Beginning C Programming

Nathan Sidles

INTRODUCTION

This manual will teach you how the basics of computer programming in a computer programming language called C.

This manual will teach you how to teach your computer.

Computer programming is the art of instructing a computer how to do a thing. These instructions can be simple things, like adding one and two and storing the result, or complex things, like playing chess – anything, really. The things you can teach your computer to do are limited only by your imagination and your patience. All you have to do is learn its language.

This manual is designed to teach you that language. In chapter one, we're going to start with the basics of how computers talk to themselves and each other; in chapter two, we'll learn how to create sentences in one of your computer's languages; then in succeeding chapters we'll learn how to use different kinds of words and then string those words together to create a story, or in other words, a computer program!

And then we'll be done.

PREREQUISITES FOR THIS MANUAL

You need to have three things before you begin this manual. They are, in order of importance:

1. **Desire**. You have to want to learn how to create beautiful and useful things.
2. **No fear**. We're going to take things to the max in this manual. It is going to be extreme. You can't be afraid.
3. **A basic familiarity with computers**. You need to know how to use a computer for everyday things, like using a mouse, opening a file, and playing Snood.

What if I'm afraid of my computer?

Apple co-founder Steve Wozniak once said, "Never trust a computer you can't throw out a window."

Do you have a computer? Do you have a window? Do you have enough money to buy a new computer? If so, then you have the advantage.

As you go through this manual, remember that computer programming is an art, not a science. It is open-ended, like painting a portrait, composing a symphony, or writing a poem. Remember: what you create is bounded only by your imagination and your patience.

So! Are you ready to begin?

Then let's go!

TABLE OF CONTENTS

CHAPTER 1: BACKGROUND INFORMATION

This chapter is not essential.

It only explains the reasons for the things we're going to do. You can skip it if you're in a hurry.

Computer programming is the art of instructing a computer how to do a thing. This is trickier than it sounds.

Computers **do not speak English**. They speak in magnetism and electricity. In order to instruct a computer, you must translate your desires from English ("I want you to open a window, Mr. Computer, and I want you to do it *now*.") to a language the computer will understand. In order to instruct a computer, you must bridge the divide between man and machine.

Programming languages **bridge this divide**. They turn your specific instructions (that almost look like English) into magnetism and electricity.

In order to use these languages efficiently, you should understand how they work: how computers "talk" and how computers "listen." This chapter addresses these subjects, as well as introducing C, the specific programming language that we'll be learning.

This chapter is divided into three sections:

So! Are you ready to take the **Nestea Plunge** into these pools of knowledge?

Then let's go!

SECTION 1A: HOW DO COMPUTERS TALK?

Computers think, talk, and remember in magnetism and electricity.

Computers use the orientation of little tiny magnets to store data as either a "0" or as a "1" (depending on which side of the magnet is facing up). These 0s and 1s are the fundamental level of computer data storage. Each 0 or 1 is called a **bit** (short for "**b**inary dig**it**").

Enough 0s and 1s can store almost any information, if you can figure out how to encode it.

A computer's hard drive stores trillions of these bits, and the computer reads them as needed. The computer's processor takes the bits that it reads and interprets them as direct instructions to its circuits. These instructions dictate a few tiny tasks billions of times per second: send an electric signal down this wire, send an electric signal down that wire, flip this magnet from 0 to 1, and so on.

These billions of **tiny tasks** combine to complete millions of **small tasks**: turn this pixel this particular shade of white, remember the character that is being typed, calculate the multiple of these two numbers, and so on.

These many **small tasks** combine to complete a **complicated task**: a window displaying text.

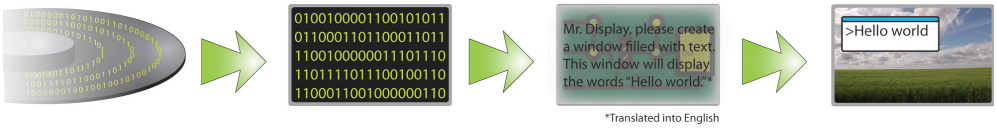

Figure 1. How a computer does a thing.

SECTION 1B: HOW DO COMPUTERS LISTEN?

Computers must do more than talk; they must also listen. What if you want to teach your computer something new? What if you want to **slip the surly bonds of Earth** and dance the skies?

