
Challenge	173
IV. Event-Driven Applications	175
27. Your First iOS Application	177
Getting started with iTahDoodle	177
BNRAppDelegate	179

Adding a C helper function	180
Objects in iTahDoodle	181
Model-View-Controller	182
The application delegate	183
Setting up views	184
Running on the iOS simulator	185
Wiring up the table view	186
Adding new tasks	189
Saving task data	189
For the More Curious: What about main()?	190
28. Your First Cocoa Application	191
Edit BNRDocument.h	192
A look at Interface Builder	193
Edit BNRDocument.xib	194
Making connections	198
Revisiting MVC	202
Edit BNRDocument.m	202
Challenges	204
V. Advanced Objective-C	205
29. init	207
Writing init methods	207
A basic init method	208
Using accessors	209
init methods that take arguments	210
Deadly init methods	215
30. Properties	217
Property attributes	218
Mutability	218
Lifetime specifiers	218
Advice on atomic vs. nonatomic	220
Key-value coding	221
Non-object types	222
31. Categories	225
32. Blocks	227
Defining blocks	227
Using blocks	228
Declaring a block variable	228
Assigning a block	229
Passing in a block	230
typedef	233
Return values	233
Memory management	234
The block-based future	235
Challenges	235
Anonymous block	235
NSNotificationCenter	236
VI. Advanced C	237
33. Bitwise Operations	239

Bitwise-OR	240
Bitwise-AND	241
Other bitwise operators	242
Exclusive OR	242
Complement	243
Left-shift	243
Right-shift	244
Using enum to define bit masks	245
More bytes	245
Challenge	245
34. C Strings	247
char	247
char *	248
String literals	250
Converting to and from NSString	251
Challenge	252
35. C Arrays	253
36. Command-Line Arguments	257
37. Switch Statements	261
Next Steps	263

Part I

Getting Started

You and This Book

Let's talk about you for a minute. You want to write applications for iOS or Mac OS X, but you haven't done much (or any) programming in the past. Your friends have raved about my other books (*iOS Programming: The Big Nerd Ranch Guide* and *Cocoa Programming for Mac OS X*), but they are written for experienced programmers. What should you do? Read this book.

There are similar books, but this one is the one you should read. Why? I've been teaching people how to write applications for iOS and the Mac for a long time now, and I've identified what you need to know at this point in your journey. I've worked hard to capture that knowledge and dispose of everything else. There is a lot of wisdom and very little fluff in this book.

My approach is a little unusual. Instead of simply trying to get you to understand the syntax of Objective-C, I'll show you how programming works and how experienced programmers think about it.

Because of this approach, I'm going to cover some heavy ideas early in the book. You should not expect this to be an easy read. In addition, nearly every idea comes with a programming experiment. This combination of learning concepts and immediately putting them into action is the best way to learn programming.

C and Objective-C

When you run a program, a file is copied from the file system into memory (RAM), and the instructions in that file are executed by your computer. Those instructions are inscrutable to humans. So, humans write computer programs in a programming language. The very lowest-level programming language is called *assembly code*. In assembly code, you describe every step that the CPU (the computer's brain) must take. This code is then transformed into *machine code* (the computer's native tongue) by an *assembler*.

Assembly language is tediously long-winded and CPU-dependent (because the brain of your latest iMac can be quite different from the brain of your well-loved, well-worn PowerBook). In other words, if you want to run the program on a different type of computer, you will need to rewrite the assembly code.

To make code that could be easily moved from one type of computer to another, we developed "high-level languages." With high-level languages, instead of thinking about a particular CPU, you could express the instructions in a general way, and a program (called a *compiler*) would transform that code into highly-optimized, CPU-specific machine code. One of these languages is C. C programmers write code in the C language, and a C compiler then converts the C code into machine code.

The C language was created in the early 1970s at AT&T. The Unix operating system, which is the basis for Mac OS X and Linux, was written in C with a little bit of assembly code for very low-level operations. The Windows operating system is also mostly written in C.

The Objective-C programming language is based on C, but it adds support for object-oriented programming. Objective-C is the programming language that is used to write applications for Apple's iOS and Mac OS X operating systems.

How this book works

In this book, you will learn enough of the C and Objective-C programming languages to learn to develop applications for the Mac or for iOS devices.

Why am I going to teach you C first? Every effective Objective-C programmer needs a pretty deep understanding of C. Also, a lot of the ideas that look complicated in Objective-C have very simple roots in C. I will often introduce an idea using C and then push you toward mastery of the same idea in Objective-C.

This book was designed to be read in front of a Mac. You will read explanations of ideas and carry out hands-on experiments that will illustrate those ideas. These experiments aren't optional. You won't really understand the book unless you do them. The best way to learn programming is to type in code, make typos, fix your typos, and become physically familiar with the patterns of the language. Just reading code and understanding the ideas in theory won't do much for you and your skills.

For even more practice, there are exercises called *Challenges* at the end of each chapter. These exercises provide additional practice and will make you more confident of what you've just learned. I strongly suggest you do as many of the *Challenges* as you can.

You will also see sections called *For the More Curious* at the end of some chapters. These are more in-depth explanations of the topics covered in the chapter. They are not absolutely essential to get you where you're going, but I hope you'll find them interesting and useful.

Big Nerd Ranch hosts a forum where readers discuss this book and the exercises in it. You can find it at <http://forums.bignerdranch.com/>.

You will find this book and programming in general much more pleasant if you know how to touch-type. Touch-typing, besides being much faster, enables you to look at your screen and book instead of at the keyboard. This makes it much easier to catch your errors as they happen. It is a skill that will serve you well for your entire career.

How the life of a programmer works

By starting this book, you've decided to become a programmer. You should know what you've signed up for.

The life of a programmer is mostly a never-ending struggle. Solving problems in an always-changing technical landscape means that programmers are always learning new things. In this case, "learning new things" is a euphemism for "battling against our own ignorance." Even if a programmer is working with a familiar technology, sometimes the software we create is so complex that simply understanding what's going wrong can often take an entire day.

If you write code, you will struggle. Most professional programmers learn to struggle hour after hour, day after day, without getting (too) frustrated. This is another skill that will serve you well. If you are

curious about the life of programmers and modern software projects, I highly recommend the book *Dreaming in Code* by Scott Rosenberg.

Now it's time to jump in and write your first program.

Your First Program

Now that we know how this book is organized, it's time to see how programming for the Mac and for iPhone and iPad works. To do that, you will

- install Apple's Developer Tools
- create a simple project using those tools
- explore how these tools are used to make sure our project works

At the end of this chapter, you will have successfully written your first program for the Mac.

Installing Apple's developer tools

To write applications for Mac OS X (the Macintosh) or iOS (the iPhone and iPad), you will be using Apple's developer tools. You can download these tools from <http://developer.apple.com/> or purchase them from the Mac App Store.

After you've installed the tools, find the `/Developer` folder at the root level of your hard drive. This folder contains what you need to develop applications for Mac OS X desktops and iOS mobile devices.

Our work in this book is going to be conducted almost entirely with one application – Xcode, which is found in the `/Developer/Applications` folder. (It is a good idea to drag the Xcode icon over to the dock; you'll be using it an awful lot.)

Getting started with Xcode

Xcode is Apple's *Integrated Development Environment*. That means that everything you need to write, build, and run new applications is in Xcode.

A note on terminology: anything that is executable on a computer we call a *program*. Some programs have graphical user interfaces; we will call these *applications*.

Some programs have no graphical user interface and run for days in the background; we call these *daemons*. Daemons sound scary, but they aren't. You probably have about 60 daemons running on your Mac right now. They are waiting around, hoping to be useful. For example, one of the daemons running on your system is called `pboard`. When you do a copy and paste, the `pboard` daemon holds onto the data that you are copying.

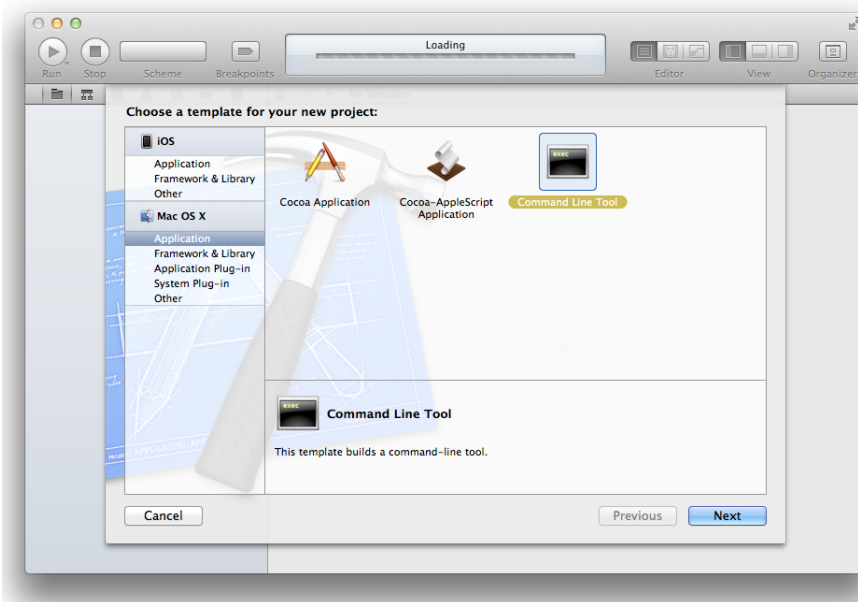
Some programs have no graphical user interface and run for a short time in the terminal; we call these *command-line tools*. In this book, you will be writing mostly command-line tools to focus on programming essentials without the distraction of creating and managing a user interface.

Now we're going to create a simple command-line tool using Xcode so you can see how it all works.

When you write a program, you create and edit a set of files. Xcode keeps track of those files in a *project*. Launch Xcode. From the File menu, choose New and then New Project....

To help you get started, Xcode suggests a number of possible project templates. You choose a template depending on what sort of program you want to write. In the lefthand column, select Application from the Mac OS X section. Then choose Command Line Tool from the choices that appear to the right.

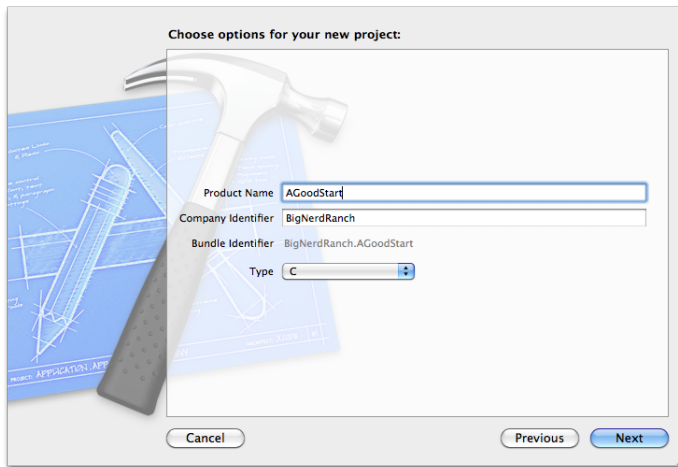
Figure 2.1 Choosing a template



Press the Next button.

Name your new project AGoodStart. The company identifier won't matter for our exercises in this book, but you have to enter one here to continue. You can use BigNerdRanch or another name. From the Type pop-up menu, choose C because you will write this program in C.

Figure 2.2 Choose options



Press the Next button.

Now choose the folder in which your project directory will be created. You won't need a repository for version control, so you can uncheck that box. Finally, click the Create button.

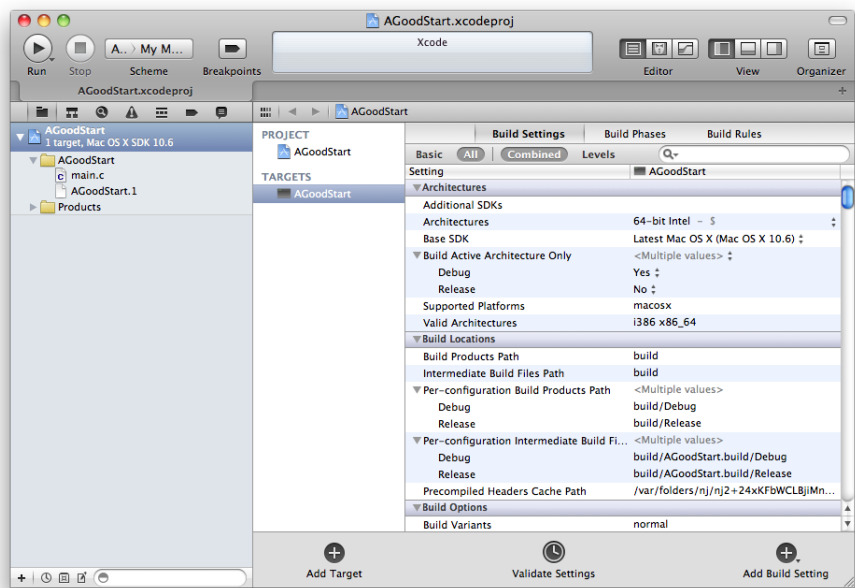
You'll be creating this same type of project for the next several chapters. In the future, I'll just say, "Create a new C Command Line Tool named *program-name-here*" to get you to follow this same sequence.

