Objective-C Programming

pre-course workbook

AARON HILLEGASS & MIKEY WARD

BiG
nerD
ranch

Objective-C Programming: Pre-Course Workbook

by Aaron Hillegass and Mikey Ward

Big Nerd Ranch, LLC.
1989 College Ave NE
Atlanta, GA 30317
(404) 478-9005
http://www.bignerdranch.com/
book-comments@bignerdranch.com

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

ISBN-10 032194206X
ISBN-13 978-0321942067

Second edition, November 2013 (but, uh, updated on 2018-01-19)

Table of Contents

Part I

Getting Started

1
Writing C with Xcode

To get started writing in Objective-C, you will need to:

- install Apple's Developer Tools
- create a simple project using those tools
- explore how these tools are used to make sure your 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 OS X (the Mac) or iOS (the iPhone and friends), you will be using Apple's developer tools. The main application that you will need is called Xcode.

Xcode is only available on the Mac (not Windows or Linux), so you will need a Mac to work with this book. In addition, this book is based on Xcode 9, which is compatible with macOS Sierra (10.12.6) and later.

You can download the latest version of Xcode for free from the Mac App Store. You may want to drag the Xcode icon onto your Dock from your `Applications` folder; you will be using it an awful lot.

Getting started with Xcode

Xcode is Apple's *Integrated Development Environment*. 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 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 are not. 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.

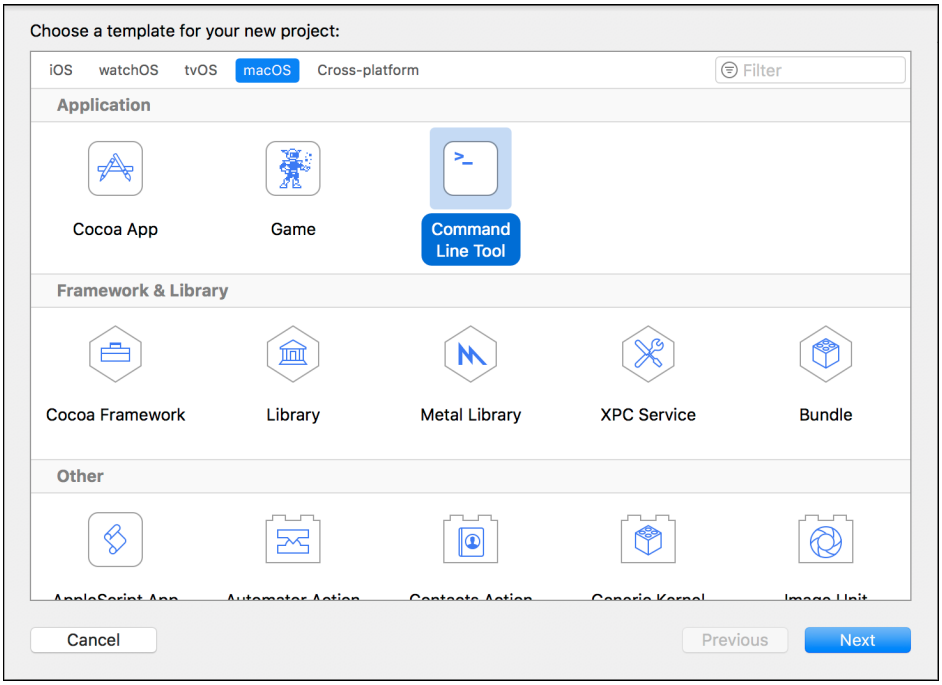
Some programs have no graphical user interface and run for a short time in the terminal; we call these *command-line tools*. In this book, you will be writing mostly command-line tools to focus on programming essentials without the distraction of creating and managing a user interface.

Now you are 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 Project….

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

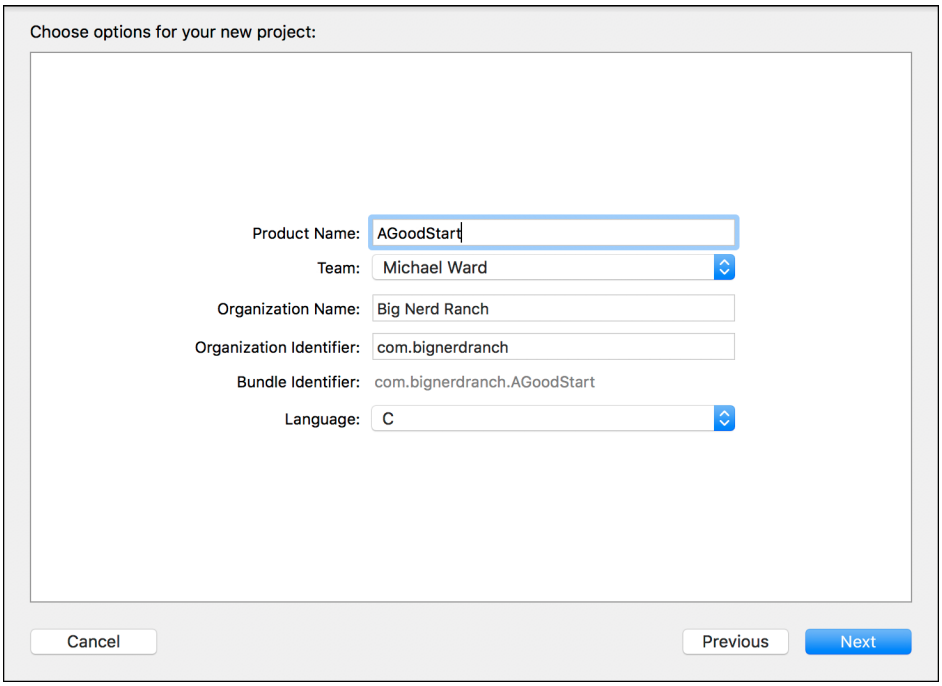
Figure 1.1 Choosing a template



Click the Next button.

Name your new project AGoodStart. The organization name and company identifier will not matter for the exercises in this book, but they are required to continue. Use Big Nerd Ranch and com.bignerdranch. From the Type pop-up menu, select C.

Figure 1.2 Choosing project options



Click the Next button.

In the next window, choose the folder in which you want your project directory to be created. (If you are unsure, accept the default location that Xcode suggests.) You will not need a repository for version control, so uncheck the box labeled Create git repository. Finally, click the Create button.

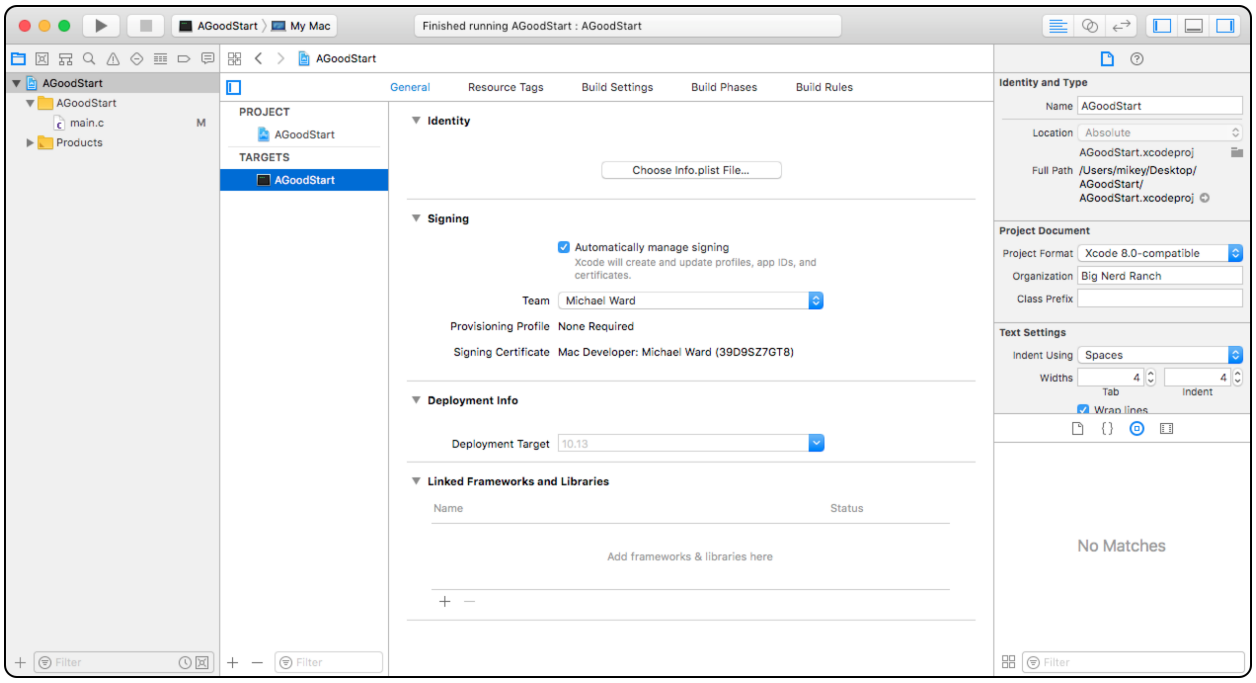
You will be creating this same type of project for the next several chapters. In the future, we will just say, “Create a new C Command Line Tool named *program-name-here*” to get you to follow this same sequence.

Why are you creating C projects? Objective-C is built on top of the C programming language. You will need to have an understanding of parts of C before you can get to the particulars of Objective-C.

Where do I start writing code?

After creating your project, you will be greeted by a window displaying lots of information about AGoodStart.