Then you have to talk to speak a language that your computer understands or can translate: a computer programming language.

Computers basically "listen" to computer programming languages in the **reverse** of how they "talk": first, they read one big complicated task (that you specify) and turn it into many small tasks, and then into billions of tiny tasks, the instructions to which are then stored as bits on the computer's hard drive.

A computer would be confused by both of these commands:

Bring me the cow and fish.

Birng me the cow and the fish.

Why?

The first command is ambiguous: "fish" is both a noun and a verb. The second command is misspelled.

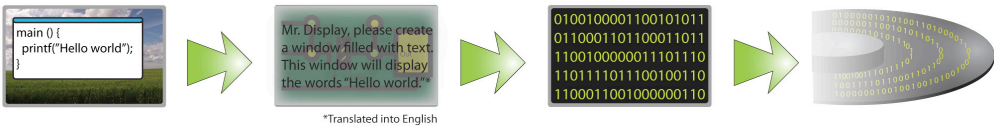

Figure 2. How a computer learns a thing.

Writing out that first big complicated task in a way that makes sense to the computer is the **great challenge of computer programming**, and it is the purpose of computer programming languages.

At this point in the manual, please turn the page.

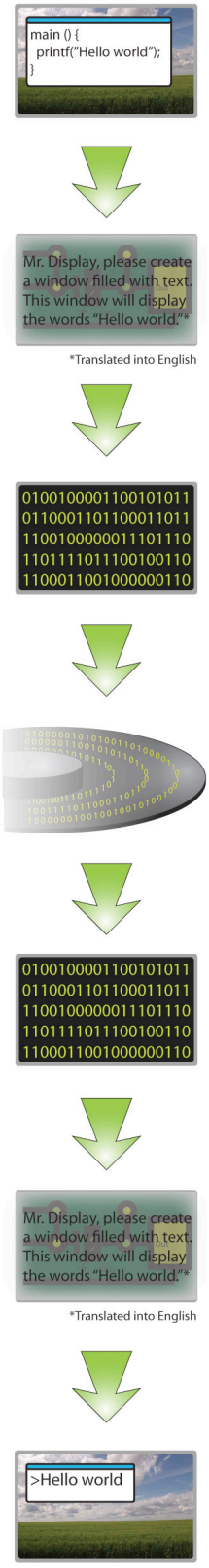

Figure 3. How C works.

That instruction ("please turn the page") was spoken in a language that you understand. If it had said, "please ," then you would be baffled. Is that a command? An insult? A come-on? A threat? Gibberish?

A computer works the same way that you do. Computer programming languages are languages that can be translated into commands that a computer can understand and execute.

These languages are **precise** and **rigorous**: precise, because each thing you say to them must have only a single meaning; and rigorous, because each thing you say to them must be said exactly correctly.

Thus, instructing a computer is both **harder** and **easier** than instructing a human. It is harder because computers speak their own languages. It is easier because within those languages there are no ambiguities. When you know how to instruct a computer properly, it will always do exactly what you say.

SECTION 1C: THE C PROGRAMMING LANGUAGE

The language that we'll be instructing our computer in is called C. We'll be learning C because it's simple, foundational, and efficient.

You create a program in C by writing (in plain text) a series of commands, instructions, and defintiions with the proper grammar and syntax. This text file is turned into processor instructions by a program called a "compiler." The computer then runs this program after it is compiled. Figure 3 illustrates this process.

C is a simple, sensible language. It was created in the early 1970s, it is very easy to learn, and many other programming languages are based on it and add to it.

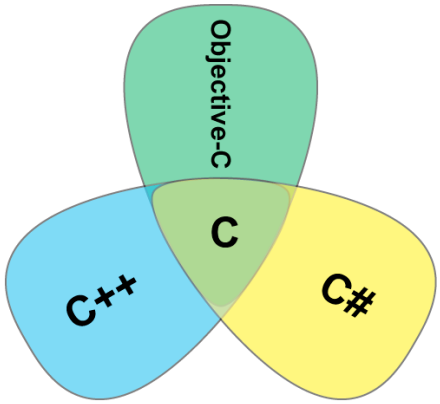

Figure 4. Languages based on C.