(Why C? Remember, Objective-C is built on top of the C programming language. You'll need to have an understanding of parts of C before we can get to the particulars of Objective-C.)

Where do I start writing code?

After creating your project, you'll be greeted by a window that shows how AGoodStart will be produced.

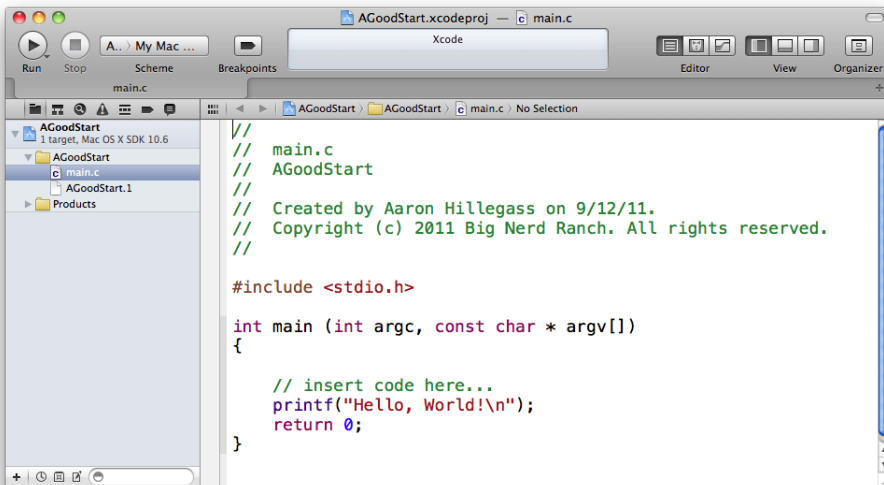
Figure 2.3 First view of the AGoodStart project



This window includes details like which versions of Mac OS X can run your application, the configurations to use when compiling the code that you write, and any localizations that have been applied to your project. But let’s ignore those details for now and find a simple starting point to get to work.

Near the top of the lefthand panel, find a file called `main.c` and click on it. (If you don’t see `main.c`, click the triangle next to the folder labeled `AGoodStart` to reveal its contents.)

Figure 2.4 Finding main.c in the AGoodStart group



Notice that our original view with the production details changes to show the contents of `main.c`. The `main.c` file contains a function called **main**.

A *function* is a list of instructions for the computer to execute, and every function has a name. In a C or Objective-C program, **main** is the function that is called when a program first starts.

```

#include <stdio.h>

int main (int argc, const char * argv[]) {
    // insert code here...
    printf("Hello, World!\n");
    return 0;
}

```

In this function, you'll find the two kinds of information you write in a program: code and comments.

- Code is the set of instructions that tell the computer to do something.
- Comments are ignored by the computer, but we programmers use them to document code we've written. The more difficult the programming problem you are trying to solve, the more comments will help document how you solved the problem. That becomes especially important when you return to your work months later, look at code you forgot to comment, and think, "I'm sure this solution is brilliant, but I have absolutely no memory of how it works."

In C and Objective-C, there are two ways to distinguish comments from code:

- If you put `//` at the beginning of a line of code, everything from those forward slashes to the end of that line is considered a comment. You can see this used in Apple’s “insert code here...” comment.
- If you have more extensive remarks in mind, you can use `/*` and `*/` to mark the beginning and end of comments that span more than one line.

These rules for marking comments are part of the *syntax* of C. Syntax is the set of rules that governs how code must be written in a given programming language. These rules are extremely specific, and if you fail to follow them, your program won’t work.

While the syntax regarding comments is fairly simple, the syntax of code can vary widely depending on what the code does and how it does it. But there’s one feature that remains consistent: every *statement* ends in a semicolon. (We’ll see examples of code statements in just a moment.) If you forget a semicolon, you will have made a syntax error, and your program won’t work.

Fortunately, Xcode has ways to warn you of these kinds of errors. In fact, one of the first challenges you will face as a programmer is interpreting what Xcode tells you when something goes wrong and then fixing your errors. You’ll get to see some of Xcode’s responses to common syntax errors as we go through the book.

Let’s make some changes to `main.c`. First, we need to make some space. Find the curly braces (`{` and `}`) that mark the beginning and the end of the `main` function. Then delete everything in between them.

Now update `main.c` to look like the code below. You’ll add a comment, two lines of code, and another comment to the `main` function. For now, don’t worry if you don’t understand what you are typing. The idea is to get started. You have an entire book ahead to learn what it all means.

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    // Print the beginning of the novel
    printf("It was the best of times.\n");
    printf("It was the worst of times.\n");
    /* Is that actually any good?
       Maybe it needs a rewrite. */

    return 0;
}
```

(Notice that the new code you need to type in is shown in a bold font. The code that isn’t bold is code that is already in place. That’s a convention we’ll use for the rest of the book.)


As you type, you may notice that Xcode tries to make helpful suggestions. This feature is called *code completion*, and it is very handy. You may want to ignore it right now and focus on typing things in all yourself. But as you continue through the book, start playing with code completion and how it can help you write code more conveniently and more accurately. You can see and set the different options for code completion in Xcode’s preferences, which are accessible from the Xcode menu.


In addition, keep an eye on the font color. Xcode uses different font colors to make it easy to identify comments and different parts of your code. (For example, comments are green.) This comes in handy, too: after a while of working with Xcode, you begin to instinctively notice when the colors don’t look right. Often, this is a clue that there is a syntax error in what you’ve written (like a forgotten semicolon). And the sooner you know that you’ve made a syntax error, the easier it is to find and fix it.

These color differences are just one way in which Xcode lets you know when you (may) have done something wrong.

How do I run my program?

When the contents of your `main.c` file match what you see above, it's time to run your program and see what it does. This is a two-step process. Xcode *builds* your program and then *runs* it. When building your program, Xcode prepares your code to run. This includes checking for syntax and other kinds of errors.

Look again at the lefthand area of the Xcode window. This area is called the navigator area. At the top of the navigator area is a series of buttons. You are currently viewing the *project navigator*, which shows you the files in your project. The project navigator's icon is .

Now find and click the  button to reveal the *log navigator*. The *log* is Xcode's way of communicating with you when it is building and running your program.

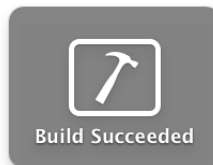
You can also use the log for your own purposes. For instance, the line in your code that reads

```
printf("It was the best of times.\n");
```

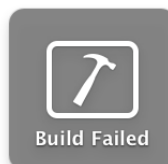
is an instruction to display the words "It was the best of times." in the log.

Since you haven't built and run your program yet, there isn't anything in the log navigator. Let's fix that. In the upper lefthand corner of the project window, find the button that looks suspiciously like the Play button in iTunes or on a DVD player. If you leave your cursor over that button, you'll see a tool tip that says **Build** and then run the current scheme. That is Xcode-speak for "Press this button, and I will build and run your program."

If all goes well, you'll be rewarded with the following:



If not, you'll get this:



What do you do then? Carefully compare your code with the code in the book. Look for typos and missing semicolons. Xcode will highlight the lines it thinks are problematic. After you find the problem, click the Run button again. Repeat until you have a successful build.

(Don't get disheartened when you have failed builds with this code or with any code you write in the future. Making and fixing mistakes helps you understand what you're doing. In fact, it's actually better than lucking out and getting it right the first time.)

Once your build has succeeded, find the item at the top of the log navigator labeled **Debug AGoodStart**. Click this item to display the log from the most recent run of your program.

The log can be quite verbose. The important part is the Dickens quote at the end. That's your code being executed!

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1705) (Tue Jul  5 07:36:45 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys001
[Switching to process 2723 thread 0x0]
It was the best of times.
It was the worst of times.
```

(As I'm writing this, Apple is working on a new debugger called LLDB. Eventually it will replace GDB, the current debugger. If you aren't seeing all the GDB information, it means that LLDB is now Xcode's standard debugger. The future must be a terrific place; I envy you.)

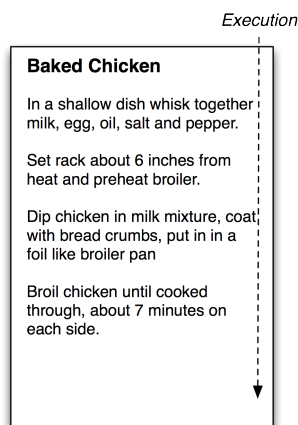
So what is a program?

Now that you've built and run your program, let's take a look inside. A program is a collection of functions. A function is a list of operations for the processor to execute. Every function has a name, and the function that you just wrote is named **main**. There was also another function – **printf**. You didn't write this function, but you did use it. (We'll find out where **printf** comes from in Chapter 5.)

To a programmer, writing a function is a lot like writing a recipe: "Stir a quart of water slowly until it boils. Then mix in a cup of flour. Serve while hot."

In the mid-1970's, Betty Crocker started selling a box containing a set of recipe cards. A recipe card is a pretty good metaphor for a function. Like a function, each card has a name and a set of instructions. The difference is that you execute a recipe, and the computer executes a function.

Figure 2.5 A recipe card named Baked Chicken



Betty Crocker's cooking instructions are in English. In the first part of this book, your functions will be written in the C programming language. However, a computer processor expects its instructions in machine code. How do we get there?

When you write a program in C (which is relatively pleasant for you), the *compiler* converts your program's functions into machine code (which is pleasant and efficient for the processor). The compiler is itself a program that is run by Xcode when you press the Run button. Compiling a program is the same as building a program, and we'll use these terms interchangeably.

When you run a program, the compiled functions are copied from the hard drive into memory, and the function called **main** is executed by the processor. The **main** function usually calls other functions. For example, your **main** function called the **printf** function. (We'll see more about how functions interact in Chapter 5.)

Don't stop

At this point, you've probably dealt with several frustrations: installation problems, typos, and lots of new vocabulary. And maybe nothing you've done so far makes any sense. That is completely normal.

As I write this, my son Otto is six. Otto is baffled several times a day. He is constantly trying to absorb knowledge that doesn't fit into his existing mental scaffolding. Bafflement happens so frequently, that it doesn't really bother him. He never stops to wonder, "Why is this so confusing? Should I throw this book away?"

As we get older, we are baffled much less often – not because we know everything, but because we tend to steer away from things that leave us bewildered. For example, reading a book on history is quite pleasant because we get nuggets of knowledge that we can hang from our existing mental scaffolding. This is easy learning.

Learning a new language is an example of difficult learning. You know that there are millions of people who work in that language effortlessly, but it seems incredibly strange and awkward in your mouth. And when people speak it to you, you are often flummoxed.

Learning to program a computer is also difficult learning. You will be baffled from time to time – especially here at the beginning. This is fine. In fact, it's kind of cool. It is a little like being six again.

Stick with this book; I promise that the bewilderment will cease before you get to the final page.

Part II

How Programming Works

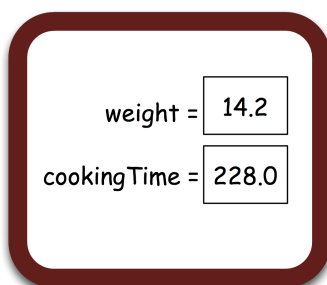
In these next chapters, you will create many programs that demonstrate useful concepts. These command-line programs are nothing that you'll show off to your friends, but there should be a small thrill of mastery when you run them. You're moving from computer user to computer programmer.

Your programs in these chapters will be written in C. Note that these chapters are not intended to cover the C language in detail. Quite the opposite: honed from years of teaching, this is the essential subset of what new-to-programming people need to know about programming and programming in C before learning Objective-C programming.

Variables and Types

Continuing with the recipe metaphor from the last chapter, sometimes a chef will keep a small blackboard in the kitchen for storing data. For example, when unpacking a turkey, he notices a label that says “14.2 Pounds.” Before he throws the wrapper away, he will scribble “weight = 14.2” on the blackboard. Then, just before he puts the turkey in the oven, he will calculate the cooking time (15 minutes + 15 minutes per pound) by referring to the weight on the blackboard.

Figure 3.1 Keeping track of data with a blackboard



During execution, a program often needs places to store data that will be used later. A place where one piece of data can go is known as a *variable*. Each variable has a name (like `cookingTime`) and a *type* (like a number). In addition, when the program executes, the variable will have a value (like 228.0).

Types

In a program, you create a new variable by *declaring* its type and name. Here’s an example of a variable declaration:

```
float weight;
```

The type of this variable is `float`, and its name is `weight`. At this point, the variable doesn’t have a value.

In C, you must declare the type of each variable for two reasons:

- The type lets the compiler check your work for you and alert you to possible mistakes or problems. For instance, say you have a variable of a type that holds text. If you ask for its logarithm, the compiler will tell you something like “It doesn’t make any sense to ask for this variable’s logarithm.”
- The type tells the compiler how much space in memory (how many bytes) to reserve for that variable.