Figure 1.3 First view of the AGoodStart project



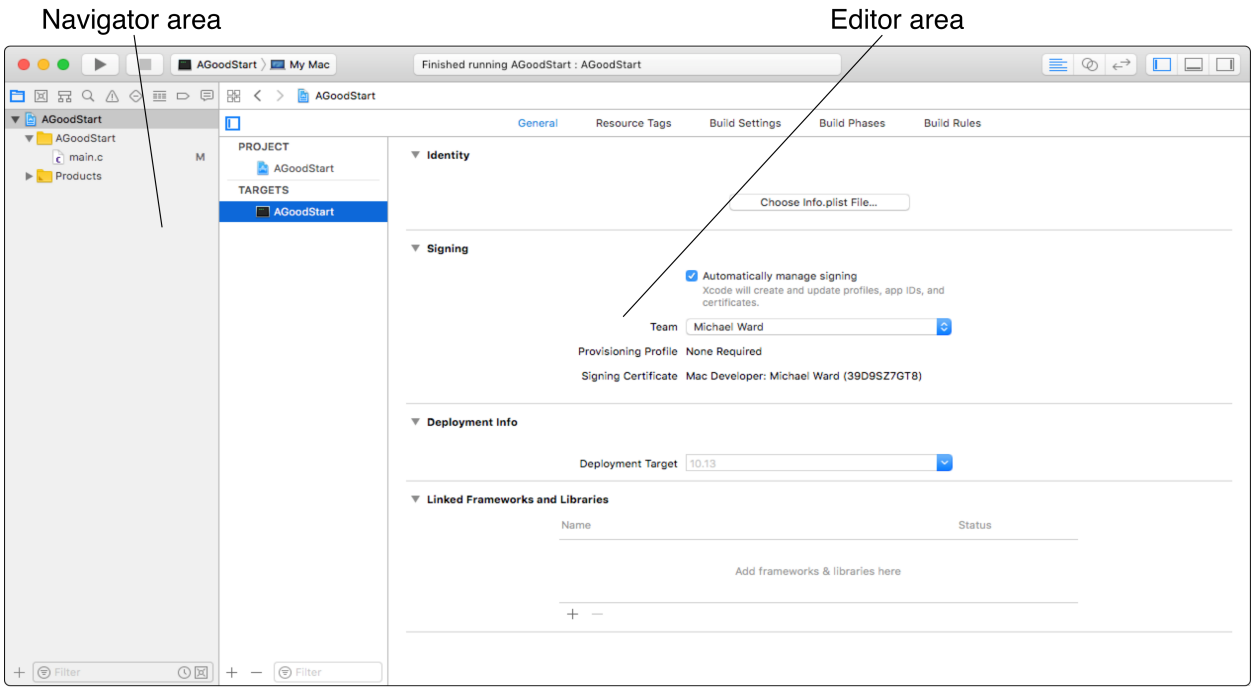
This window is more detailed than you need, so let’s make it a little simpler.

First, at the top right corner of the window, find three buttons that look like this:  .

These buttons hide and show different areas of the window. You will not need the righthand area until later, so click the righthand button to hide it.

You now have two areas at your disposal: the *navigator area* on the left and the *editor area* on the right.

Figure 1.4 Navigator and editor areas in Xcode

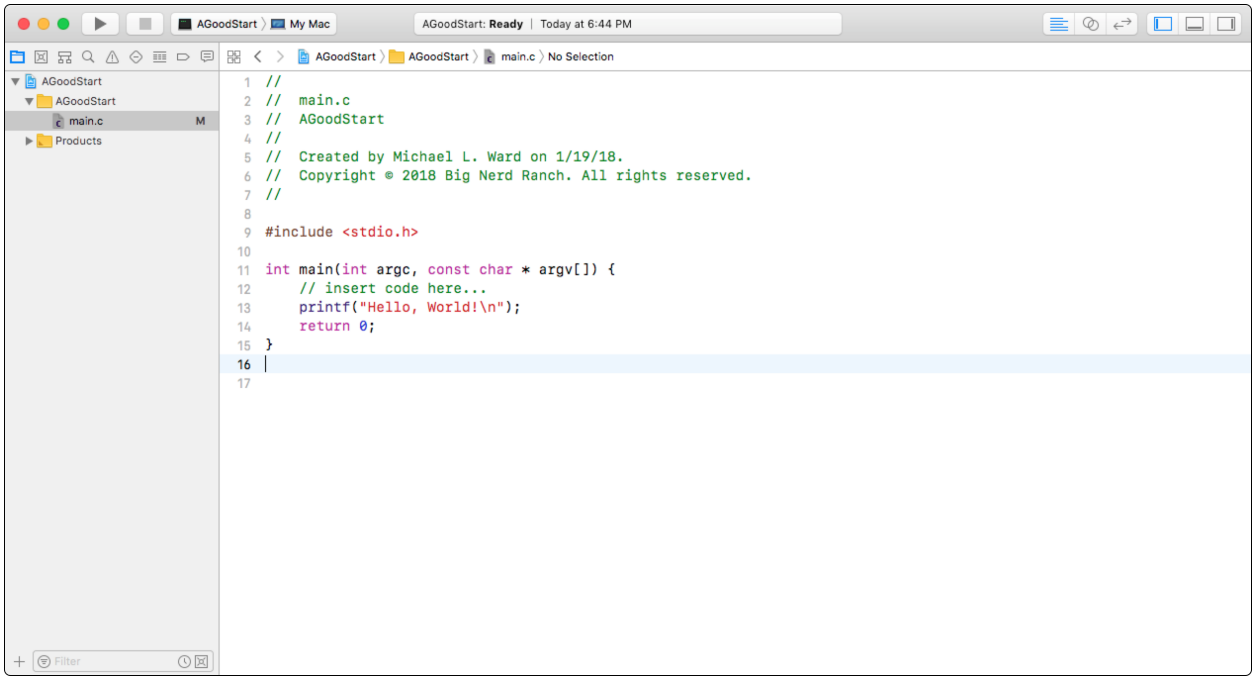


The navigator area displays the current navigator. There are several navigators, and each one provides a different way to examine the contents of your project. You are looking at the *project navigator*. This navigator lists the files that make up your project.

In the project navigator, find a file named `main.c` and click on it. (If you do not see `main.c`, click the triangle next to the folder labeled AGoodStart to reveal its contents.)

When you select `main.c` in the project navigator, the editor area changes to display the contents of this file (Figure 1.5).

Figure 1.5 Selecting `main.c` in the project navigator



The `main.c` file contains a *function* named **`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 name of 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;
}
```

This function contains the two kinds of information that 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 have written. The more difficult the programming problem you are trying to solve, the more comments will help document how you solved the problem. The importance of this documentation becomes apparent when you return to your work months later, look at code you forgot to comment, and think, “I am 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 `//` in 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, 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 will not 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 is one feature that remains consistent: every *statement* ends in a semicolon.

(You will see examples of code statements in just a moment.) If you forget a semicolon, you will have made a syntax error, and your program will not 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 will 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, you need to make some space. Find the curly braces ({ and }) that mark the beginning and end of the **`main`** function. Then delete everything between them.

Now replace the contents of the **`main`** function with what the contents shown below. You will add a comment, two code statements, and another comment. Do not worry if you do not 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 that you need to type in is shown in a bold font. The code that is not bold is code that is already there and will show you where to add the new code. This is a convention that we will 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 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. Select Xcode → Preferences and then open the Text Editing preferences.)

In addition, Xcode uses different font colors to make it easy to identify comments and different parts of your code. For example, comments are always green. After a while of working with Xcode, you will begin to instinctively notice when the colors do not look right. Often, this is a clue that you have made a syntax error. And the sooner you know that you have made an error, the easier it is to find and fix it.

How do I run my program?

It is 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.

In the upper lefthand corner of the project window, find the button that looks suspiciously like the play button in iTunes or on a Blu-ray player. If you leave your cursor over that button, you will see a tool tip that says Build and then run the current scheme. Click this button.

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

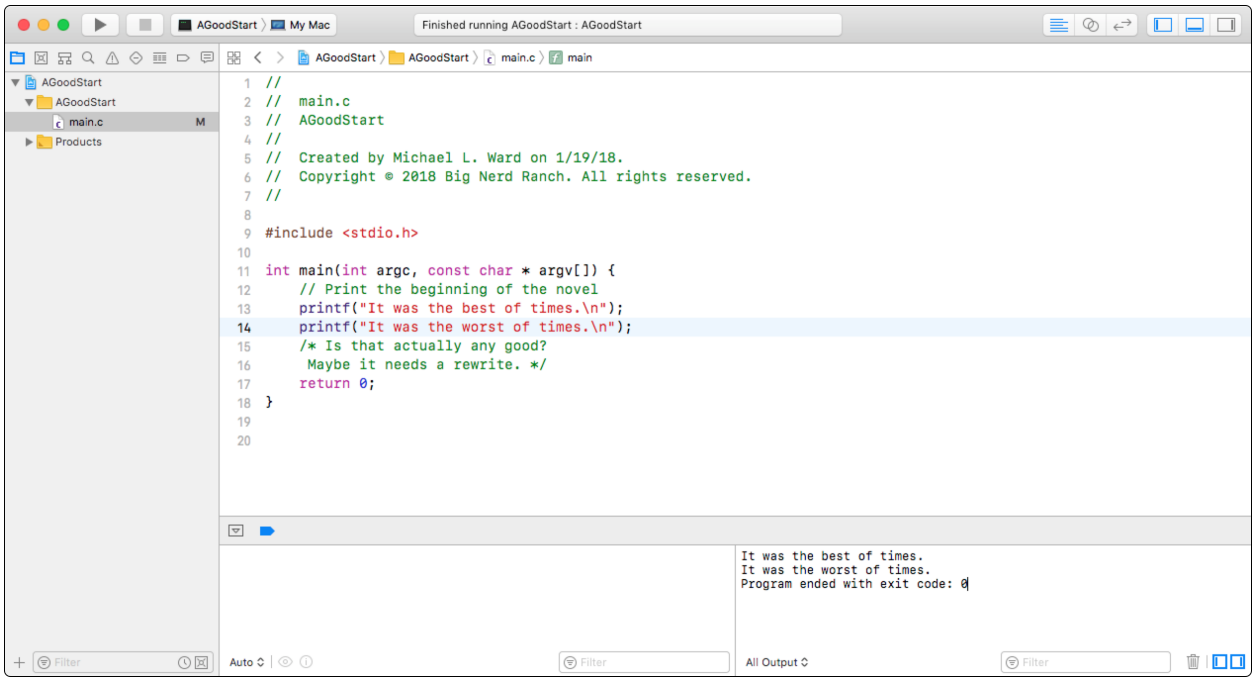
If not, you will get this:

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 that it thinks are problematic. After you find and fix the problem, click the Run button again. Repeat until you have a successful build.

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

After your build has succeeded, a new area will appear at the bottom of the window (Figure 1.6). The right half of this area is the *console*. The console shows the output from your code being executed:

Figure 1.6 Output in console at bottom-right



So, what is a program?

Now that you have built and run your first program, let's take a quick look inside to see how it works.

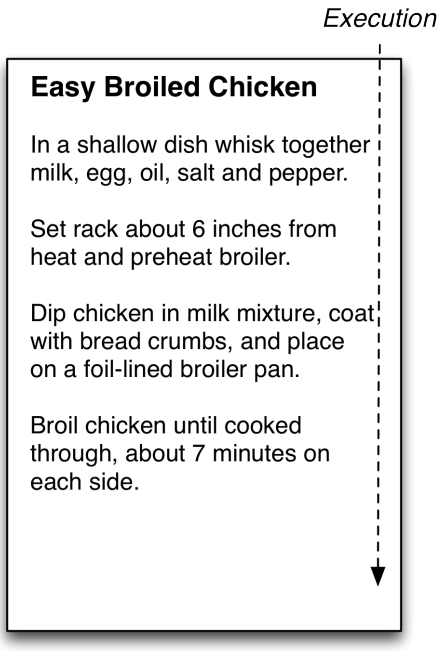
A program is a collection of functions. A function is a list of operations for the processor to execute. Every function has a name, and the function that you just wrote is named **`main`**.

When programmers talk about functions, we usually include a pair of empty parentheses. Thus, the **`main`** function is referred to as **`main()`**.

There was another function in your program – **`printf()`**. You did not write this function, but you did use it.

To a programmer, writing a function is a lot like writing a recipe card. Like a function, a recipe card has a name and a set of instructions. The difference is that you execute a recipe, and the computer executes a function.

Figure 1.7 A recipe card named Easy Broiled Chicken



These 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 you 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 will use these terms interchangeably.

When you run a program, the compiled functions are copied from the hard drive into memory, and the function named **`main`** is executed by the processor. The **`main`** function usually calls other functions. For example, your **`main`** function called the **`printf`** function. You will learn more about how functions work in Chapter 4.

Don't stop

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

Aaron's son Otto is six. Otto is baffled several times a day. He is constantly trying to absorb knowledge that does not fit into his existing mental scaffolding. Bafflement happens so frequently that it does not 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 can be 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 speak 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 is kind of cool. It is a little like being six again.

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

Part II

Enough C to Be Dangerous

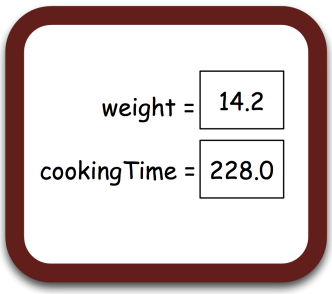
In these next chapters, you will create many programs that demonstrate useful concepts. These command-line programs are nothing that you will show off to your friends, but there should be a small thrill of mastery when you run them. You are moving from computer user to computer programmer.

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 information about programming and programming in C that new-to-programming people need to know before learning Objective-C programming.

2
Variables and Types

Continuing with the cooking 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 2.1 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 is an example of a variable declaration:

```
float weight;
```

The type of this variable is `float` (which we will define in a moment), and its name is `weight`. At this point, the variable does not 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 does not 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 to each type in more detail in later chapters.

`short`, `int`, `long`	These three types are whole numbers; they do not require a decimal point. A `short` usually has fewer bytes of storage than a `long`, and an `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 is usually treated as a character, like the letter `'a'`.
pointer	A pointer holds a memory address. It is declared using the asterisk character. For example, a variable declared as `int *` can hold a memory address where an `int` is stored. It does not hold the actual number's value, but if you know the address of the `int`, then you can get to its value. Pointers are very useful, and there will be more on pointers later. Much more.
`struct`	A `struct` (or *structure*) is a type made up of other types. You can also create new `struct` definitions. For example, imagine that you wanted a `GeoLocation` type that contains two float members: `latitude` and `longitude`. 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 do not accidentally type new code into the old project.

Now create a new project (File → 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;

    // Store 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;
}
```

(Wondering about the `\n` that keeps turning up in your code? You will learn what it does in Chapter 5.)

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. Your output in the console should look like this:

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

Back in your code, let's review what you have done. In the 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 righthand 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 righthand side of the `=`.

For example, in this line:

```
weight = 14.2;
```

the expression is just `14.2`.

An expression can have multiple steps. For example, when evaluating the expression `15.0 + 15.0 * weight`, the computer first multiplies `weight` by `15.0` and then adds that result to `15.0`. Why does the multiplication come first? We say that multiplication has *precedence* over addition.

To change the order in which operations are normally executed, you use parentheses:

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

Now the expression in the parentheses is evaluated first, so the computer first does the addition and then multiplies `weight` by 30.0.

Challenge

Welcome to your first challenge!

Most chapters in this book will finish with a challenge exercise to do on your own. Some challenges (like the one you are about to do) are easy and provide practice doing the same thing you did in the chapter. Other challenges are harder and require more problem-solving. Doing these exercises cements what you have learned and builds confidence in your skills. We cannot encourage you enough to take them on.

(If you get stuck while working on a challenge, take a break and come back and try again fresh. If that does not work, visit the forum for this book at `forums.bignerdranch.com` for help.)

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 `float`s. Print the result using **`printf()`**. Refer to the code in this chapter if you need to check your syntax.

3
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 will not create a project in this chapter. Instead, consider the code examples carefully based on what you have learned in the last two chapters.

Here is 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 do not 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 3.1 shows the common operators used when comparing numbers (and other types that the computer evaluates as numbers):

Table 3.1 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 *not* equal?

The == operator deserves an additional note: In programming, the == operator is what is 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 3.2 shows the three logical operators:

Table 3.2 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 will not discuss it here.)

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

```
// Is it lighter than air?
if (!(truckWeight > 0.0)) {
    printf("The truck has zero or negative weight. Hauling helium?\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.

```
BOOL 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 `BOOL` for boolean variables, so that is what we use here. (`BOOL` is an alias for an integer type.) To use `BOOL` in a C function, like **`main()`**, you would need to include in your program the file where this type is defined:

```
#include <objc/objc.h>
```

You will learn more about including files in the next chapter.

When curly braces are optional

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 does not 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 indention of the second statement? While indention is very helpful for human readers of code, it means nothing to the compiler.)

In this book, we will always include the curly braces.

else if

What if you have more than two possibilities? You can test for them one by one using `else if`. For example, suppose 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 do not 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 is useful when you want to catch everything that did not meet the earlier conditions.

For the more curious: conditional operators

It is not uncommon to 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* because it takes three operands).

```
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.

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?

4
Functions

In Chapter 2, you learned that a variable is 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 is a lot in this chapter – three new projects, a new tool, and many new ideas. Let's get started with an exercise that will demonstrate 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.

Create a new C Command Line Tool named ClassCertificates. (Select File → New → Project... or use the keyboard shortcut Command-Shift-N to get started.)

Your first thought in writing this program might be:

```
int main (int argc, const char * argv[])
{
    printf("Kate 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("Liz 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 have realized that you need a function, you need to write one. Open `main.c` in your ClassCertificates project and write a new function named **`congratulateStudent`**. This function should go just before **`main()`** in the file.

```
#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);
}

int main(int argc, const char * argv[])
{
   ...
```

(Wondering what the `%s` and `%d` mean? Puzzled by the type `char *`? Hold on for now; we will get there.)

Now edit **`main()`** to use your new function:

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

    return 0;
}
```

Build and run the program. Find your output in the console. You may need to resize the bottom area. You can do this by clicking on the area's grey header and then dragging to adjust its size. (You can resize any of the areas in Xcode the same way.)

The output should be identical to what you would have seen if you had typed in everything yourself.

```
Kate 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.
Liz 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 your 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 will 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.

In your first call to **`congratulateStudent()`**, you passed three arguments: `"Kate", "Cocoa", 5`.

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

For now, 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`.

Now, when **`congratulateStudent()`** uses the `numDays` variable within the function, its value will be `5`. 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 the first proposed version of ClassCertificates with all the repetitive typing. What is the point of using a function instead? To save on the typing? Well, yes, but that is definitely not all. Partitioning your code into functions makes it easier to make changes and to find and fix bugs. You can make a change or fix a typo in one place, and it will have the effects you want everywhere you call that function.

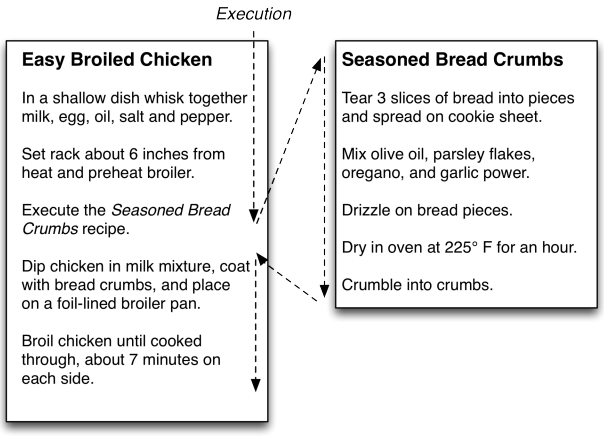
Another benefit to writing functions is reusability. Now that you have written this handy function, you could use it in another program.

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 you began to execute the “Easy Broiled Chicken” card, you would discover that the third instruction says “Execute the Seasoned Bread Crumbs recipe,” which is explained on another card. A programmer would say, “The Easy Broiled Chicken function *calls* the Seasoned Bread Crumbs function.”

Figure 4.1 Recipe cards



Similarly, **`main()`** can call other functions. For example, **`main()`** in ClassCertificates called the **`congratulateStudent()`**, which in turn called **`printf()`**.

While you are preparing the bread crumbs, you stop executing the steps on the “Easy Broiled Chicken” card. When the bread crumbs are ready, you resume working through the “Easy Broiled Chicken” card.

Similarly, the **`main`** function stops executing and “blocks” until the function it called is done executing. To see this happen, you are going to call a **`sleep`** function that does nothing but wait a number of seconds. This function is declared in the file `unistd.h`. At the top of `main.c`, include this file:

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

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

In your **`main`** function, call the **`sleep`** function after the calls to **`congratulateStudent()`**.