Programming languages like the ones in figure 4 are supersets of C. They can do everything C can, only more easily – if C were basic English, C++ might add mathematical symbols, while Objective-English might add proper nouns, and so on. Many more language share C's syntax, language, and grammar.

C and its supersets and derivatives have been used to create everything from calculators to operating systems. Learn C and you can **conquer the universe**.

Our next chapter begins this journey.

CHAPTER 2: BASIC C CONCEPTS

We're going to be learning how to program in a programming language called **C**.

Writing code in C is a lot like writing English: you need to create a story, the story of a little program that opens a window, or plays chess, or flies an airplane, or calculates numbers. This story must have **characters, action, a beginning, and an end**.

This chapter introduces you the C equivalents of all these English things. This chapter shows you how C is structured as a programming language, how you can organize a C program to do the things you want, how to create your first C program, and some general tips and tools of the trade.

It is divided into six sections:

So! Are you ready to begin?

Then let's go!

SECTION 2A: PARTS OF A C PROGRAM

There are six basic things (for want of a better word) in C programming that you are going to use: notes, variables, operators, functions, statements, and libraries. Each of them is analogous to a part of English:

- **Notes**. Notes are the sections of your code that are invisible to the compiler. They are like notes you can write in the margins of a book. They are described in section 2D below.
- **Functions**. Functions are commands to action for your program. They are like sentences or paragraphs. They are described in chapter 3.
- **Variables**. Variables are containers for your program's data. They are like nouns (or collective nouns). They are described in chapter 4.
- **Operators**. Operators are actors or definers of your program's variables. They are like verbs. They are described in chapter 5.
- **Statements**. Statements are control structures for your program's flow. They are like branches, or the page choices of the "Choose Your Own Adventure" books. They are described in chapter 6.
- **Libraries**. Libraries are collections of code and data. They are like references for scientific papers. They are described in chapter 7.

This part is really important.

These six things can have only **three actions** happen to them: they can be declared, defined, or called. Declaring something announces to your program its nature. Defining something tells your programs its value and what it does. Calling something tells your program to use that thing in the manner you demand.

Operators and statements are already defined as part of the C programming language. Libraries are defined and called in the same statement.

SECTION 2B: C PROGRAM ORGANIZATION

All C programs use these six things in various permutations and combinations, but always with strict rules of order and precedence. It is almost like poetry.

The structure is this:

1. Preprocessing directives (**libraries**).
2. Global **variable** declarations or definitions.
3. **Function** declarations.
4. The "main" **function** definition.
    a. Local **variable** definitions or declarations.
    b. All **function** calls.
    c. **Variable** calls.
    d. **Operators**.
    e. **Statements**.

The "main" function starts and encompasses the program's flow. All functions are called within it. All C programs must have it. See chapter 3 for more information.

5. **Function** definitions.
    a. Local **variable** definitions or declarations.
    b. **Function** calls.
    c. **Variable** calls.
    d. **Operators**.
    e. **Statements**.

And that's all! Notes can go anywhere.

SECTION 2C: CREATING YOUR FIRST PROGRAM

These things need to work together: most functions need variables, most variables should have functions, and so on. We will be using a basic program to introduce these concepts, but that program will include a lot of stuff you don't understand.

So! You must now create your first C program so you can see what's in it.

Here's how you do it:

See the Appendix for how to set up and use your C compiler.

1. Open your C compiler.
2. Create a new C program file.
3. Type the following text:

```
#include <stdio.h>
 int main (void) {
   printf ( "Hello world!\n" );
 return 0;
}
```

This program follows the organization of section 2B: the preprocessing directive ("#include <stdio.h>"), the function definition ("int main void { ... }"), and the function call within the function definition ("printf (...)").

You'll be using that preprocessing directive and that main function definition a lot before we really explain what they do. We apologize for the inconvenience.

4. Compile and run your program.

And you're done!

You will see a window appear that says "Hello world!" If not, see section 2F below.

SECTION 2D: ADDING NOTES TO YOUR PROGRAM

C is written in text and is structured like a novel, but that doesn't always make it easy for humans to read. Notes are a way to add information in C files that the computer will not interpret as part of the program, but which you can see. Adding notes is often essential to understanding your programs.

Notes in C are exactly like notes here, except less cutesy.