Here is an overview of the commonly used types. We will return in more detail to each type in later chapters.

short, int, long	These three types are whole numbers; they don’t require a decimal point. A short usually has fewer bytes of storage than a long, and int is in between. Thus, you can store a much larger number in a long than in a short.
float, double	A float is a floating point number – a number that can have a decimal point. In memory, a float is stored as a mantissa and an exponent. For example, 346.2 is represented as 3.462×10^2 . A double is a double-precision number, which typically has more bits to hold a longer mantissa and larger exponents.
char	A char is a one-byte integer that we usually treat as a character, like the letter 'a'.
pointers	A pointer holds a memory address. It is declared using the asterisk character. For example, a variable declared as <code>int *</code> can hold a memory address where an int is stored. It doesn’t hold the actual number’s value, but if you know the address of the int then you can easily get to its value. Pointers are very useful, and there will be more on pointers later. Much more.
struct	A struct (or <i>structure</i>) is a type made up of other types. You can also create new struct definitions. For example, imagine that you wanted a <code>GeoLocation</code> type that contains two float members: <code>latitude</code> and <code>longitude</code> . In this case, you would define a struct type.

These are the types that a C programmer uses every day. It is quite astonishing what complex ideas can be captured in these five simple ideas.

A program with variables

Back in Xcode, you are going to create another project. First, close the `AGoodStart` project so that you don’t accidentally type new code into the old project.

Now create a new project (File → New → New Project...). This project will be a C Command Line Tool named `Turkey`.

In the project navigator, find this project’s `main.c` file and open it. Edit `main.c` so that it matches the following code.

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    // Declare the variable called 'weight' of type float
    float weight;

    // Put a number in that variable
    weight = 14.2;

    // Log it to the user
    printf("The turkey weighs %f.\n", weight);


    // Declare another variable of type float
    float cookingTime;

    // Calculate the cooking time and store it in the variable
    // In this case, '*' means 'multiplied by'
    cookingTime = 15.0 + 15.0 * weight;


    // Log that to the user
    printf("Cook it for %f minutes.\n", cookingTime);

    // End this function and indicate success

    return 0;
}
```

Build and run the program. You can either click the Run button at the top left of the Xcode window or use the keyboard shortcut Command-R. Then click the  button to get to the log navigator. Select the item at the top labeled Debug Turkey to show your output. It should look like this:

```
The turkey weighs 14.200000.
Cook it for 228.000000 minutes.
```

Now click the  button to return to the project navigator. Then select `main.c` so that you can see your code again. Let's review what you've done here.

In your line of code that looks like this:

```
float weight;
```

we say that you are “declaring the variable `weight` to be of type `float`.”

In the next line, your variable gets a value:

```
weight = 14.2;
```

You are copying data into that variable. We say that you are “assigning a value of 14.2 to that variable.”

In modern C, you can declare a variable and assign it an initial value in one line, like this:

```
float weight = 14.2;
```

Here is another assignment:

```
cookingTime = 15.0 + 15.0 * weight;
```

The stuff on the right-hand side of the `=` is an *expression*. An expression is something that gets evaluated and results in some value. Actually, every assignment has an expression on the right-hand side of the `=`.

For example, in this line:

```
weight = 14.2;
```

the expression is just `14.2`.

Variables are the building blocks of any program. This is just an introduction to the world of variables. You'll learn more about how variables work and how to use them as we continue.

Challenge

Create a new C Command Line Tool named `TwoFloats`. In its `main()` function, declare two variables of type `float` and assign each of them a number with a decimal point, like `3.14` or `42.0`. Declare another variable of type `double` and assign it the sum of the two `floats`. Print the result using `printf()`. Refer to the code in this chapter if you need to check your syntax.

4

if/else

An important idea in programming is taking different actions depending on circumstances. Have all the billing fields in the order form been filled out? If so, enable the Submit button. Does the player have any lives left? If so, resume the game. If not, show the picture of the grave and play the sad music.

This sort of behavior is implemented using `if` and `else`, the syntax of which is:

```
if (conditional) {  
    // execute this code if the conditional evaluates to true  
} else {  
    // execute this code if the conditional evaluates to false  
}
```

You won't create a project in this chapter. Instead, consider the code examples carefully based on what you've learned in the last two chapters.

Here's an example of code using `if` and `else`:

```
float truckWeight = 34563.8;  
  
// Is it under the limit?  
if (truckWeight < 40000.0) {  
    printf("It is a light truck\n");  
} else {  
    printf("It is a heavy truck\n");  
}
```

If you don't have an `else` clause, you can just leave that part out:

```
float truckWeight = 34563.8;  
  
// Is it under the limit?  
if (truckWeight < 40000.0) {  
    printf("It is a light truck\n");  
}
```

The conditional expression is always either true or false. In C, it was decided that 0 would represent false, and anything that is not zero would be considered true.

In the conditional in the example above, the `<` operator takes a number on each side. If the number on the left is less than the number on the right, the expression evaluates to 1 (a very common way of expressing trueness). If the number on the left is greater than or equal to the number on the right, the expression evaluates to 0 (the only way to express falseness).

Operators often appear in conditional expressions. Table 4.1 shows the common operators used when comparing numbers (and other types that the computer evaluates as numbers):

Table 4.1 Comparison operators

<	Is the number on the left less than the number on the right?
>	Is the number on the left greater than the number on the right?
<=	Is the number on the left less than or equal to the number on the right?
>=	Is the number on the left greater than or equal to the number on the right?
==	Are they equal?
!=	Are they <i>not</i> equal?

The == operator deserves an additional note: In programming, the == operator is what’s used to check for equality. We use the single = to *assign* a value. Many, many bugs have come from programmers using = when they meant to use ==. So stop thinking of = as “the equals sign.” From now on, it is “the assignment operator.”

Some conditional expressions require logical operators. What if you want to know if a number is in a certain range, like greater than zero and less than 40,000? To specify a range, you can use the logical AND operator (&&):

```
if ((truckWeight > 0.0) && (truckWeight < 40000.0)) {  
    printf("Truck weight is within legal range.\n");  
}
```

Table 4.2 shows the three logical operators:

Table 4.2 Logical operators

&&	Logical AND -- true if and only if both are true
	Logical OR -- false if and only if both are false
!	Logical NOT -- true becomes false, false becomes true

(If you are coming from another language, note that there is no logical exclusive OR in Objective-C, so we won’t discuss it here.)

The logical NOT operator (!) negates the expression contained in parentheses to its right.

```
// Is it not in the legal range?  
if (!((truckWeight > 0.0) && (truckWeight < 40000.0))) {  
    printf("Truck weight is not within legal range.\n");  
}
```

Boolean variables

As you can see, expressions can become quite long and complex. Sometimes it is useful to put the value of the expression into a handy, well-named variable.

```
B00L isNotLegal = !((truckWeight > 0.0) && (truckWeight < 40000.0));  
if (isNotLegal) {  
    printf("Truck weight is not within legal range.\n");  
}
```

A variable that can be true or false is a *boolean* variable. Historically, C programmers have always used an int to hold a boolean value. Objective-C programmers typically use the type B00L for boolean variables, so that’s what we use here. (B00L is just an alias for an integer type.)

A syntax note: if the code that follows the conditional expression consists of only one statement, then the curly braces are optional. So the following code is equivalent to the previous example.

```
BOOL isNotLegal = !((truckWeight > 0.0) && (truckWeight < 40000.0));
if (isNotLegal)
    printf("Truck weight is not within legal range.\n");
```

However, the curly braces are necessary if the code consists of more than one statement.

```
BOOL isNotLegal = !((truckWeight > 0.0) && (truckWeight < 40000.0));
if (isNotLegal) {
    printf("Truck weight is not within legal range.\n");
    printf("Impound truck.\n");
}
```

Why? Imagine if you removed the curly braces.

```
BOOL isNotLegal = !((truckWeight > 0.0) && (truckWeight < 40000.0));
if (isNotLegal)
    printf("Truck weight is not within legal range.\n");
    printf("Impound truck.\n");
```

This code would make you very unpopular with truck drivers. In this case, every truck gets impounded regardless of weight. When the compiler doesn't find a curly brace after the conditional, only the next statement is considered part of the if construct. Thus, the second statement is always executed. (What about the indentation of the second statement? While indentation is very helpful for human readers of code, it means nothing to the compiler.)

else if

What if you have more than two possibilities? You can test for them one-by-one using `else if`. For example, imagine that a truck belongs to one of three weight categories: floating, light, and heavy.

```
if (truckWeight <= 0) {
    printf("A floating truck\n");
} else if (truckWeight < 40000.0) {
    printf("A light truck\n");
} else {
    printf("A heavy truck\n");
}
```

You can have as many `else if` clauses as you wish. They will each be tested in the order in which they appear until one evaluates as true. The “in the order in which they appear” part is important. Be sure to order your conditions so that you don't get a false positive. For instance, if you swapped the first two tests in the above example, you would never find a floating truck because floating trucks are also light trucks. The final `else` clause is optional, but it's useful when you want to catch everything that did not meet the earlier conditions.

For the More Curious: Conditional (ternary) operator

It is not uncommon that you will use `if` and `else` to set the value of an instance variable. For example, you might have the following code:

```
int minutesPerPound;
if (isBoneless)
    minutesPerPound = 15;
else
    minutesPerPound = 20;
```

Whenever you have a scenario where a value is assigned to a variable based on a conditional, you have a candidate for the *conditional operator*, which is `?`. (You will sometimes see it called the *ternary operator*).

```
int minutesPerPound = isBoneless ? 15 : 20;
```

This one line is equivalent to the previous example. Instead of writing `if` and `else`, you write an assignment. The part before the `?` is the conditional. The values after the `?` are the alternatives for whether the conditional is found to be true or false.

If this notation strikes you as odd, there is nothing wrong with continuing to use `if` and `else` instead. I suspect over time you will embrace the ternary operator as a concise way to do conditional value assignment. More importantly, you will see it used by other programmers, and it will be nice to understand what you see!

Challenge

Consider the following code snippet:

```
int i = 20;
int j = 25;

int k = ( i > j ) ? 10 : 5;

if ( 5 < j - k ) { // first expression
    printf("The first expression is true.");
} else if ( j > i ) { // second expression
    printf("The second expression is true.");
} else {
    printf("Neither expression is true.");
}
```

What will be printed to the console?

5

Functions

Back in Chapter 3, I introduced the idea of a variable: a name associated with a chunk of data. A function is a name associated with a chunk of code. You can pass information to a function. You can make the function execute code. You can make a function return information to you.

Functions are fundamental to programming, so there's a lot in this chapter – three new projects, a new tool, and many new ideas. Let's get started with an exercise that demonstrates what functions are good for.

When should I use a function?

Suppose you are writing a program to congratulate students for completing a Big Nerd Ranch course. Before worrying about retrieving the student list from a database or about printing certificates on spiffy Big Nerd Ranch paper, you want to experiment with the message that will be printed on the certificates.

To do that experiment, create a new project: a C Command Line Tool named `ClassCertificates`.

Your first thought in writing this program might be:

```
int main (int argc, const char * argv[])
{
    printf("Mark has done as much Cocoa Programming as I could fit into 5 days\n");
    printf("Bo has done as much Objective-C Programming as I could fit into 2 days\n");
    printf("Mike has done as much Python Programming as I could fit into 5 days\n");
    printf("Ted has done as much iOS Programming as I could fit into 5 days\n");

    return 0;
}
```

Does the thought of typing all this in bother you? Does it seem annoyingly repetitive? If so, you have the makings of an excellent programmer. When you find yourself repeating work that is very similar in nature (in this case, the words in the `printf` statement), you want to start thinking about a function as a better way of accomplishing the same task.

How do I write and use a function?

Now that you've realized that you need a function, you need to write one. Open `main.c` in your `ClassCertificates` project and add a new function before the `main` function. Name this function `congratulateStudent`.

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

```
void congratulateStudent(char *student, char *course, int numDays)
{
    printf("%s has done as much %s Programming as I could fit into %d days.\n",
           student, course, numDays);
}
```

(Wondering what the %s and %d mean? Hold on for now; we'll talk about those in the next chapter.)

Now edit **main** to use your new function:

```
int main (int argc, const char * argv[])
{
    congratulateStudent("Mark", "Cocoa", 5);
    congratulateStudent("Bo", "Objective-C", 2);
    congratulateStudent("Mike", "Python", 5);
    congratulateStudent("Ted", "iOS", 5);

    return 0;
}
```

Build and run the program. You will probably get a *warning* marked by an exclamation point inside a small yellow triangle. A warning in Xcode will not prevent your program from running; it just draws your attention to a possible problem. The text of the warning is to the right of the code. This warning says something like No previous prototype for function 'congratulateStudent'. Ignore this warning for now, and we'll come back to it at the end of this section.

Find your output in the log navigator. It should be identical to what you would have seen if you had typed in everything yourself.

```
Mark has done as much Cocoa Programming as I could fit into 5 days.
Bo has done as much Objective-C Programming as I could fit into 2 days.
Mike has done as much Python Programming as I could fit into 5 days.
Ted has done as much iOS Programming as I could fit into 5 days.
```

Think about what you have done here. You noticed a repetitive pattern. You took all the shared characteristics of the problem (the repetitive text) and moved them into a separate function. That left the differences (student name, course name, number of days). You handled those differences by adding three *parameters* to the function. Let's look again at the line where you name the function.