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

    return 0;
}
```

Build and run the program. You should see a two-second pause between each message of congratulations. That is because the **`main`** function stops running until the **`sleep`** function is done sleeping.

Standard libraries

Your computer came with many functions built-in. Actually, that is a little misleading – here is the truth: Before macOS was installed on your computer, it was nothing but an expensive space heater. Among the things that were installed as part of macOS were files containing a collection of precompiled functions. These collections are called the *standard libraries*.

Two of the files that make up the standard libraries are `stdio.h` and `unistd.h`. When you include these files in your program, you can then use the functions that they contain. **`printf()`** is in `stdio.h`; **`sleep()`** is in `unistd.h`.

The standard libraries have two purposes:

- They represent big chunks of code that you do not 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 will learn how to browse the documentation for iOS and macOS in class.

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, consider a function that computed how long to cook a turkey. It might look like this:

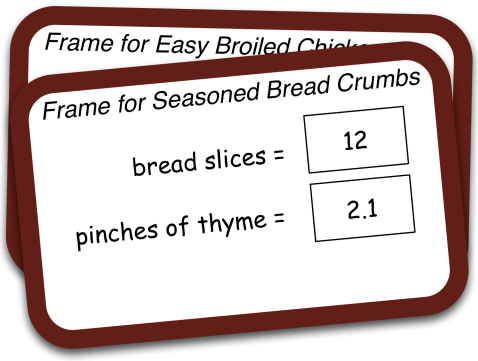
```
void showCookTimeForTurkey(int pounds)
{
   int necessaryMinutes = 15 + 15 * pounds;
   printf("Cook for %d minutes.\n", necessaryMinutes);
}
```

`necessaryMinutes` is a local variable. It will come into existence when **`showCookTimeForTurkey()`** starts 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 gets 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 Easy Broiled Chicken recipe. In your kitchen, each recipe that is in progress gets its own blackboard, so you have a blackboard for the Easy Broiled 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 Easy Broiled Chicken. After all, you have suspended execution of Easy Broiled Chicken to make Seasoned Bread Crumbs. You will not need the Easy Broiled 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 4.2 Two blackboards in a stack



Programmers use the word *stack* to describe where the frames are stored in memory. When a function is called, its frame is pushed onto the top of the stack. When a function finishes executing, we say that it *returns*. That is, it pops its frame off the stack and lets the function that called it resume execution.

Let's look more closely at how the stack works by putting **`showCookTimeForTurkey()`** into a program. Create a new C Command Line Tool named TurkeyTimer. Edit `main.c` to look like this:

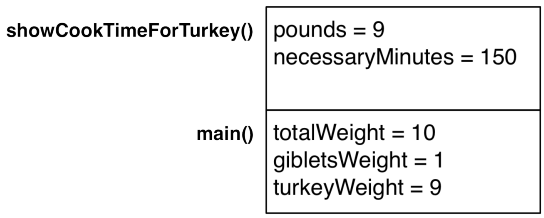
```
#include <stdio.h>

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;
}
```

Build and run the program. You should see the following output:

```
Cook for 150 minutes.
```

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

Figure 4.3 Two frames on the stack



The stack is last-in, first-out. That is, **showCookTimeForTurkey()**'s frame is popped off the stack before **main()**'s frame is popped 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, that is, it is copied into the function's frame.

Scope

In a function definition, any pair of curly braces { ... }) define the *scope* of the code that is in between them. A variable cannot be accessed outside of the scope that it is declared in. In fact, it does not exist outside of the scope that it is declared in.

Any pair of braces, whether they are a part of a function definition, an if statement, or a loop, defines its own scope that restricts the availability of any variables declared within them.

Add the following code to your **showCookTimeForTurkey** function:

```
void showCookTimeForTurkey(int pounds)
{
    int necessaryMinutes = 15 + 15 * pounds;
    printf("Cook for %d minutes.\n", necessaryMinutes);
    if (necessaryMinutes > 120) {
        int halfway = necessaryMinutes / 2;
        printf("Rotate after %d of the %d minutes.\n", halfway, necessaryMinutes);
    }
}
```

Build and run the program.

The **printf** statement in this example can access variables that are in the scope defined by the curly braces of the if statement, like halfway. It can also access variables in the outer scope defined by the **showCookTimeForTurkey** function itself, like necessaryMinutes.

Now move the **printf** call outside of the if statement's scope:

```
void showCookTimeForTurkey(int pounds)
{
    int necessaryMinutes = 15 + 15 * pounds;
    printf("Cook for %d minutes.\n", necessaryMinutes);
    if (necessaryMinutes > 120) {
        int halfway = necessaryMinutes / 2;
        printf("Rotate after %d of the %d minutes.\n", halfway, necessaryMinutes);
    }
    printf("Rotate after %d of the %d minutes.\n", halfway, necessaryMinutes);
}
```

Build and run the program again. The program will not run, and you will get a build error: Use of undeclared identifier 'halfway'. Outside of the if statement's scope, the halfway variable does not exist. Stylish programmers would say that, "When the **printf()** call is made, the halfway variable has fallen out of scope."

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()**:

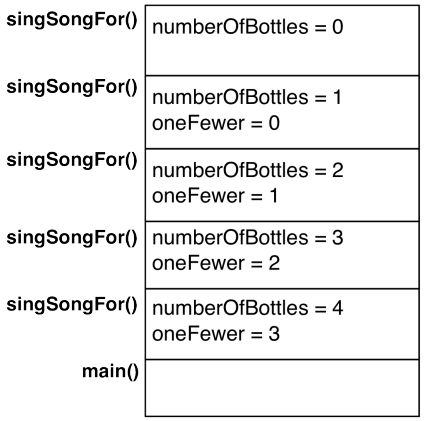
```
#include <stdio.h>

void singSongFor(int numberOfBottles)
{
    if (numberOfBottles == 0) {
        printf("There are simply no more bottles of beer on the wall.\n\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\n",
               oneFewer);
        singSongFor(oneFewer); // This function calls itself!

        // Print a message just before the function ends
        printf("Put a bottle in the recycling, %d empty bottles in the bin.\n",
               numberOfBottles);
    }
}


int main(int argc, const char * argv[])
{

    // We could sing 99 verses, but 4 is easier to think about
    singSongFor(4);
    return 0;
}
```

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

```
4 bottles of beer on the wall. 4 bottles of beer.
Take one down, pass it around, 3 bottles of beer on the wall.

3 bottles of beer on the wall. 3 bottles of beer.
Take one down, pass it around, 2 bottles of beer on the wall.

2 bottles of beer on the wall. 2 bottles of beer.
Take one down, pass it around, 1 bottles of beer on the wall.