```
void congratulateStudent(char *student, char *course, int numDays)
```

Each parameter has two parts: the type of data the argument represents and the name of the parameter. Parameters are separated by commas and placed in parentheses to the right of the name of the function.

What about the void to the left of our function name? That is the type of information returned from the function. When you do not have any information to return, you use the keyword void. We'll talk more about returning later in this chapter.

You also used, or *called*, your new function in **main**. When you called **congratulateStudent**, you passed it values. Values passed to a function are known as *arguments*. The argument's value is then assigned to the corresponding parameter name. That parameter name can be used inside the function as a variable that contains the passed-in value.

Let's get specific. In the first call to **congratulateStudent**, you pass three arguments: "Mark", "Cocoa", 5.

```
congratulateStudent("Mark", "Cocoa", 5);
```

For now, we'll focus on the third argument. When 5 is passed to **congratulateStudent**, it is assigned to the third parameter, `numDays`. Arguments and parameters are matched up in the order in which they appear. They must also be the same (or very close to the same) type. Here, 5 is an integer value, and the type of `numDays` is `int`. Good.

Now, when **congratulateStudent** uses, or *references*, the `numDays` variable within the function, its value will be 5. You can see `numDays` is referenced just before the semi-colon. Finally, you can prove that all of this worked by looking at the first line of the output, which correctly displays the number of days.

Look back to our first proposed version of `ClassCertificates` with all the repetitive typing. What's the point of using a function instead? To save on the typing? Well, yes, but that's definitely not all. It's also about error-checking. The less you type and the more the computer crunches, the fewer chances for typos. Also if you do mistype a function name, Xcode will alert you, but Xcode has no idea if you've mistyped text.

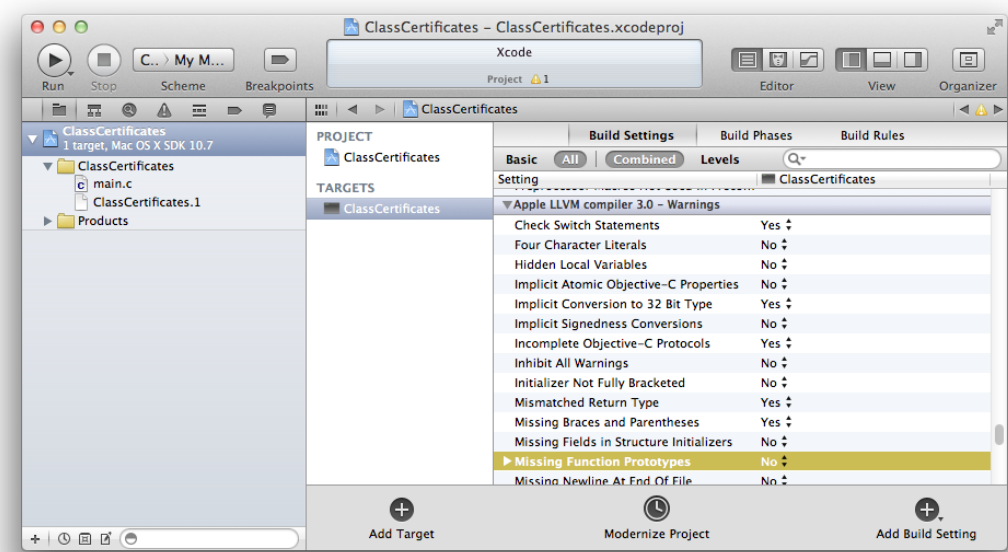
Another benefit to writing functions is reusability. Now that you've written this handy function, you could use it in another program. Making changes is simpler, too. You only need to adjust the wording of the congratulatory phrase in one place for it to take effect everywhere.

The final benefit of functions is if there is a "bug," you can fix that one function and suddenly everything that calls it will start working properly. Partitioning your code into functions makes it easier to understand and maintain.

Now back to that warning in your code. It is pretty common to *declare* a function in one place and *define* it in another. Declaring a function just warns the compiler that a function with a particular name is coming. Defining the function is where you describe the steps that should be executed. In this exercise, you actually declared and defined your function in the same place. Because this is uncommon, Xcode issues a warning if your function was not declared in advance.

It is OK to ignore this warning in any of the projects you build in this book. Or you can take the time to disable it. To do so, select the `ClassCertificates` target, which is the item at the top of the project navigator. In the editor pane, select **All** under the **Build Settings** tab. Scroll through the different build settings and find the **Missing Function Prototypes** setting. Change this setting to **No**.

Figure 5.1 Disabling Missing Function Prototypes warning

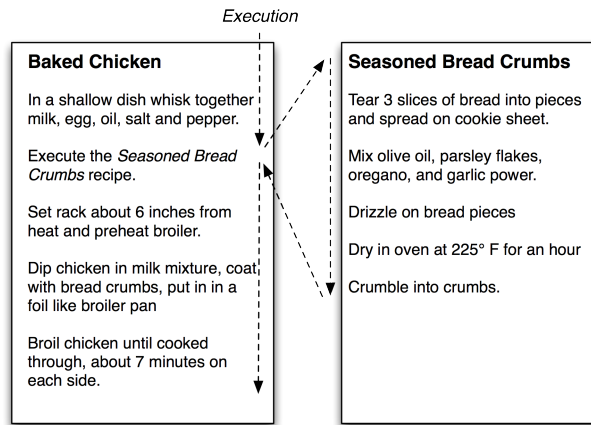


How functions work together

A *program* is a collection of functions. When you run a program, those functions are copied from the hard drive into memory, and the processor finds the function called “main” and executes it.

Remember that a function is like a recipe card. If I began to execute the “Baked Chicken” card, I would discover that the second instruction is “Make Seasoned Bread Crumbs,” which is explained on another card. A programmer would say “the Baked Chicken function *calls* the Seasoned Bread Crumbs function.”

Figure 5.2 Recipe cards



Similarly, the **main** function can call other functions. For example, your **main** function in **ClassCertificates** called the **congratulateStudent** function, which in turn called **printf**.

While you are preparing the seasoned bread crumbs, you stop executing the “Baked Chicken” card. When the bread crumbs are ready, you resume working through the “Baked Chicken” card.

Similarly, the **main** function stops executing and “blocks” until the function it calls is done executing. To see this happen, we’re going to call a **sleep** function that does nothing but wait a number of seconds. In your **main** function, add a call to **sleep**.

```
int main (int argc, const char * argv[])
{
    congratulateStudent("Mark", "Cocoa", 5);
    sleep(2);
    congratulateStudent("Bo", "Objective-C", 2);
    sleep(2);
    congratulateStudent("Mike", "PHP and PostgreSQL", 5);
    sleep(2);
    congratulateStudent("Ted", "iOS", 5);

    return 0;
}
```

Build and run the program. (Ignore the warning about an implicit declaration for now.) You should see a 2-second pause between each message of congratulations. That’s because the **main** function stops running until the **sleep** function is done sleeping.

Notice that when you call a function, you use its name and a pair of parentheses for its arguments. Thus, when we talk about functions, we usually include a pair of empty parentheses. From now on, we will say **main()** when we talk about the **main** function.

Your computer came with many functions built-in. Actually, that is a little misleading – here is the truth: Before Mac OS X was installed on your computer, it was nothing but an expensive space

heater. Among the things that were installed as part of Mac OS X were files containing a collection of precompiled functions. These collections are called *the standard libraries*. **sleep()** and **printf()** are included in these standard libraries.

At the top of `main.c`, you included the file `stdio.h`. This file contains a declaration of the function **printf()** and lets the compiler check to make sure that you are using it correctly. The function **sleep()** is declared in `stdlib.h`. Include that file, too, so that the compiler will stop complaining that **sleep()** is implicitly declared:

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

void congratulateStudent(char *student, char *course, int numDays)
{
    ...
}
```

The standard libraries serve two functions:

- They represent big chunks of code that you don't need to write and maintain. Thus, they empower you to build much bigger, better programs than you would be able to do otherwise.
- They ensure that most programs look and feel similar.

Programmers spend a lot of time studying the standard libraries for the operating systems that they work on. Every company that creates an operating system also has documentation for the standard libraries that come with it. You'll learn how to browse the documentation for iOS and Mac OS X in Chapter 16.

Local variables, frames, and the stack

Every function can have *local variables*. Local variables are variables declared inside a function. They exist only during the execution of that function and can only be accessed from within that function. For example, imagine that you were writing a function that computed how long to cook a turkey. It might look like this:

```
void showCookTimeForTurkey(int pounds)
{
    int necessaryMinutes = 15 + 15 * pounds;
    printf("Cook for %d minutes.\n", necessaryMinutes);
}
```

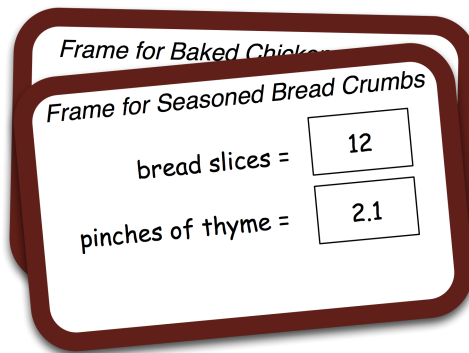
`necessaryMinutes` is a local variable. It came into existence when **showCookTimeForTurkey()** started to execute and will cease to exist once that function completes execution. The parameter of the function, `pounds`, is also a local variable. A parameter is a local variable that has been initialized to the value of the corresponding argument.

A function can have many local variables, and all of them are stored in the *frame* for that function. Think of the frame as a blackboard that you can scribble on while the function is running. When the function is done executing, the blackboard is discarded.

Imagine for a moment that you are working on the Baked Chicken recipe. In your kitchen, all recipes get their own blackboards, so you have a blackboard for the Baked Chicken recipe ready. Now, when

you call the Seasoned Bread Crumbs recipe, you need a new blackboard. Where are you going to put it? Right on top of the blackboard for Baked Chicken. After all, you've suspended execution of Baked Chicken to make Seasoned Bread Crumbs. You won't need the Baked Chicken frame until the Seasoned Bread Crumbs recipe is complete and its frame is discarded. What you have now is a stack of frames.

Figure 5.3 Two blackboards in a stack



Programmers say, “When a function is called, its frame is created on top of *the stack*. When the function finishes executing, its frame is popped off the stack and destroyed.”

Let's look more closely at how the stack works by putting `showCookTimeForTurkey()` into a hypothetical program:

```
void showCookTimeForTurkey(int pounds)
{
    int necessaryMinutes = 15 + 15 * pounds;
    printf("Cook for %d minutes.\n", necessaryMinutes);
}

int main(int argc, const char * argv[])
{
    int totalWeight = 10;
    int gibletsWeight = 1;
    int turkeyWeight = totalWeight - gibletsWeight;
    showCookTimeForTurkey(turkeyWeight);
    return 0;
}
```

Recall that `main()` is always executed first. `main()` calls `showCookTimeForTurkey()`, which begins executing. What, then, does this program's stack look like just after `pounds` is multiplied by 15?

Figure 5.4 Two frames on the stack

<code>showTurkeyCookTime()</code>	<code>pounds = 9</code> <code>necessaryMinutes = 135</code>
<code>main()</code>	<code>totalWeight = 10</code> <code>gibletsWeight = 1</code> <code>turkeyWeight = 9</code>

The stack is always last-in, first-out. That is, the `showCookTimeForTurkey()` will pop its frame off the stack before `main()` pops its frame off the stack.

Notice that `pounds`, the single parameter of `showCookTimeForTurkey()`, is part of the frame. Recall that a parameter is a local variable that has been assigned the value of the corresponding argument. For this example, the variable `turkeyWeight` with a value of 9 is passed as an argument to `showCookTimeForTurkey()`. Then that value is assigned to the parameter `pounds` and copied to the function's frame.

Recursion

Can a function call itself? You bet! We call that *recursion*. There is a notoriously dull song called “99 Bottles of Beer.” Create a new C Command Line Tool named `BeerSong`. Open `main.c` and add a function to write out the words to this song and then kick it off in `main()`:

```
#include <stdio.h>

void singTheSong(int numberOfBottles)
{
    if (numberOfBottles == 0) {
        printf("There are simply no more bottles of beer on the wall.\n");
    } else {
        printf("%d bottles of beer on the wall. %d bottles of beer.\n",
               numberOfBottles, numberOfBottles);
        int oneFewer = numberOfBottles - 1;
        printf("Take one down, pass it around, %d bottles of beer on the wall.\n",
               oneFewer);
        singTheSong(oneFewer); // This function calls itself!
        printf("Put a bottle in the recycling, %d empty bottles in the bin.\n",
               numberOfBottles);
    }
}

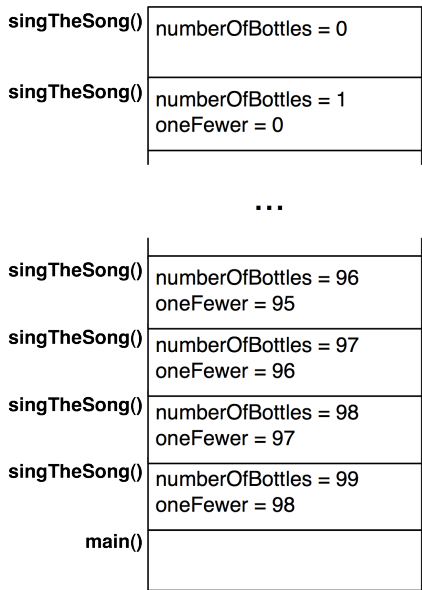
int main(int argc, const char * argv[])
{
    singTheSong(99);
    return 0;
}
```

Build and run the program. The output looks like this:

99 bottles of beer on the wall. 99 bottles of beer.
Take one down, pass it around, 98 bottles of beer on the wall.
98 bottles of beer on the wall. 98 bottles of beer.
Take one down, pass it around, 97 bottles of beer on the wall.
97 bottles of beer on the wall. 97 bottles of beer.
...
1 bottles of beer on the wall. 1 bottles of beer.
Take one down, pass it around, 0 bottles of beer on the wall.
There are simply no more bottles of beer on the wall.
Put a bottle in the recycling, 1 empty bottles in the bin.
Put a bottle in the recycling, 2 empty bottles in the bin.
...
Put a bottle in the recycling, 98 empty bottles in the bin.
Put a bottle in the recycling, 99 empty bottles in the bin.

What does the stack look like when the last bottle is taken off the wall?

Figure 5.5 Frames on the stack for a recursive function



Discussing the frames and the stack is usually not covered in a beginning programming course, but I’ve found the ideas to be exceedingly useful to new programmers. First, it gives you a more concrete understanding of the answers to questions like “What happens to my local variables when the function finishes executing?” Second, it helps you understand the *debugger*. The debugger is a program that helps you understand what your program is actually doing, which, in turn, helps you find and fix “bugs” (problems in your code). When you build and run a program in Xcode, the debugger is *attached* to the program so that you can use it.

Looking at the frames in the debugger

You can use the debugger to browse the frames on the stack. To do this, however, you have to stop your program in mid-execution. Otherwise, `main()` will finish executing, and there won’t be any

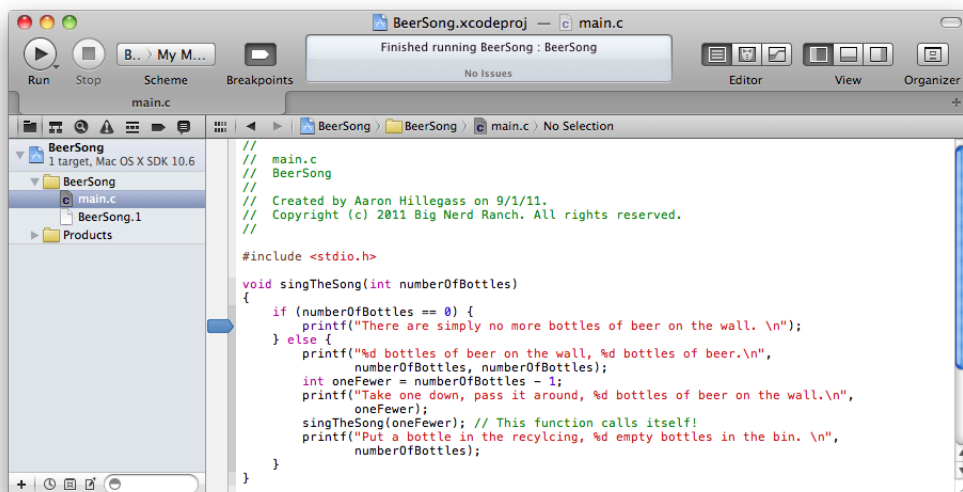
frames left to look at. To see as many frames as possible in our `BeerSong` program, we want to halt execution on the line that prints “There are simply no more bottles of beer on the wall.”

How do we do this? In `main.c`, find the line


```
printf("There are simply no more bottles of beer on the wall.\n");
```

There are two shaded columns to the left of your code. Click on the wider, lefthand column next to this line of code.

Figure 5.6 Setting a breakpoint



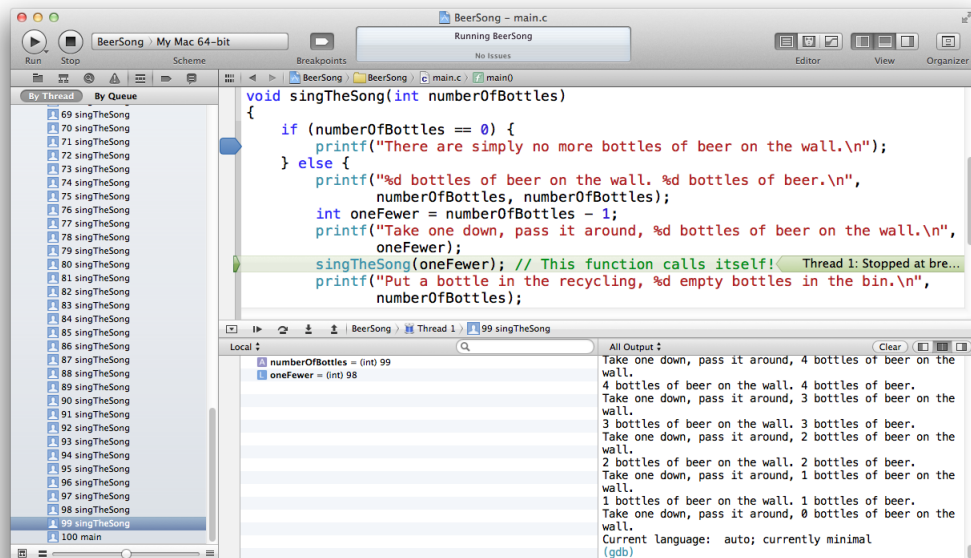
The blue indicator shows that you’ve set a *breakpoint*. A breakpoint is a location in code where you want the debugger to pause the execution of your program. Run the program again. It will start and then stop right before it executes the line where you set the breakpoint.


Now your program is temporarily frozen in time, and you can examine it more closely. In the navigator area, click the  icon to open the *debug navigator*. This navigator shows all the frames currently on the stack, also called a *stack trace*.


In the stack trace, frames are identified by the name of their function. Given your program consists almost entirely of a recursive function, these frames have the same name and you must distinguish them by the value of `oneFewer` that gets passed to them. At the bottom of the stack, you will, of course, find the frame for `main()`.


You can select a frame to see the variables in that frame and the source code for the line of code that is currently being executed. Select the frame for the first time `singTheSong` is called.

Figure 5.7 Frames on the stack for a recursive function



You can see this frame's variables and their values on the bottom left of the window. To the right, you can also see the output in an area called the *console*. (If you don't see the console, find the  buttons at the right of the screen towards the bottom half. Click the middle button to reveal the console.) In the console, you can see the effect of your breakpoint: the program stopped before reaching the line that ends the song.

Now we need to remove the breakpoint so that the program will run normally. You can simply drag the blue indicator off the gutter. Or click the  icon at the top of the navigator area to reveal the *breakpoint navigator* and see all the breakpoints in a project. From there, you can select your breakpoint and delete it.

To resume execution of your program, click the  button on the debugger bar between the editor and the variables view.

We just took a quick look at the debugger here to demonstrate how frames work. However, using the debugger to set breakpoints and browse the frames in a program's stack will be helpful when your program is not doing what you expect and you need to look at what is really happening.

return

Many functions return a value when they complete execution. You know what type of data a function will return by the type that precedes the function name. (If a function doesn't return anything, its return type is `void`.)

Create a new C Command Line Tool named `Degrees`. In `main.c`, add a function before `main()` that converts a temperature from Celsius to Fahrenheit. Then update `main()` to call the new function.

```
#include <stdio.h>

float fahrenheitFromCelsius(float cel)
{
    float fahr = cel * 1.8 + 32.0;
    printf("%f Celsius is %f Fahrenheit\n", cel, fahr);
    return fahr;
}

int main(int argc, const char * argv[])
{
    float freezeInC = 0;
    float freezeInF = fahrenheitFromCelsius(freezeInC);
    printf("Water freezes at %f degrees Fahrenheit\n", freezeInF);
    return 0;
}
```

See how we took the return value of `fahrenheitFromCelsius()` and assigned it to the `freezeInF` variable of type `float`? Pretty slick, huh?

The execution of a function stops when it returns. For example, imagine that you had this function:

```
float average(float a, float b)
{
    return (a + b)/2.0;
    printf("The mean justifies the end\n");
}
```

If you called this function, the `printf()` call would never get executed.

A natural question, then, is “Why do we always return 0 from `main()`?” When you return 0 to the system, you are saying “Everything went OK.” If you are terminating the program because something has gone wrong, you’ll return 1.

This may seem contradictory to how 0 and 1 work in `if` statements; because 1 is true and 0 is false, it’s natural to think of 1 as success and 0 as failure. So think of `main()` as returning an error report. In that case, 0 is good news! Success is a lack of errors.

To make this clearer, some programmers use the constants `EXIT_SUCCESS` and `EXIT_FAILURE`, which are just aliases for 0 and 1 respectively. These constants are defined in the header file `stdlib.h`:

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

float fahrenheitFromCelsius(float cel)
{
    float fahr = cel * 1.8 + 32.0;
    printf("%f Celsius is %f Fahrenheit\n", cel, fahr);
    return fahr;
}

int main(int argc, const char * argv[])
{
    float freezeInC = 0;
    float freezeInF = fahrenheitFromCelsius(freezeInC);
    printf("Water freezes at %f degrees Fahrenheit\n", freezeInF);
    return EXIT_SUCCESS;
}
```

In this book, we will generally use 0 instead of EXIT_SUCCESS.

Global and static variables

In this chapter, we talked about local variables that only exist while a function is running. There are also variables that can be accessed from any function at any time. We call these *global variables*. To make a variable global, you declare it outside of a particular function. For example, you could add a `lastTemperature` variable that holds the temperature that was converted from Celsius. Add a global variable to the program:

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

// Declare a global variable
float lastTemperature;

float fahrenheitFromCelsius(float cel)
{
    lastTemperature = cel;
    float fahr = cel * 1.8 + 32.0;
    printf("%f Celsius is %f Fahrenheit\n", cel, fahr);
    return fahr;
}
int main(int argc, const char * argv[])
{
    float freezeInC = 0;
    float freezeInF = fahrenheitFromCelsius(freezeInC);
    printf("Water freezes at %f degrees Fahrenheit\n", freezeInF);
    printf("The last temperature converted was %f\n", lastTemperature);
    return EXIT_SUCCESS;
}
```

Any complex program will involve dozens of files containing different functions. Global variables are available to the code in every one of those files. Sometimes sharing a variable between different files is what you want. But, as you can imagine, having a variable that can be accessed by multiple functions can also lead to great confusion. To deal with this, we have *static variables*. A static variable is like a global variable in that it is declared outside of any function. However, a static variable is only accessible from the code in the file where it was declared. So you get the non-local, “exists outside of any function” benefit while avoiding the “you touched my variable!” issue.

You can change your global variable to a static variable, but because you have only one file, `main.c`, it will have no effect whatsoever.

```
// Declare a static variable
static float lastTemperature;
```

Both static and global variables can be given an initial value when they are created:

```
// Initialize lastTemperature to 50 degrees
static float lastTemperature = 50.0;
```

If you don’t give them an initial value, they are automatically initialized to zero.

In this chapter, you have learned about functions. When we get to Objective-C in Part III, you will hear the word *method* – a method is very, very similar to a function.

Challenge

The interior angles of a triangle must add up to 180 degrees. Create a new C Command Line Tool named `Triangle`. In `main.c`, write a function that takes the first two angles and returns the third. Here's what it will look like when you call it:

```
#include <stdio.h>

// Add your new function here

int main(int argc, const char * argv[])
{
    float angleA = 30.0;
    float angleB = 60.0;
    float angleC = remainingAngle(angleA, angleB);
    printf("The third angle is %.2f\n", angleC);
    return 0;
}
```

The output should be:

The third angle is 90.00

6

Numbers

We've used numbers to measure and display temperature, weight, and how long to cook a turkey. Now let's take a closer look at how numbers work in C programming. On a computer, numbers come in two flavors: integers and floating-point numbers. You have already used both. This chapter is an attempt to codify what a C programmer needs to know about these numbers.

printf()

But before we get to numbers, let's take a closer look at the **printf()** function you've been using. **printf()** prints a *string* to the log. A string is a string of characters. Basically, it's text.

Reopen your `ClassCertificates` project. In `main.c`, find **congratulateStudent()**.

```
void congratulateStudent(char *student, char *course, int numDays)
{
    printf("%s has done as much %s Programming as I could fit into %d days.\n",
           student, course, numDays);
}
```

What does this call to **printf()** do? Well, you've seen the output; you know what it does. Now let's figure out how.

printf() is a function that accepts a string as an argument. You can make a *literal string* (as opposed to a string that's stored in a variable) by surrounding text in double quotes.

The string that **printf()** takes as an argument is known as the *format string*, and the format string can have *tokens*. The three tokens in this string are `%s`, `%s`, and `%d`. When the program is run, the tokens are replaced with the values of the variables that follow the string. In this case, those variables are `student`, `course`, and `numDays`. Notice that they are replaced in order in the output. If you swapped `student` and `course` in the list of variables, you would see

Cocoa has done as much Mark Programming as I could fit into 5 days.

However, tokens and variables are not completely interchangeable. The `%s` token expects a string. The `%d` expects an integer. (Try swapping them and see what happens.)