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, 3 empty bottles in the bin.
Put a bottle in the recycling, 4 empty bottles in the bin.
```

What does the stack look like when the last bottle is taken off the wall, but none have been put in the recycling bin?

Figure 4.4 Frames on the stack for a recursive function



Confused? Here is what happened:

- **`main()`** called **`singSongFor(4)`**.
- **`singSongFor(4)`** printed a verse and called **`singSongFor(3)`**.
- **`singSongFor(3)`** printed a verse and called **`singSongFor(2)`**.
- **`singSongFor(2)`** printed a verse and called **`singSongFor(1)`**.
- **`singSongFor(1)`** printed a verse and called **`singSongFor(0)`**.
- **`singSongFor(0)`** printed “There are simply no more bottles of beer on the wall.” And returned.
- **`singSongFor(1)`** resumed execution, printed the recycling message, and returned.
- **`singSongFor(2)`** resumed execution, printed the recycling message, and returned.
- **`singSongFor(3)`** resumed execution, printed the recycling message, and returned.
- **`singSongFor(4)`** resumed execution, printed the recycling message, and returned.
- **`main()`** resumed, returned, and the program ended.

Discussing frames and the stack is usually not covered in a beginning programming course, but we have found the ideas to be exceedingly useful to new programmers. First, these concepts give you a more concrete understanding of the answers to questions like “What happens to my local variables when the function finishes executing?” Second, they help 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 frames in the debugger

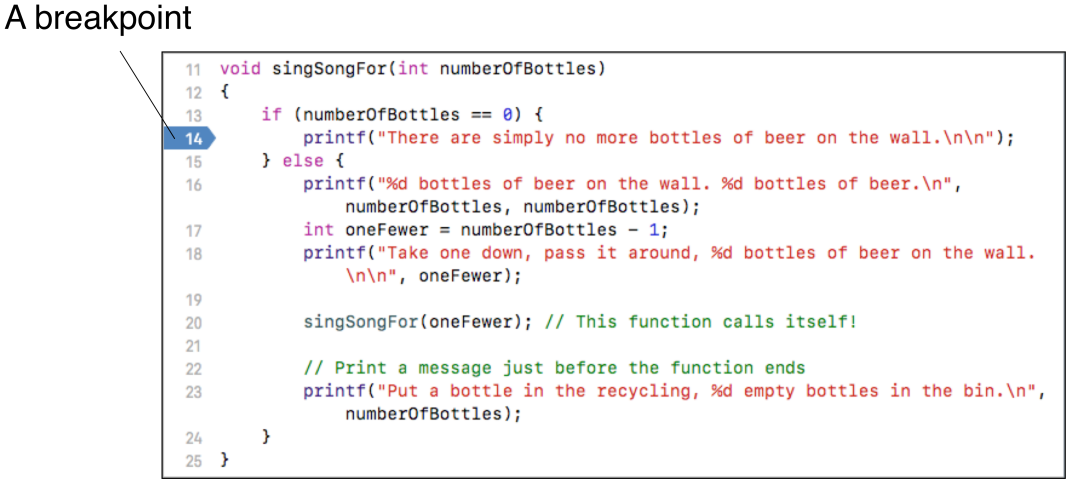
You can use the debugger to browse the frames on the stack. To do this, however, you have to stop your program in mid-execution. Otherwise, **`main()`** will finish executing, and there will not be any frames left to look at. To see as many frames as possible in your BeerSong program, you want to halt execution on the line that prints “There are simply no more bottles of beer on the wall.”

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

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

There is white space to the left of your code. Click near the left edge of the Editor Area (it may have line numbers in it) next to this line of code to set a *breakpoint*.

Figure 4.5 Setting a breakpoint



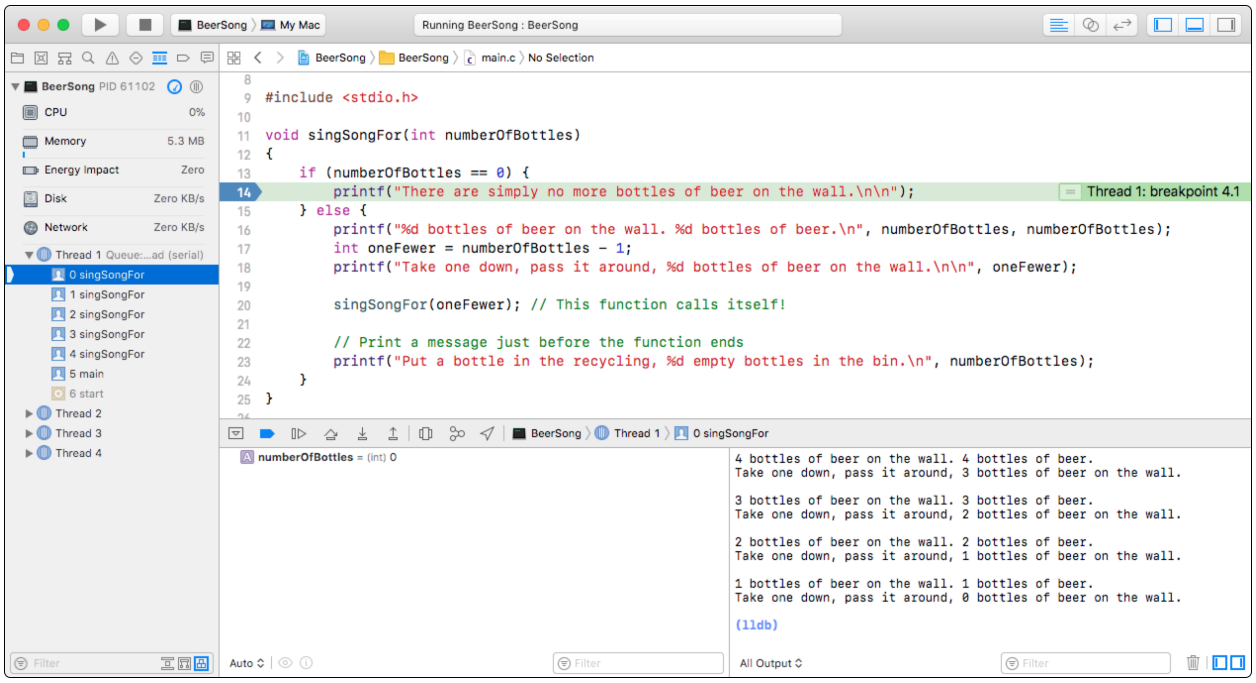
A breakpoint is a location in code where you want the debugger to pause the execution of your program. Run the program again. You can see from the output in the console that your program stopped (or "broke") right before executing the line on which you set the breakpoint.

The program is temporarily frozen, and you can examine it more closely. The navigator area has switched to displaying the *debug navigator*, which 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. Since 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 is the frame for **`main()`**.

You can select a frame from the stack 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 **`singSongFor()`** is called.

Figure 4.6 Selecting frame for `singSongFor(4)`

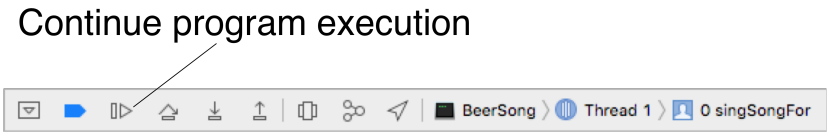


You can see this frame's variables and their values in the bottom area to the left of the console. This area is called the *variables view*.

Now you need to remove the breakpoint so that the program will run normally. Right-click the blue indicator and select Delete Breakpoint.

To resume execution of your program, click the ▶ button on the grey bar above the variables view.

Figure 4.7 Resuming BeerSong



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 does not 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 you take the return value of **`fahrenheitFromCelsius()`** and assign it to the `freezeInF` variable of type `float`? Build and run the program.

The execution of a function stops when it returns. For example, take a look at 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 return 1.

This may seem contradictory to how `0` and `1` work in `if` statements; because `1` is true and `0` is false, it is 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 Degrees:

```
#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 do not give them an initial value, they are automatically initialized to zero.

In this chapter, you have learned about functions. When you 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 is 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
```

5
Format Strings

Now that you know how functions work, let's take a closer look at the **`printf`** function that you have been using to write to the log.

The **`printf`** function accepts a *string* as an argument and prints it to the log. A string is a "string" of characters strung together like beads on a necklace. Typically, a string is text.

A *literal string* is text surrounded by double quotes. In the AGoodStart project from Chapter 1, you called **`printf()`** with literal string arguments:

```
    // Print the beginning of the novel
    printf("It was the best of times.\n");
    printf("It was the worst of times.\n");
```

Your output looked like this:

```
It was the best of times.
It was the worst of times.
```

You can store a literal string in a variable of type `char *`:

```
    char *myString = "Here is a string";
```

This is a *C string*. You could have created C strings in AGoodStart and passed them to **`printf()`**:

```
    // Write the beginning of the novel
    char *firstLine = "It was the best of times.\n";
    char *secondLine = "It was the worst of times.\n";

    // Print the beginning of the novel
    printf(firstLine);
    printf(secondLine);
```

Your output would have looked exactly the same.

Using tokens

The **`printf`** function can do more than just print literal strings. You can also use **`printf()`** to create custom strings at runtime using tokens and variables.

Reopen your ClassCertificates project. In `main.c`, find **`congratulateStudent()`**. Within this function, you call **`printf()`** and pass it a string with three tokens and three variables as arguments.

```
    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);
    }
```

When you pass a string containing one or more tokens to **`printf()`**, the string that you pass is called the *format string*. In this example, the format string includes three tokens: `%s`, `%s`, and `%d`.

When the program is run, the tokens are replaced with the values of the corresponding variable arguments. In this case, those variables are `student`, `course`, and `numDays`. Your output looked something like this:

```
Liz has done as much iOS Programming as I could fit into 5 days.
```

Tokens are replaced in order in the output: the first variable replaces the first token, and so on. Thus, if you swapped `student` and `course` in the list of variables, you would see

```
iOS has done as much Liz Programming as I could fit into 5 days.
```

On the other hand, not all tokens and variables are interchangeable. The token you choose tells **`printf()`** how the variable's value should be formatted. The `%s` token tells **`printf()`** to format the value as a string. The `%d` tells **`printf()`** to format the value an integer. (The `d` stands for "decimal.")

If you use the wrong token, such as using `%d` when the substitution is the string `"Ted"`, **`printf()`** will try to represent `"Ted"` with an integer value, which will give you strange results.

There are other tokens for other types. You will learn and use several more as you continue working through this book.

Escape sequences

The `\n` that you put at the end of your strings is an *escape sequence*. An escape sequence begins with `\`, which is the *escape character*. This character tells the compiler that the character that comes immediately after does not have its usual meaning.

The `\n` represents the new-line character. In **`printf()`** statements, you include a new-line character when you want output to continue on a new line. Try removing one of these new-lines and see what happens to your output.

Another escape sequence is `\"`. You use it when you need to include quotation marks within a literal string. The escape character tells the compiler to treat the `"` as part of the literal string and not as an instruction to end the string. Here is an example:

```
printf("\"It doesn't happen all at once,\" said the Skin Horse.\n");
```

And here is the output:

```
"It doesn't happen all at once," said the Skin Horse.
```

Challenge

Create a new project (C Command Line Tool) named Squarer. Write a program that computes and displays the square of integer. Put the numbers in quotation marks. Your output should look something like this:

```
"5" squared is "25".
```

6
Numbers

You have 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.

Integers

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

To address these different tasks, 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 tasks 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. Why? $2^8 = 256$ possible numbers. And we choose to start at 0.

A signed 64-bit number can hold any integer from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. One bit for the sign leaves $2^{63} = 9,223,372,036,854,775,808$. There is only one zero.

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

```
UInt32 x; // An unsigned 32-bit integer
SInt16 y; // A 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` an 8-bit integer? When C was designed, nearly everyone used ASCII to represent characters. ASCII gave each commonly used character a number. For example, 'B' was represented by the number 66. The numbers went up to 127, so we could easily fit any ASCII character into 8 bits. To deal with other character systems (like Cyrillic or Kanji), we needed a lot more than 8 bits. For now, live with ASCII characters, and we will talk about dealing with other encodings (like Unicode) in class.

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 will talk more about that in Chapter 8.

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 (i.e., 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 you have 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 will return to hexadecimal numbers in class.)

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 assume 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 will get `11 / 3 = 3.666667`, right? Nope. You get `11 / 3 = 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. Add the modulus operator to get a statement that includes the remainder:

```
#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 will get 3.666667. Here is 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.

NSInteger and NSUInteger

At this moment, Xcode supports the development of both 32-bit and 64-bit applications. In an effort to make it easy for you to write code that will work elegantly on either system, Apple introduced `NSInteger` and `NSUInteger`. These are 32-bit integers on 32-bit systems; They are 64-bit integers on 64-bit systems. `NSInteger` is signed. `NSUInteger` is unsigned.

`NSInteger` and `NSUInteger` are used extensively in Apple's libraries, so when you start working in Objective-C, you will use them a lot.

The recommended way of outputting them with **`printf()`** is a little surprising. Because Apple does not want you to make too many assumptions about what is really behind them, it is recommended that you cast them to the appropriate `long` before trying to display them:

```
NSInteger x = -5;
NSUInteger y = 6;
printf("Here they are: %ld, %lu", (long)x, (unsigned long)y);
```

Operator shorthand

All the operators that you have 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 are 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
```

The math library

If you will be doing a lot of math, you will need the math library. To see what is 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. It should be `0.841`. The sine function is declared like this:

```
double sin(double x);
```

A note about comments

As you type in exercises, do not be shy about adding comments of your own to help you remember what code is doing. Get in the habit of commenting code. Write useful and specific comments that could be understood by someone else reading your code or by you in the future coming back to review the code.