Notice that `student` and `course` are declared as type `char *`. For now, just read `char *` as a type that is a string. We'll come back to strings in Objective-C in Chapter 14 and back to `char *` in Chapter 34.

Finally, what's with the `\n`? In **printf()** statements, you have to include an explicit new-line character or all the log output will run together on one line. `\n` represents the new-line character.

Now let's get back to numbers.

Integers

An integer is a number without a decimal point – a whole number. Integers are good for problems like counting. Some problems, like counting every person on the planet, require really large numbers. Other problems, like counting the number of children in a classroom, require numbers that aren't as large.

To address these different problems, integer variables come in different sizes. An integer variable has a certain number of bits in which it can encode a number, and the more bits the variable has, the larger the number it can hold. Typical sizes are: 8-bit, 16-bit, 32-bit, and 64-bit.

Similarly, some problems require negative numbers, while others do not. So, integer types come in signed and unsigned varieties.

An unsigned 8-bit number can hold any integer from 0 to 255. How did I get that? $2^8 = 256$ possible numbers. And we choose to start at 0.

A signed 64-bit number can hold any integer from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807. $2^{63} = 9,223,372,036,854,775,808$ minus one bit for the sign (+ or -).

When you declare an integer, you can be very specific:

```
UInt32 x; // An unsigned 32-bit integer
SInt16 y; // An signed 16-bit integer
```

However, it is more common for programmers just to use the descriptive types that you learned in Chapter 3.

```
char a;      // 8 bits
short b;     // Usually 16 bits (depending on the platform)
int c;       // Usually 32 bits (depending on the platform)
long d;      // 32 or 64 bits (depending on the platform)
long long e; // 64 bits
```

Why is `char` a number? Any character can be described as an 8-bit number, and computers prefer to think in numbers. What about sign? `char`, `short`, `int`, `long`, and `long long` are signed by default, but you can prefix them with `unsigned` to create the unsigned equivalent.

Also, the sizes of integers depend on the platform. (A *platform* is a combination of an operating system and a particular computer or mobile device.) Some platforms are 32-bit and others are 64-bit. The difference is in the size of the memory address, and we'll talk more about that in Chapter 8.

Apple has created two integer types that are 32-bit on 32-bit platforms and 64-bit on 64-bit platforms:

```
NSInteger g;
NSUInteger h;
```

In much of Apple's code, you will see these types used. They are, for all intents and purposes, the same as `long` and `unsigned long`.

Tokens for displaying integers

Create a new project: a C Command Line Tool called `Numbers`. In `main.c`, create an integer and print it out in base-10 (as a decimal number) using `printf()`:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    int x = 255;
    printf("x is %d.\n", x);
    return 0;
}
```

You should see something like

x is 255.

As we've seen, %d prints an integer as a decimal number. What other tokens work? You can print the integer in base-8 (octal) or base-16 (hexadecimal). Add a couple of lines to the program:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    int x = 255;
    printf("x is %d.\n", x);
    printf("In octal, x is %o.\n", x);
    printf("In hexadecimal, x is %x.\n", x);

    return 0;
}
```

When you run it, you should see something like:

x is 255.
In octal, x is 377.
In hexadecimal, x is ff.

(We'll return to hexadecimal numbers in Chapter 33.)

What if the integer has lots of bits? You slip an l (for long) or an ll (for long long) between the % and the format character. Change your program to use a long instead of an int:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    long x = 255;
    printf("x is %ld.\n", x);
    printf("In octal, x is %lo.\n", x);
    printf("In hexadecimal, x is %lx.\n", x);

    return 0;
}
```

If you are printing an unsigned decimal number, you should use %u:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    unsigned long x = 255;
    printf("x is %lu.\n", x);

    // Octal and hex already assumed the number was unsigned
    printf("In octal, x is %lo.\n", x);
    printf("In hexadecimal, x is %lx.\n", x);

    return 0;
}
```

Integer operations

The arithmetic operators `+`, `-`, and `*` work as you would expect. They also have the precedence rules that you would expect: `*` is evaluated before `+` or `-`. In `main.c`, replace the previous code with a calculation:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);

    return 0;
}
```

You should see

```
3 * 3 + 5 * 2 = 19
```

Integer division

Most beginning C programmers are surprised by how integer division works. Try it:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);
    printf("11 / 3 = %d\n", 11 / 3);

    return 0;
}
```

You'll get `11 / 3 = 3.666667`, right? Nope. You get `11 / 3 is 3`. When you divide one integer by another, you always get a third integer. The system rounds off toward zero. (So, `-11 / 3` is `-3`)

This actually makes sense if you think "11 divided by 3 is 3 with a remainder of 2." And it turns out that the remainder is often quite valuable. The modulus operator (`%`) is like `/`, but it returns the remainder instead of the quotient:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);
    printf("11 / 3 = %d remainder of %d \n", 11 / 3, 11 % 3);

    return 0;
}
```

What if you *want* to get 3.666667? You convert the `int` to a `float` using the *cast operator*. The cast operator is the type that you want placed in parentheses to the left of the variable you want converted. Cast your denominator as a `float` before you do the division:

```
int main (int argc, const char * argv[])
{
    printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);
    printf("11 / 3 = %d remainder of %d \n", 11 / 3, 11 % 3);
    printf("11 / 3.0 = %f\n", 11 / (float)3);

    return 0;
}
```

Now, floating point division will be done instead of integer division, and you'll get 3.666667. Here's the rule for integer vs. floating-point division: `/` is integer division only if both the numerator and denominator are integer types. If either is a floating-point number, floating-point division is done instead.

Operator shorthand

All the operators that you've seen so far yield a new result. So, for example, to increase `x` by 1, you would use the `+` operator and then assign the result back into `x`:

```
int x = 5;
x = x + 1; // x is now 6
```

C programmers do these sorts of operations so often that operators were created that change the value of the variable without an assignment. For example, you can increase the value held in `x` by 1 with the *increment operator* (`++`):

```
int x = 5;
x++; // x is now 6
```

There is also a *decrement operator* (`--`) that decreases the value by 1:

```
int x = 5;
x--; // x is now 4
```

What if you want to increase `x` by 5 instead of just 1? You could use addition and assignment:

```
int x = 5;
x = x + 5; // x is 10
```

But there is a shorthand for this, too:

```
int x = 5;
x += 5; // x is 10
```

You can think of the second line as “assign `x` the value of `x + 5`.” In addition to `+=`, there is also `-=`, `*=`, `/=`, and `%=`.

To get the absolute value of an `int`, you use a function instead of an operator. The function is **`abs()`**. If you want the absolute value of a `long`, use **`labs()`**. Both functions are declared in `stdlib.h`:

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

int main (int argc, const char * argv[])
{
    printf("3 * 3 + 5 * 2 = %d\n", 3 * 3 + 5 * 2);
    printf("11 / 3 = %d remainder of %d\n", 11 / 3, 11 % 3);
    printf("11 / 3.0 = %f\n", 11 / (float)3);
    printf("The absolute value of -5 is %d\n", abs(-5));

    return 0;
}
```

Floating-point numbers

If you need a number with a decimal point, like 3.2, you use a floating-point number. Most programmers think of a floating-point number as a mantissa multiplied by 10 to an integer exponent. For example, 345.32 is thought of as 3.4532×10^2 . And this is essentially how they are stored: a 32-bit floating number has 8 bits dedicated to holding the exponent (a signed integer) and 23 bits dedicated to holding the mantissa with the remaining 1 bit used to hold the sign.

Like integers, floating-point numbers come in several sizes. Unlike integers, floating-point numbers are *always* signed:

```
float g;           // 32-bits
double h;          // 64-bits
long double i;     // 128-bits
```

Tokens for displaying floating-point numbers

`printf()` can also display floating point numbers, most commonly using the tokens `%f` and `%e`. In `main.c`, replace the integer-related code:

```
int main (int argc, const char * argv[])
{
    double y = 12345.6789;
    printf("y is %f\n", y);
    printf("y is %e\n", y);

    return 0;
}
```

When you build and run it, you should see:

```
y is 12345.678900
y is 1.234568e+04
```

So `%f` uses normal decimal notation, and `%e` uses scientific notation.

Notice that `%f` is currently showing 6 digits after the decimal point. This is often a bit much. Limit it to two digits by modifying the token:

```
int main (int argc, const char * argv[])
{
    double y = 12345.6789;
    printf("y is %.2f\n", y);
    printf("y is %.2e\n", y);
    return 0;
}
```

When you run it, you should see:

```
y is 12345.68
y is 1.23e+04
```

Functions for floating-point numbers

The operators `+`, `-`, `*`, and `/` do exactly what you would expect. If you will be doing a lot of math, you will need the math library. To see what's in the math library, open the Terminal application on your Mac and type `man math`. You will get a great summary of everything in the math library: trigonometry, rounding, exponentiation, square and cube root, etc.

If you use any of these math functions in your code, be sure to include the math library header at the top that file:

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

One warning: all of the trig-related functions are done in radians, not degrees!

Challenge

Use the math library! Add code to `main.c` that displays the sine of 1 radian. Show the number rounded to three decimal points.

7

Loops

In Xcode, create yet another new project: a C Command Line Tool named Coolness.

The first program I ever wrote printed the words, “Aaron is Cool”. (I was 10 at the time.) Write that program now:

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    printf("Aaron is Cool\n");
    return 0;
}
```

Build and run the program.

Let’s suppose for a moment that you could make my 10-year-old self feel more confident if the program printed the affirmation a dozen times. How would you do that?

Here’s the dumb way:

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    printf("Aaron is Cool\n");
    return 0;
}
```

The smart way is to create a loop.

The while loop

The first loop we’ll use is a while loop. The while construct works something like the if construct we discussed in Chapter 4. You give it an expression and a block of code contained by curly braces. In the

if construct, if the expression is true, the block of code is run once. In the while construct, the block is run again and again until the expression becomes false.

Rewrite the `main()` function to look like this:

```
#include <stdio.h>

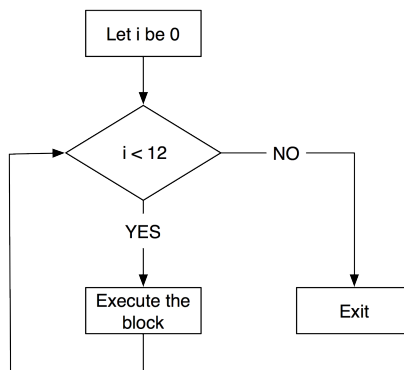
int main(int argc, const char * argv[])
{
    int i = 0;
    while (i < 12) {
        printf("%d. Aaron is Cool\n", i);
        i++;
    }
    return 0;
}
```

Build and run the program.

The conditional (`i < 12`) is being checked before each execution of the block. The first time it evaluates to false, execution leaps to the code after the block.

Notice that the second line of the block increments `i`. This is important. If `i` wasn't incremented, then this loop, as written, would continue forever because the expression would always be true. Here's a flow-chart of this while loop:

Figure 7.1 while loop



The for loop

The while loop is a general looping structure, but C programmers use the same basic pattern a lot:

```
some initialization
while (some check) {
    some code
    some last step
}
```

So, the C language has a shortcut: the for loop. In the for loop, the pattern shown above becomes:

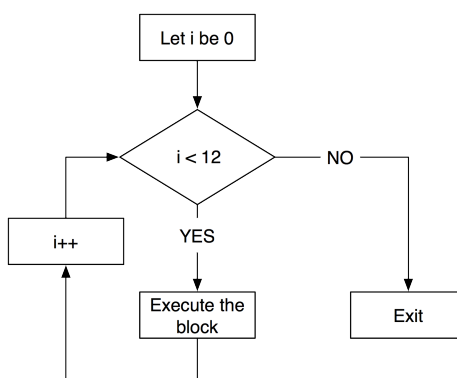
```
for (some initialization; some check; some last step) {
    some code;
}
```

Change the program to use a for loop:

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    for (int i = 0; i < 12; i++) {
        printf("%d. Aaron is Cool\n", i);
    }
    return 0;
}
```

Figure 7.2 for loop



Note that in this simple loop example, you used the loop to dictate the number of times something happens. More commonly, however, loops are used to *iterate* through a collection of items, such as a list of names. For instance, I could modify this program to use a loop in conjunction with a list of friends' names. Each time through the loop, a different friend would get to be cool. We'll see more of collections and loops starting in Chapter 15.

break

Sometimes it is necessary to stop the loop's execution from the inside the loop. For example, let's say you want to step through the positive integers looking for the number x , where $x + 90 = x^2$. Your plan is to step through the integers 0 through 11 and pop out of the loop when you find the solution. Change the code:

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int i;
    for (i = 0; i < 12; i++) {
        printf("Checking i = %d\n", i);
    }
}
```

```

    if (i + 90 == i * i) {
        break;
    }
}
printf("The answer is %d.\n", i);
return 0;
}

```

Build and run the program. You should see

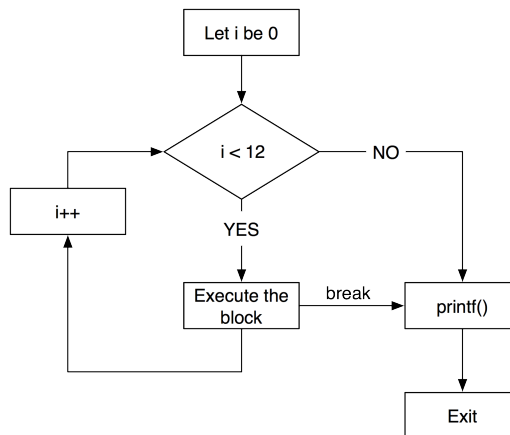
```

Checking i = 0
Checking i = 1
Checking i = 2
Checking i = 3
Checking i = 4
Checking i = 5
Checking i = 6
Checking i = 7
Checking i = 8
Checking i = 9
Checking i = 10
The answer is 10.

```

Notice that when `break` is called execution skips directly to the end of the code block.

Figure 7.3 Breaking out of a loop



continue

Sometimes you will find yourself in the middle of the code block, and you need to say, “Forget the rest of this run through the code block and start the next run through the code block.” This is done with the `continue` command. For example, what if you were pretty sure that no multiples of 3 satisfied the equation? How would you avoid wasting precious time checking those?

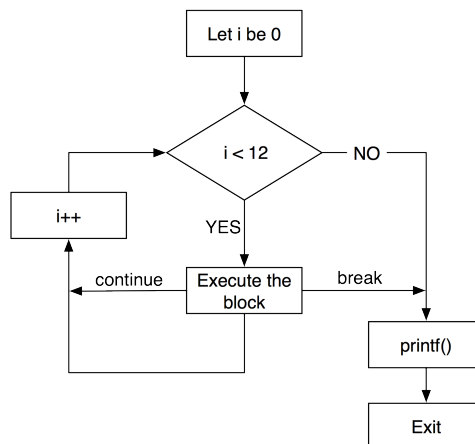
```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int i;
    for (i = 0; i < 12; i++) {
        if (i % 3 == 0) {
            continue;
        }
        printf("Checking i = %d\n", i);
        if (i + 90 == i * i) {
            break;
        }
    }
    printf("The answer is %d.\n", i);
    return 0;
}
```

Build and run it:

```
Checking i = 1
Checking i = 2
Checking i = 4
Checking i = 5
Checking i = 7
Checking i = 8
Checking i = 10
The answer is 10.
```

Figure 7.4 continue



The do-while loop

None of the cool kids use the do-while loop, but for completeness, here it is. The do-while loop doesn't check the expression until it has executed the block. Thus, it ensures that the block is always executed at least once. If you rewrote the original exercise to use a do-while loop, it would look like this:

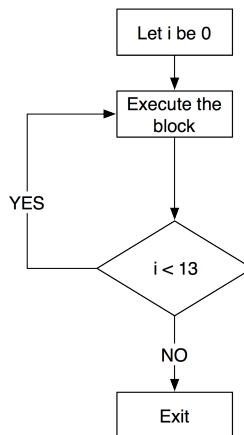
```
int main(int argc, const char * argv[])
{
    int i = 0;
    do {
        printf("%d. Aaron is Cool\n", i);
        i++;
    } while (i < 13);
    return 0;
}
```

Notice the trailing semicolon. That's because unlike the other loops, a do-while loop is actually one long statement:

```
do { something } while ( something else stays true );
```

Here's a flow-chart of this do-while loop:

Figure 7.5 do-while loop



Challenge

Write a program that counts backward from 99 through 0 by 3, printing each number. However, if the number is divisible by 5, it should also print the words “Found one!”. Thus, the output should look something like this:

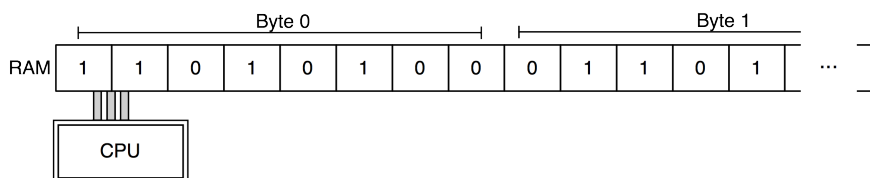
```
99
96
93
90
Found one!
87
...
0
Found one!
```

Addresses and Pointers

Your computer is, at its core, a processor (the Central Processing Unit or CPU) and a vast meadow of switches (the Random-Access memory or RAM) that can be turned on or off by the processor. We say that a switch holds one *bit* of information. You'll often see 1 used to represent “on” and 0 used to represent “off.”

Eight of these switches make a *byte* of information. The processor can fetch the state of these switches, do operations on the bits, and store the result in another set of switches. For example, the processor might fetch a byte from here and another byte from there, add them together, and store the result in a byte way over someplace else.

Figure 8.1 Memory and the CPU



The memory is numbered, and we typically talk about the *address* of a particular byte of data. When people talk about a 32-bit CPU or a 64-bit CPU, they are usually talking about how big the address is. A 64-bit CPU can deal with much, much more memory than a 32-bit CPU.

Getting addresses

In Xcode, create a new project: a C Command Line Tool named Addresses.

The address of a variable is the location in memory where the value for that variable is stored. To get the variable's address, you use the `&` operator:

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    int i = 17;
    printf("i stores its value at %p\n", &i);
}
```

```
    return 0;
}
```

Notice the `%p` token. That's the token you can replace with a memory address. Build and run the program. You'll see something like:

```
i stores its value at 0xbffff738
```

although your computer may put `i` at a completely different address. Memory addresses are nearly always printed in hexadecimal format.

In a computer, everything is stored in memory, and thus everything has an address. For example, a function starts at some particular address. To get that address, you just use the function's name:

```
int main(int argc, const char * argv[])
{
    int i = 17;
    printf("i stores its value at %p\n", &i);
    printf("this function starts at %p\n", main);
    return 0;
}
```

Build and run the program.

Storing addresses in pointers

What if you wanted to store an address in a variable? You could stuff it into an unsigned integer that was the right size, but the compiler will help you catch your mistakes if you are more specific when you give that variable its type. For example, if you wanted a variable named `ptr` that holds the address where a float can be found, you would declare it like this:

```
float *ptr;
```

We say that `ptr` is a variable that is a *pointer* to a float. It doesn't store the value of a float; it points to an address where a float may be stored.

Declare a new variable named `addressOfI` that is a pointer to an `int`. Assign it the address of `i`.

```
int main(int argc, const char * argv[])
{
    int i = 17;
    int *addressOfI = &i;
    printf("i stores its value at %p\n", addressOfI);
    printf("this function starts at %p\n", main);
    return 0;
}
```

Build and run the program. You should see no change in its behavior.

We're using integers right now to be simple. But if you're wondering what the point of pointers is, I hear you. It would be just as easy to pass the integer value assigned to this variable as it is to pass its address. Soon, however, your data will be much larger and much more complex than single integers. That's why we pass addresses. It's not always possible to pass a copy of data you want to work with, but you can always pass the *address* of where that data begins. And it's easy to access data once you have its address.

Getting the data at an address

If you have an address, you can get the data stored there using the `*` operator. Have the log display the value of the integer stored at `addressOfI`.

```
int main(int argc, const char * argv[])
{
    int i = 17;
    int *addressOfI = &i;
    printf("i stores its value at %p\n", addressOfI);
    printf("this function starts at %p\n", main);
    printf("the int stored at addressOfI is %d\n", *addressOfI);
    return 0;
}
```

Notice that the asterisk is used two different ways. The first is in the declaration where you declare the variable `addressOfI` to be an `int *`. That is, it is a pointer to a place where an `int` can be stored.

The second is where you read the `int` value that is stored at the address stored in `addressOfI`. (Pointers are also called references. Thus, using the pointer to read data at the address is sometimes called *dereferencing* the pointer.)

You can also use the `*` operator on the left-hand side of an assignment to store data at a particular address:

```
int main(int argc, const char * argv[])
{
    int i = 17;
    int *addressOfI = &i;
    printf("i stores its value at %p\n", addressOfI);
    *addressOfI = 89;
    printf("Now i is %d\n", i);
    return 0;
}
```

Build and run your program.

Don't worry if you don't have pointers squared away in your mind just yet. We'll spend a lot of time working with pointers in this book, so you'll get plenty of practice.

Now let's make a common programming mistake. Remove the `*` from the fourth line of `main()` so that it reads

```
addressOfI = 89;
```

Notice Xcode pops up a warning that says *Incompatible integer to pointer conversion assigning to 'int **' to 'int'*. Fix the problem.

How many bytes?

Given that everything lives in memory and that you now know how to find the address where data starts, the next question is "How many bytes does this data type consume?"

Using `sizeof()` you can find the size of a data type. For example,

```
int main(int argc, const char * argv[])
{
    int i = 17;
    int *addressOfI = &i;
    printf("i stores its value at %p\n", addressOfI);
    *addressOfI = 89;
    printf("Now i is %d\n", i);
    printf("An int is %zu bytes\n", sizeof(int));
    printf("A pointer is %zu bytes\n", sizeof(int *));
    return 0;
}
```

We see yet another new token in the calls to **printf()**: **%zu**. The **sizeof()** function returns a value of type **size_t**, for which **%zu** is the correct placeholder token. This one's not very common in the wild.

Build and run the program. If your pointer is 4 bytes long, your program is running in 32-bit mode. If your pointer is 8 bytes long, your program is running in 64-bit mode.

sizeof() will also take a variable as an argument, so you could have written the previous program like this:

```
int main(int argc, const char * argv[])
{
    int i = 17;
    int *addressOfI = &i;
    printf("i stores its value at %p\n", addressOfI);
    *addressOfI = 89;
    printf("Now i is %d\n", i);
    printf("An int is %zu bytes\n", sizeof(i));
    printf("A pointer is %zu bytes\n", sizeof(addressOfI));
    return 0;
}
```

NULL

Sometimes you need a pointer to nothing. That is, you have a variable that can hold an address, and you want to store something in it that makes it explicit that the variable is not set to anything. We use **NULL** for this:

```
float *myPointer;
// Set myPointer to NULL for now, I'll store a pointer there
// later in the program
myPointer = NULL;
```

What is **NULL**? Remember that an address is just a number. **NULL** is zero. This is very handy in **if** statements:

```
float *myPointer;
...
// Has myPointer been set?
if (myPointer) {
    // myPointer is not NULL
    ...do something with the data at myPointer...
} else {
    // myPointer is NULL
}
```

Later, when we discuss pointers to objects, we will use **nil** instead of **NULL**. They are equivalent, but Objective-C programmers use **nil** to mean the address where no object lives.

Stylish pointer declarations

When you declare a pointer to `float`, it looks like this:

```
float *powerPtr;
```

Because the type is a pointer to a `float`, you may be tempted to write it like this:

```
float* powerPtr;
```

This is fine, and the compiler will let you do it. However, stylish programmers don't.

Why? You can declare multiple variables in a single line. For example, if I wanted to declare variables `x`, `y`, and `z`, I could do it like this:

```
float x, y, z;
```

Each one is a `float`.

What do you think these are?

```
float* b, c;
```

Surprise! `b` is a pointer to a `float`, but `c` is just a `float`. If you want them both to be pointers, you must put a `*` in front of each one:

```
float *b, *c;
```

Putting the `*` directly next to the variable name makes this clearer.

Challenges

Write a program that shows you how much memory a `float` consumes.

On your Mac, a `short` is a 2-byte integer, and one bit is used to hold the sign (positive or negative). What is the smallest number it can store? What is the largest? An unsigned `short` only holds non-negative numbers. What is the largest number it can store?

9

Pass By Reference

There is a standard C function called `modf()`. You give `modf()` a double, and it calculates the integer part and the fraction part of the number. For example, if you give it 3.14, 3 is the integer part and 0.14 is the fractional part.

You, as the caller of `modf()` want both parts. However, a C function can only return one value. How can `modf()` give you both pieces of information?

When you call `modf()`, you will supply an address where it can stash one of the numbers. In particular, it will return the fractional part and copy the integer part to the address you supply. Create a new project: a C Command Line Tool named PBR.

Edit `main.c`:

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

int main(int argc, const char * argv[])
{
    double pi = 3.14;
    double integerPart;
    double fractionPart;

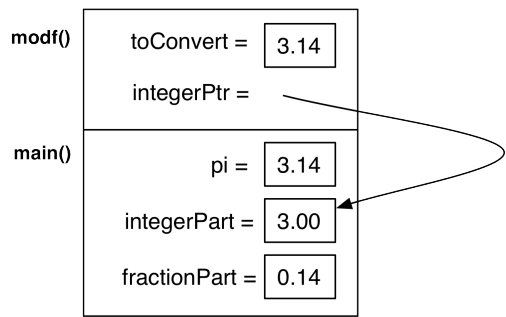
    // Pass the address of integerPart as an argument
    fractionPart = modf(pi, &integerPart);

    // Find the value stored in integerPart
    printf("integerPart = %.0f, fractionPart = %.2f\n", integerPart, fractionPart);

    return 0;
}
```

This is known as *pass-by-reference*. That is, you supply an address (also known as “a reference”), and the function puts the data there.