Comments can be helpful when you tackle challenges. For example, say you get a challenge working but are not sure it is solved in an elegant way. Leave yourself a note about what bugged you. When you are further along in the book, review old challenges and see if you can improve your solutions. This will also test your ability to write useful comments. Something like

```
// Not sure if this is right
...
```

will be useless to your future self.

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 a program to do that 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 is 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 you will use is a `while` loop. The `while` construct works something like the `if` construct discussed in Chapter 3. 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:

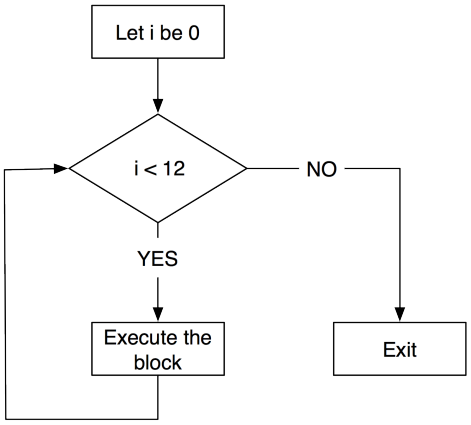
```
#include <stdio.h>

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` was not incremented, then this loop, as written, would continue forever because the expression would always be true.

Here is a flow-chart of this `while` loop:



The for loop

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

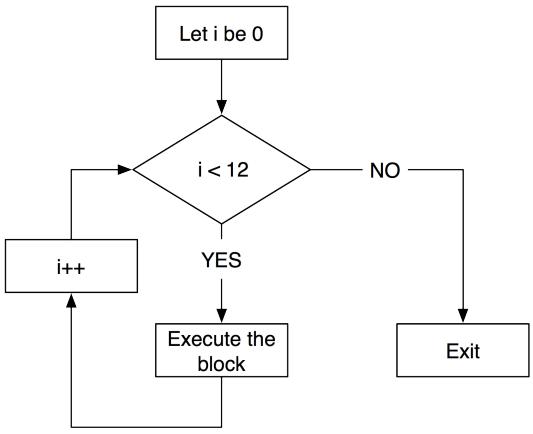
```
some initialization
while (some check) {
    some code
    some last step
}
```

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;
}
```



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, you 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. You will learn more about collections and loops starting in class.

break

Sometimes it is necessary to stop the loop's execution from 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:

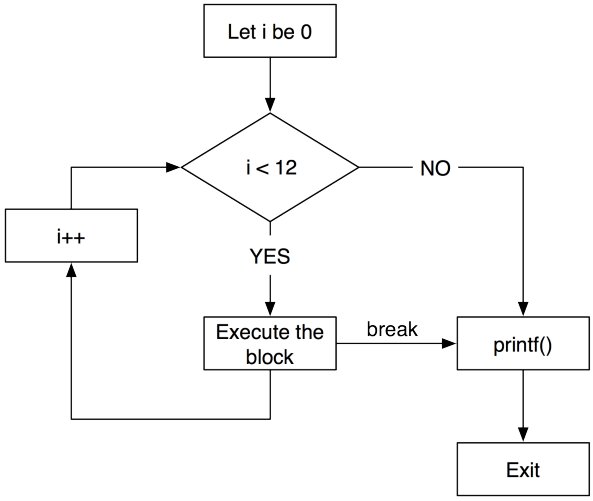
```
#include <stdio.h>

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.



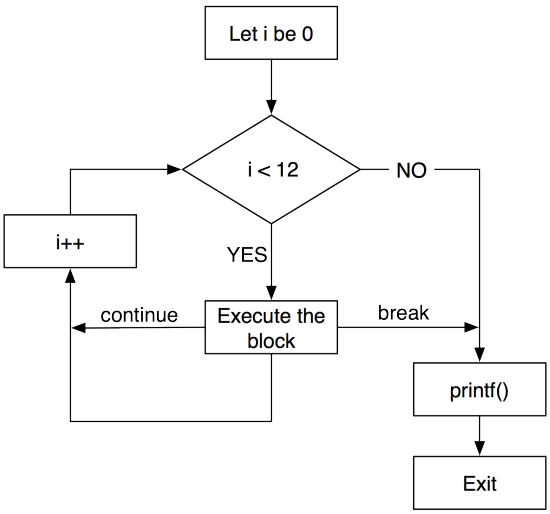
continue

Sometimes you 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.
```



The do-while loop

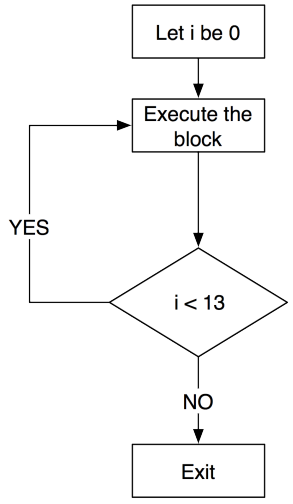
The cool kids seldom use the `do-while` loop, but for completeness, here it is. The `do-while` loop does not check the expression until it has executed the block. Thus, it ensures that the block is always executed at least once. If you rewrote the original exercise to use a `do-while` loop, it would look like this:

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

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

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

Here is a flow-chart of this `do-while` loop:



Challenge: counting down

Create a new project (C Command Line Tool) named CountDown and write a program that counts backward from 99 through 0 by 3, printing each number.

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!
```

Challenge: user input

So far, the programs you have written do some work and then output text to the console. In this challenge, you will modify your CountDown solution from the previous challenge to ask for input from the user. In particular, you will ask the user what number the countdown should start from.

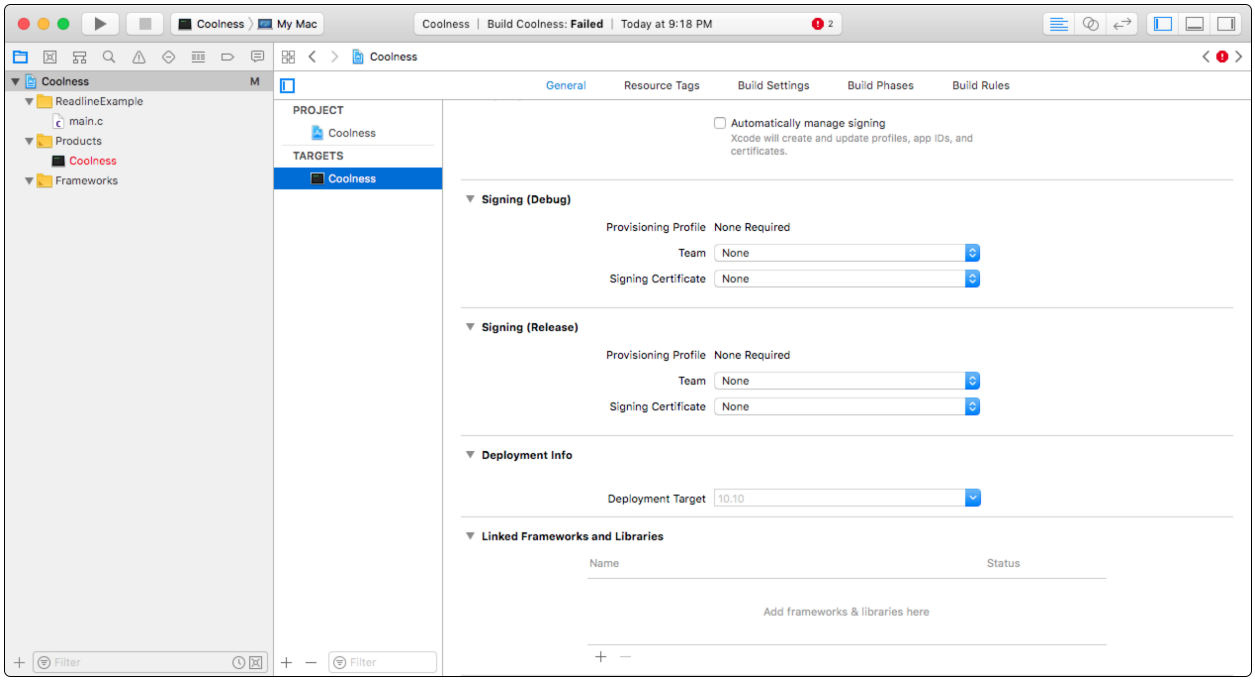
To make this happen, you need to know about two new functions: **`readline()`** and **`atoi()`** (pronounced “A to I”).

The **`readline`** function is the opposite of **`printf()`**. Rather than printing text to the screen, it gets text that user has entered.

Before you can use **`readline()`**, you must first add the library that contains it to your program.

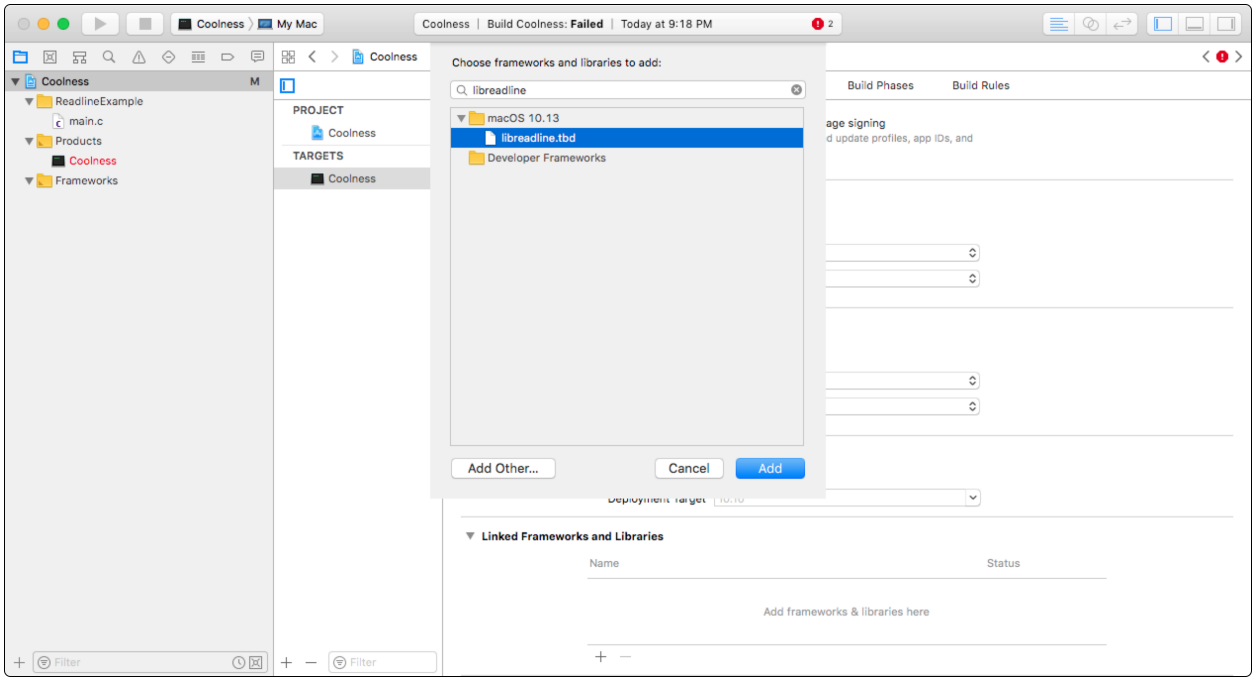
In the project navigator, click the top-level Coolness item. In the editor area, click General and then scroll to the very bottom, to the listing called Linked Frameworks and Libraries.

Figure 7.1 Adding a linked library



Click the + button below the empty list. A sheet will a appear with a list of available code libraries. Use the search box to search for `libreadline`. When it appears in the list, select it and click Add.

Figure 7.2 Libraries



Select `main.c` in the project navigator to get back to your code.

What were these steps for? Sometimes, you want to use a function that is not already provided for you. So you need to tell Xcode which *code library* contains the function you want to use.

Let's look at an example that uses the **`readline()`** function. You started this chapter with code that printed `Aaron is Cool`. What if the user could enter the name of the person that is cool? Here is what the program would look like when run. (The user input is shown in bold)

```
Who is cool? Mikey
Mikey is cool!
```

The code would look like this:

```
#include <readline/readline.h>
#include <stdio.h>
int main(int argc, const char * argv[])
{
    printf("Who is cool? ");
    const char *name = readline(NULL);
    printf("%s is cool!\n\n",name);
    return 0;
}
```

(Type this code into your Coolness project and run it, if you would like to see it in action.)

The first line of this **`main`** function is a variable declaration:

```
    const char *name;
```

Remember that `char *` is a type you can use for strings.

In the third line, you call the **`readline`** function, and pass NULL as its argument. This line gets what the user typed in and stores it in the `name` variable.

Now let's turn to the **atoi** function. This function takes a string and converts it into an integer. (The 'i' stands for integer, and the 'a' stands for ASCII.)

What good is **atoi()**? The following example code would cause an error because it attempts to store a string in a variable of type `int`.

```
int num = "23";
```

You can use **atoi()** to convert that string into an integer with a value of 23, which you can happily store in a variable of type `int`:

```
int num = atoi("23");
```

(If the string passed into **atoi()** cannot be converted into an integer, then **atoi()** returns `0`.)

With these two functions in mind, modify your code to prompt the user for input and then kick off the countdown from the desired spot. Your output should look something like this:

```
Where should I start counting? 42
42
39
36
33
30
Found one!
27
...
```

Note that in order to use the **atoi()** function, you will need to include `stdlib.h` into your program.

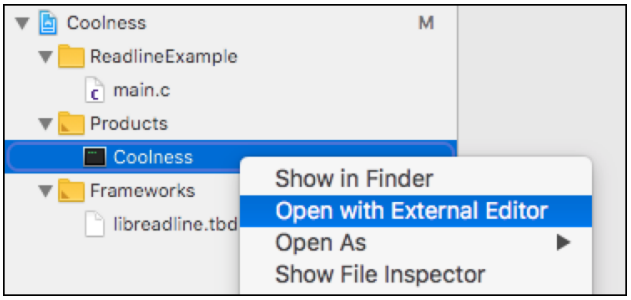
Also note that Xcode has an interesting behavior when using the **readline** function. It will duplicate text input as output:

Figure 7.3 **readline()** output

This is expected behavior in Xcode.

If you right-click on your app's Coolness product in the Products group in the Navigator area in Xcode, you can select Open With External Editor to launch your command-line app using your Mac's Terminal (see Figure 7.4). The Terminal is where command-line apps are usually invoked by Mac users, and does not have the input-echoing behavior of Xcode.

Figure 7.4 Running your app



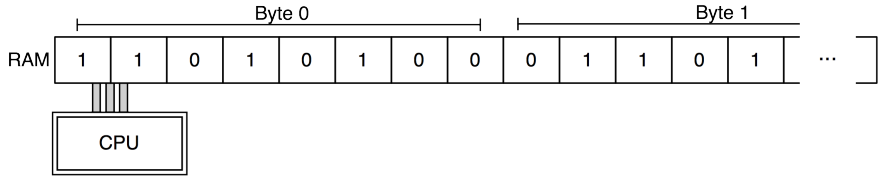
8

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 will 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 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. This is the token that you can replace with a memory address. Build and run the program.

Your output will look something like:

```
i stores its value at 0x7ffeefbff66c
```

although your computer may put `i` at a 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 does not store the value of a `float`; it can hold 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.

You are using integers right now for simplicity. But if you are wondering what the point of pointers is, we 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 is why we pass addresses. It is 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 is 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 in this example:

- When you declared `addressOfI` to be an `int *`. That is, you told the compiler “It will hold an address where an `int` can be stored.”

- When 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.

Do not worry if you do not have pointers squared away in your mind just yet. You will spend a lot of time working with pointers as you go through this book, so you will get plenty of practice.

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;
}
```

Here there is 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.

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 an address 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
}
```

Sometimes `NULL` indicates that there is no value, so you might see something like this:

```
float *measuredGravityPtr = NULL;

// Some code that might set measuredGravityPtr to be non-NULL
…

float actualGravity;

// Did we measure the gravity?
if (measuredGravityPtr) {
    actualGravity = *measuredGravityPtr;
} else {
    actualGravity = estimatedGravity(planetRadius);
}
```

Or, you can use the ternary operator to do the same thing more tersely:

```
// If measuredGravityPtr is NULL, estimate the gravity
float actualGravity =
          measuredGravityPtr ? *measuredGravityPtr : estimatedGravity(planetRadius);
```

Later, when you are learning about pointers to objects, you 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 do not.

Why? You can declare multiple variables in a single line. For example, if you wanted to declare variables `x`, `y`, and `z`, you 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.

A final note: Pointers can be difficult to get your head around at first. Do not worry if you have not mastered these ideas yet. You will be working with them for the rest of the book, and they will make more sense each time you do.

Challenge: how much memory?

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

Challenge: how much range?

On a 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 that a `short` can store? What is the largest?

An `unsigned short` only holds non-negative numbers. What is the largest number that an `unsigned short` 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?

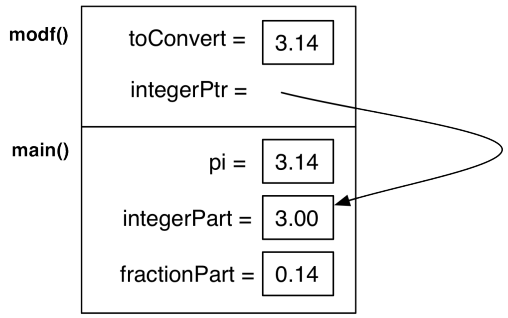
When you call **`modf()`**, you will supply an address where it can stash one of the numbers. In particular, it will return the fractional part and copy the integer part to the address you supply. Create a new project: a C Command Line Tool named PBR.

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 is 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

The world is just awesome. The variety of cultures and peoples around the world inspires a great deal of excellent output from the arts and sciences.

One complication of this diversity is that different people use different units for measuring the world around them. The scientific and engineering communities tend to have a preference for metric units (such as meters) over imperial units (such as feet and inches), due to their ease of use in mathematical calculation.

If you were to write an application for consumption by users in certain parts of the world, however, you might want to be able to print the results of your meter-based calculations using feet and inches.

How would you write a function that converts a distance in meters to the equivalent distance in feet and inches? It would need to read a floating-point number and return two others. The declaration of such a function would look like this:

```
void metersToFeetAndInches(double meters, unsigned int *ftPtr, double *inPtr);
```

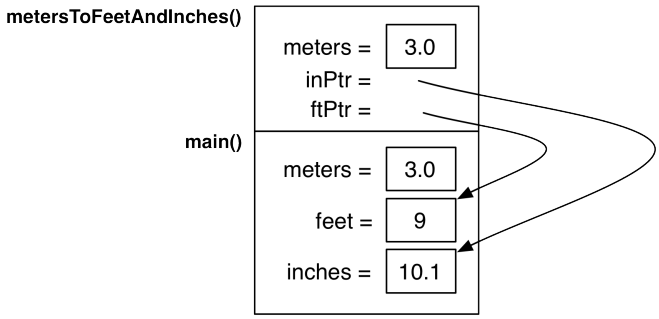
When the function is called, it will be passed a value for `meters`. It will also be supplied with locations where the values for `feet` and `inches` can be stored.

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

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

void metersToFeetAndInches(double meters, unsigned int *ftPtr, double *inPtr)
{
    // This function assumes meters is non-negative.

    // Convert the number of meters into a floating-point number of feet
    double rawFeet = meters * 3.281; // e.g. 2.4536

    // How many complete feet as an unsigned int?
    unsigned int feet = (unsigned int)floor(rawFeet);

    // Store the number of feet at the supplied address
    printf("Storing %u to the address %p\n", feet, ftPtr);
    *ftPtr = feet;

    // Calculate inches
    double fractionalFoot = rawFeet - feet;
    double inches = fractionalFoot * 12.0;

    // Store the number of inches at the supplied address
    printf("Storing %.2f to the address %p\n", inches, inPtr);
    *inPtr = inches;
}


int main(int argc, const char * argv[])
{
    double meters = 3.0;
    unsigned int feet;
    double inches;

    metersToFeetAndInches(meters, &feet, &inches);
    printf("%.1f meters is equal to %d feet and %.1f inches.", meters, feet, inches);

    return 0;
}
```

Build and run the program.