Figure 9.1 The stack as `modf()` returns



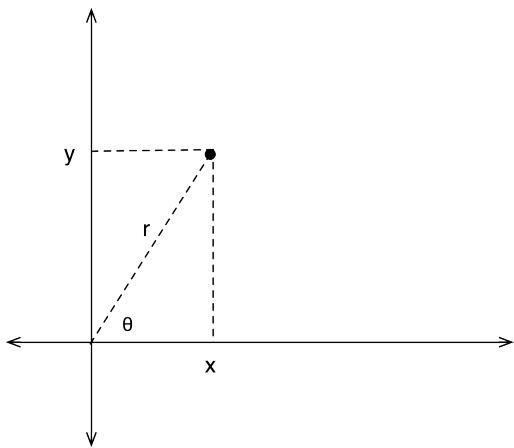
Here’s another way to think about pass-by-reference. Imagine that you give out assignments to spies. You might tell one, “I need photos of the finance minister with his girlfriend. I’ve left a short length of steel pipe at the foot of the angel statue in the park. When you get the photos, roll them up and leave them in the pipe. I’ll pick them up Tuesday after lunch.” In the spy biz, this is known as a *dead drop*.

`modf()` works just like a dead drop. You are asking it to execute and telling it a location where the result can be placed so you can find it later. The only difference is that instead of a steel pipe, you are giving it a location in memory where the result can be placed.

Writing pass-by-reference functions

There are two popular ways to describe the location of a point in 2-dimensional space: Cartesian coordinates and polar coordinates. In Cartesian coordinates, (x, y) indicates that you should go to the right x and then up y . In polar coordinates, (θ, r) indicates that you should turn to the left by θ radians and go forward r .

Figure 9.2 Polar and Cartesian coordinates



What if you wanted to write a function that converted a point in Cartesian coordinates to polar coordinates? It would need to read two floating point numbers and return two floating point numbers. The declaration of the function would look like this:

```
void cartesianToPolar(float x, float y, float *rPtr, float *thetaPtr)
```

That is, when the function is called, it will be passed values for *x* and *y*. It will also be supplied with locations where the values for radius and theta can be stored.

Now write the function near the top of your `main.c` file and call it from `main()`:

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

void cartesianToPolar(float x, float y, double *rPtr, double *thetaPtr)
{
    // Store the radius in the supplied address
    *rPtr = sqrt(x * x + y * y);

    // Calculate theta
    float theta;
    if (x == 0.0) {
        if (y == 0.0) {
            theta = 0.0;    // technically considered undefined
        } else if (y > 0) {
            theta = M_PI_2;
        } else {
            theta = - M_PI_2;
        }
    } else {
        theta = atan(y/x);
    }
    // Store theta in the supplied address
    *thetaPtr = theta;
}

int main(int argc, const char * argv[])
{
    double pi = 3.14;
    double integerPart;
    double fractionPart;

    // Pass the address of integerPart as an argument
    fractionPart = modf(pi, &integerPart);

    // Find the value stored in integerPart
    printf("integerPart = %.0f, fractionPart = %.2f\n", integerPart, fractionPart);

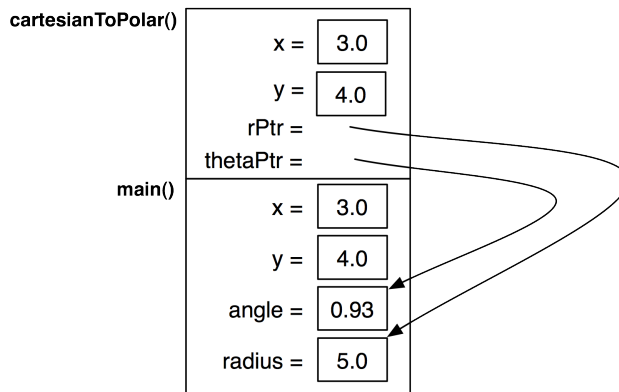
    double x = 3.0;
    double y = 4.0;
    double radius;
    double angle;

    cartesianToPolar(x, y, &angle, &radius);
    printf("(%.2f, %.2f) becomes (%.2f radians, %.2f)\n", x, y, radius, angle);

    return 0;
}
```

Build and run the program.

Figure 9.3 The stack as cartesianToPolar() returns



Avoid dereferencing NULL

Sometimes a function can supply many values by reference, but you may only care about some of them. How do you avoid declaring these variables and passing their addresses when you're not going to use them anyway? Typically, you pass `NULL` as an address to tell the function "I don't need this particular value."

This means that you should always check to make sure the pointers are non-`NULL` before you dereference them. Add these checks in `cartesianToPolar()`:

```
void cartesianToPolar(float x, float y, double *rPtr, double *thetaPtr)
{
    // Is rPtr non-NULL?
    if (rPtr) {
        // Store the radius in the supplied address
        *rPtr = sqrt(x * x + y * y);
    }

    // Is thetaPtr NULL?
    if (!thetaPtr) {
        // Skip the rest of the function
        return;
    }

    // Calculate theta
    float theta;
    if (x == 0.0) {
        if (y == 0.0) {
            theta = 0.0;    // technically considered undefined
        } else if (y > 0) {
            theta = M_PI_2;
        } else {
            theta = - M_PI_2;
        }
    } else {
        ...
    }
}
```


10

Structs

Sometimes you need a variable to hold several related chunks of data. For example, imagine that you were writing a program that computed a person's Body Mass Index. (What is your BMI? It is your weight in kilograms divided by the square of your height in meters. A BMI under 20 suggests that you may be underweight. A BMI over 30 suggests that you may be obese. It is a very imprecise tool for measuring a person's fitness, but it makes a fine programming example.) A person, for your purposes, consists of a float that represents height in meters and an int that represents weight in kilograms.

Now you're going to create your own Person type. A variable of type Person will be a structure and will have two members: a float called heightInMeters and an int called weightInKilos.

Create a new project: a C Command Line Tool called BMICalc. Edit main.c to create a structure that contains the data you need for a person:

```
#include <stdio.h>

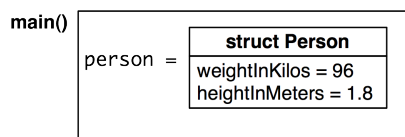
// Here is the declaration of the struct Person
struct Person {
    float heightInMeters;
    int weightInKilos;
};

int main(int argc, const char * argv[])
{
    struct Person person;
    person.weightInKilos = 96;
    person.heightInMeters = 1.8;
    printf("person weighs %i kilograms\n", person.weightInKilos);
    printf("person is %.2f meters tall\n", person.heightInMeters);
    return 0;
}
```

Notice that you access the members of a struct using a period.

Here's the frame for `main()` after the struct's members have been assigned values.

Figure 10.1 Frame after member assignments



Most of the time, you use a structure declaration over and over again. So it's common to create a typedef for the structure type. A typedef defines an alias for a type declaration and allows us to use it more like the usual data types. Change `main.c` to create and use a typedef for struct `Person`:

```
#include <stdio.h>

// Here is the declaration of the type Person
typedef struct {
    float heightInMeters;
    int weightInKilos;
} Person;

int main(int argc, const char * argv[])
{
    Person person;
    person.weightInKilos = 96;
    person.heightInMeters = 1.8;
    printf("person weighs %i kilograms\n", person.weightInKilos);
    printf("person is %.2f meters tall\n", person.heightInMeters);
    return 0;
}
```

Now that you've created a typedef, you can pass a `Person` structure to another function. Add a function named `bodyMassIndex()` that accepts a `Person` as a parameter and calculates BMI. Then update `main()` to call it:

```
#include <stdio.h>

typedef struct _Person {
    float heightInMeters;
    int weightInKilos;
} Person;

float bodyMassIndex(Person p)
{
    return p.weightInKilos / (p.heightInMeters * p.heightInMeters);
}

int main(int argc, const char * argv[])
{
    Person person;
    person.weightInKilos = 96;
    person.heightInMeters = 1.8;
    float bmi = bodyMassIndex(person);
    printf("person has a BMI of %.2f\n", bmi);
    return 0;
}
```

Challenge

The first structure I had to deal with as a programmer was struct `tm`, which the standard C library uses to hold time broken down into its components. The struct is defined:

```
struct tm {
    int    tm_sec;    /* seconds after the minute [0-60] */
    int    tm_min;    /* minutes after the hour [0-59] */
    int    tm_hour;   /* hours since midnight [0-23] */
    int    tm_mday;   /* day of the month [1-31] */
    int    tm_mon;    /* months since January [0-11] */
    int    tm_year;    /* years since 1900
```

```

int    tm_mon;      /* months since January [0-11] */
int    tm_year;     /* years since 1900 */
int    tm_wday;     /* days since Sunday [0-6] */
int    tm_yday;     /* days since January 1 [0-365] */
int    tm_isdst;    /* Daylight Savings Time flag */
long   tm_gmtoff;   /* offset from CUT in seconds */
char   *tm_zone;    /* timezone abbreviation */
};

```

The function **time()** returns the number of seconds since the first moment of 1970 in Greenwich, England. **localtime_r()** can read that duration and pack a struct `tm` with the appropriate values. (It actually takes the *address* of the number of seconds since 1970 and the *address* of an struct `tm`.) Thus, getting the current time as a struct `tm` looks like this:

```

long secondsSince1970 = time(NULL);
printf("It has been %ld seconds since 1970\n", secondsSince1970);

struct tm now;
localtime_r(&secondsSince1970, &now);
printf("The time is %d:%d:%d\n", now.tm_hour, now.tm_min, now.tm_sec);

```

The challenge is to write a program that will tell you what the date (4-30-2015 format is fine) will be in 4 million seconds.

(One hint: `tm_mon = 0` means January, so be sure to add 1. Also, include the `<time.h>` header at the start of your program.)

11

The Heap

So far, your programs have only used memory that has been in frames on the stack. This memory is automatically allocated when the function starts and automatically destroyed when the function ends. (In fact, local variables are often called *automatic variables* because of this convenient behavior.)

Sometimes, however, you need to explicitly claim a long line of bytes of memory and use it in many functions. For example, you might read a file of text into memory and then call a function that would count all the vowels in that memory. Typically, once you were finished with the text, you would let the program know you were all done so the program could reuse that memory for something else.

Programmers often use the word *buffer* to mean a long line of bytes of memory. (This explains the term “buffering” to describe the wait for YouTube to send you enough bytes for that cat video to get started.)

You claim a buffer of memory using the function `malloc()`. The buffer comes from a region of memory known as the *heap*, which is separate from the stack. When you’re done using the buffer, you call the function `free()` to release your claim on that memory and return it to the heap. Let’s say, for example, I needed a chunk of memory big enough to hold 1,000 floats.

```
#include <stdio.h>
#include <stdlib.h> // malloc and free are in stdlib

int main(int argc, const char * argv[])
{
    // Declare a pointer
    float *startOfBuffer;

    // Ask to use some bytes from the heap
    startOfBuffer = malloc(1000 * sizeof(float));

    // ...use the buffer here...

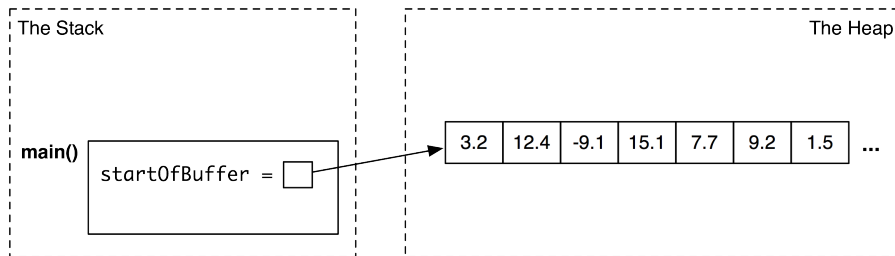
    // Relinquish your claim on the memory so it can be reused
    free(startOfBuffer);

    // Forget where that memory is
    startOfBuffer = NULL;

    return 0;
}
```

`startOfBuffer` would be a pointer to the first floating point number in the buffer.

Figure 11.1 A pointer on the stack to a buffer on the heap



At this point, most C books would spend a lot of time talking about how to read and write data in assorted locations in that buffer of floating point numbers. This book, however, is trying to get you to objects as quickly as possible. So, we will put off C arrays and pointer arithmetic until later.

You can also use `malloc()` to claim space for a struct on the heap. For example, if you wanted to allocate a `Person` struct on the heap, you might have a program like this:

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

typedef struct {
    float heightInMeters;
    int weightInKilos;
} Person;

float bodyMassIndex(Person *p)
{
    return p->weightInKilos / (p->heightInMeters * p->heightInMeters);
}

int main(int argc, const char * argv[])
{
    // Allocate memory for one Person structure
    Person *x = (Person *)malloc(sizeof(Person));

    // Fill in two members of the structure
    x->weightInKilos = 81;
    x->heightInMeters = 2.0;

    // Print out the BMI of the original Person
    float xBMI = bodyMassIndex(x);
    printf("x has a BMI of = %f\n", xBMI);

    // Let the memory be recycled
    free(x);

    // Forget where it was
    x = NULL;

    return 0;
}
```

Notice the operator `->`. `p->weightInKilos` says, “Dereference the pointer `p` to the structure and get me the member called `weightInKilos`.”

This idea of structures on the heap is a very powerful one. It forms the basis for Objective-C objects, which we turn to next.