Figure 9.2 The stack as **`metersToFeetAndInches()`** 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 are not going to use them anyway? Typically, you pass `NULL` as an address to tell the function "I do not 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 **`metersToFeetAndInches()`**:

```
void metersToFeetAndInches(double meters, unsigned int *ftPtr, double *inPtr)
{
    // This function assumes meters is non-negative.
    // Convert the number of meters into a floating-point number of feet
    double rawFeet = meters * 3.281; // e.g. 2.4536

    // How many complete feet as an unsigned int?
    unsigned int feet = (unsigned int)floor(rawFeet);

    // Store the number of feet at the supplied address
    if (ftPtr) {
        printf("Storing %u to the address %p\n", feet, ftPtr);
        *ftPtr = feet;
    }

    // Calculate inches
    double fractionalFoot = rawFeet - feet;
    double inches = fractionalFoot * 12.0;

    // Store the number of inches at the supplied address
    if (inPtr) {
        printf("Storing %.2f to the address %p\n", inches, inPtr);
        *inPtr = inches;
    }
}
```

Challenge

In **`metersToFeedAndInches()`**, you used **`floor()`** and subtraction to break `rawFeet` into its integer and fractional parts. Change **`metersToFeedAndInches()`** to use **`modf()`** instead.

10
Structs

Sometimes you need a variable to hold several related chunks of data. In C, you can do this with a *structure*, commonly called a *struct*. Each chunk of data is known as a *member* of the struct.

For example, consider a program that computes a person's Body Mass Index, or BMI. BMI is a person's weight in kilograms divided by the square of the person's height in meters. (BMI is a very imprecise tool for measuring a person's fitness, but it makes a fine programming example.)

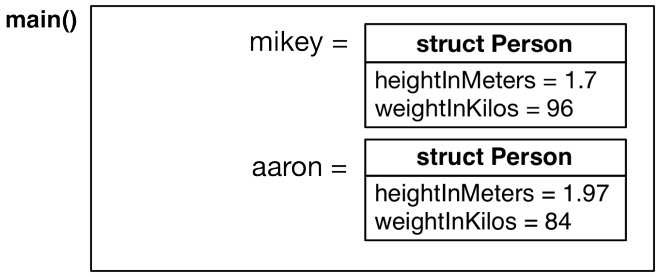
Create a new project: a C Command Line Tool named BMICalc. Edit `main.c` to declare a struct named `Person` that has two members: a `float` named `heightInMeters` and an `int` named `weightInKilos`. Then create two `Person` structs:

```
#include <stdio.h>

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

int main(int argc, const char * argv[])
{
    struct Person mikey;
    mikey.heightInMeters = 1.7;
    mikey.weightInKilos = 96;

    struct Person aaron;
    aaron.heightInMeters = 1.97;
    aaron.weightInKilos = 84;

    printf("mikey is %.2f meters tall\n", mikey.heightInMeters);
    printf("mikey weighs %d kilograms\n", mikey.weightInKilos);
    printf("aaron is %.2f meters tall\n", aaron.heightInMeters);
    printf("aaron weighs %d kilograms\n", aaron.weightInKilos);
    return 0;
}
```

Notice that you access the members of a struct using a period (stylish programmers like to say "dot"). Build and run the program and confirm the output.

Here is 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 struct declaration over and over again. So it is common to create a `typedef` for the struct type. A `typedef` defines an alias for a type declaration and allows you to use it more like the usual data types. Change `main.c` to create and use a `typedef` for `struct Person`. Notice that the code to replace is shown struck-through.

```
#include <stdio.h>
// Here is the declaration of the struct
struct Person {
    float heightInMeters;
    int weightInKilos;
    };

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

int main(int argc, const char * argv[])
{
    struct Person mikey;
    Person mikey;
    mikey.heightInMeters = 1.7;
    mikey.weightInKilos = 96;

    struct Person aaron;
    Person aaron;
    aaron.heightInMeters = 1.97;
    aaron.weightInKilos = 84;

    printf("mikey is %.2f meters tall\n", mikey.heightInMeters);
    printf("mikey weighs %d kilograms\n", mikey.weightInKilos);
    printf("aaron is %.2f meters tall\n", aaron.heightInMeters);
    printf("aaron weighs %d kilograms\n", aaron.weightInKilos);
    return 0;
}
```

You can pass a `Person` to another function. Add a function named **`bodyMassIndex()`** that accepts a `Person` as a parameter and calculates BMI. Then update **`main()`** to call this function:

```
#include <stdio.h>


// Here is the declaration of the type Person
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[])
{
    Person mikey;
    mikey.heightInMeters = 1.7;
    mikey.weightInKilos = 96;

    Person aaron;
    aaron.heightInMeters = 1.97;
    aaron.weightInKilos = 84;

    printf("mikey is %.2f meters tall\n", mikey.heightInMeters);
    printf("mikey weighs %d kilograms\n", mikey.weightInKilos);
    printf("aaron is %.2f meters tall\n", aaron.heightInMeters);
    printf("aaron weighs %d kilograms\n", aaron.weightInKilos);

    float bmi;
    bmi = bodyMassIndex(mikey);
    printf("mikey has a BMI of %.2f\n", bmi);

    bmi = bodyMassIndex(aaron);
    printf("aaron has a BMI of %.2f\n", bmi);

    return 0;
}
```

Here you create a local variable `bmi` to hold the return value of **`bodyMassIndex()`**. You retrieve and print out the Mikey's BMI. Then you reuse the variable to retrieve and print out Aaron's BMI.

Challenge

The first struct 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_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);
```

Your challenge is to write a program that will tell you what the date (`4-30-2018` 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 used one kind of memory – frames on the stack. Recall that every function has a frame where its local variables are stored. This memory is automatically allocated when a function starts and automatically deallocated when the function ends. In fact, local variables are sometimes called *automatic variables* because of this convenient behavior.

Sometimes, however, you need to claim a contiguous chunk of memory yourself – a *buffer*. Programmers often use the word buffer to mean a long line of bytes of memory. The buffer comes from a region of memory known as the *heap*, which is separate from the stack.

On the heap, the buffer is independent of any function's frame. Thus, it can be used across many functions. For example, you could claim a buffer of memory intended to hold some text. You could then call a function that would read a text file into the buffer, call a second function that would count all the vowels in the text, and call a third function to spellcheck it. When you were finished using the text, you would return the memory that was in the buffer to the heap.

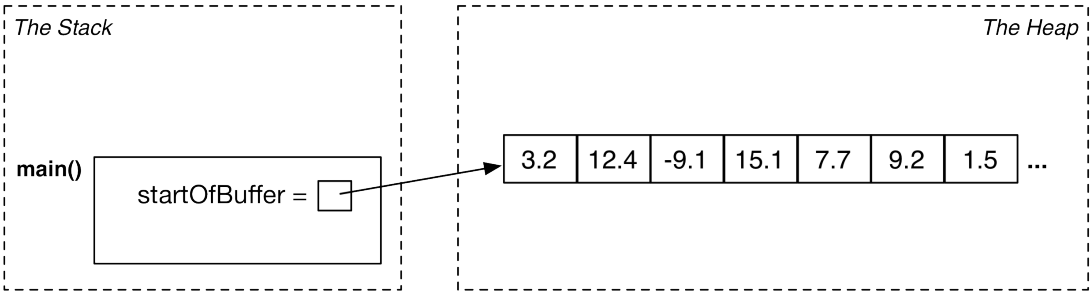
You request a buffer of memory using the function **`malloc()`**. When you are done using the buffer, you call the function **`free()`** to release your claim on that memory and return it to the heap.

Let's say, for example, you needed a chunk of memory big enough to hold 1,000 floats. Note the crucial use of **`sizeof()`** to get the right number of bytes for your buffer.

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

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` is a pointer to the address of the first floating point number in the buffer.

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



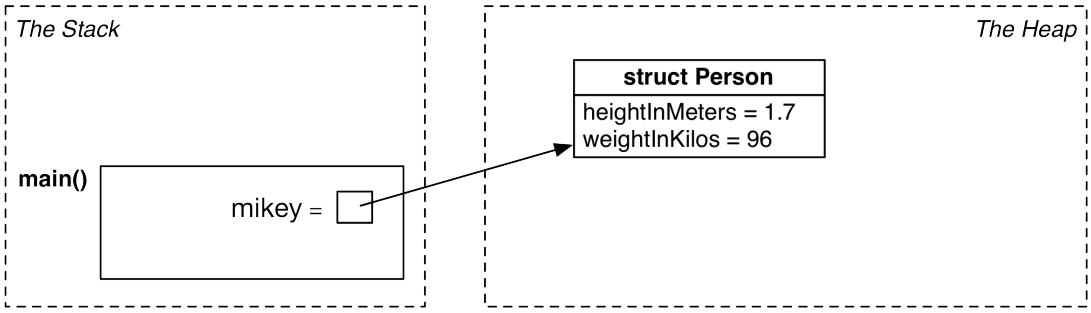
At this point, most C books would spend a lot of time talking about how to use `startOfBuffer` to read and write data in different locations in the buffer of floating pointer numbers. This book, however, is trying to get you to objects as quickly as possible. So, we will put off these concepts until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

In Chapter 10, you created a struct as a local variable in **`main()`**'s frame on the stack. You can also allocate a buffer on the heap for a struct. To create a `Person` struct on the heap, you could write 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 struct
    Person *mikey = malloc(sizeof(Person));

    // Fill in two members of the struct
    mikey->weightInKilos = 96;
    mikey->heightInMeters = 1.7;

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

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

    // Forget where it was
    mikey = NULL;

    return 0;
}
```

Notice the operator `->`. The code `p->weightInKilos` says, "Dereference the pointer `p` to the struct it references, and get me the struct's member called `weightInKilos`."

Figure 11.2 A pointer on the stack to a struct on the heap



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