Notes are added to C files by prefixing information with "//".

Here's how you do it:

1. Open your C compiler.
2. Create a new C program file.
3. Type the following text:

```
#include <stdio.h>

int main (void) {
   printf ( "Hello world!\n" );
 return 0;
}
```

4. Add a comment into the second line:

```
#include <stdio.h>

//This is a comment outside a function!

int main (void) {
  printf ( "Hello world!\n" );
 return 0;
}
```

5. Compile and run your program.

   And you're done!

   You will see a window appear that says "Hello world!" If not, see section 2E below. Note that your comment did not appear in the "Hello world!" window, though it is still there in your C file.

SECTION 2E: PROGRAMMING BEST PRACTICES

Now you have made your first program and you have added your first comments! You know the basic things in C and you know what they can do in the most basic way.

Before you go further, you should learn some programming practices that will save you time and frustration later on:

1. **Document what you do**. Use the note feature we learned in the last section to make notes of what each part of your code does. It knows, but you may not.
2. **Group like things together**. You can define functions and variables wherever, but it's best to do it all in one place. That way, you can change them if need be.
3. **Format your program consistently**. C ignores spaces and blank lines. This means you can indent parts of your program differently. All information inside a function, for example, should be indented.

4. **Keep a consistent naming and capitalizing style**. You can name your variables and functions whatever you like, but if you keep a consistent naming and capitalizing style, you can always tell what is what.

   InterCap is the standard convention. It separates new words in variable names by capital letters. The first letter is capitalized for functions and uncapitalized for variables.

5. **Make backups**. Your program deserves to live forever. Sometimes you may need to revert to an earlier version, before you messed everything up.

You don't have to do these things. Your code will run without them and you might even save some time, but your mother may cry a little bit when she decompiles it.

SECTION 2F: ERROR MESSAGES YOU WILL RECEIVE

There are two kinds of errors you can make in C: semantic errors and syntax errors. The good news is that there are only two kinds of errors, and that neither is going to destroy your computer.

There is no bad news.

A syntax error would be the command "Desloy the missiles." Is it deploy the missiles or destroy the missiles?

A semantic error would be the command "Deploy the missiles" when you have not defined a missile variable.

A **syntax error** is an error of typing (or knowledge). You will have included an invalid character, forgotten a semi-colon, or leave a bracket or two unopened. These errors are typically minor and sprinkled like daises through your code. Weed them out! Daisies are not a commercial crop.

A **semantic error** is an error of structure. It is an infinite loop, a self-reference, a paradox. This does not typically result in a rampant A.I., but just in case, your computer will refuse to compile any code containing these things.

You will make errors in your code. Your compiler will usually tell you when you have made an error, as well as its kind and location, but not always. If the error is not actually destructive, minor, or is just amusing, your program will compile without any error message.

CHAPTER 3: FUNCTIONS

Functions are what make C programs come alive. They are what make your program actually do things. They can be surprisingly complex: functions within functions within functions, bounded only by your imagination and your memory.

Recall from the last chapter (page 8) that there are three things that you can do with functions: you can declare them, you can define them, and you can call them.

Functions must either be declared or defined before they are called.

This chapter introduces functions, gives you their place and use in C programs, gives you special functions for user input, and shows you some examples of how they can be used.

It is divided into four sections:

SECTION 3A: DECLARING A FUNCTION

Declaring a function is not necessary if you are calling a function *after* you have defined it, but it is good practice to declare all your functions at the top of your program anyway.

Functions are introduced with a function declaration. A function declaration serves as a placeholder for your function. If your program encounters a call to that function before that function is defined, that function declaration tells it to scan the program for that definition later on.

A function declaration has three parts:

- **Its return type**. A function's return type is the kind of data it will output. For more information on return types, see chapter 4.
- **Its name**. A function's name is how it is referenced by the program. This name should be made in accordance with the conventions on page 10.
- **Its parameter list**. A function's parameter list is a list of the variables that can affect the function's output.

A parameter list of "void" means that your function has no inputs.

Here's an example:

```
int SayHello( void )
```

"Int" is the return type, "SayHello" is the name, and "void" is the parameter list.

SECTION 3B: DEFINING A FUNCTION

Functions are defined with a function definition. A function definition describes the function. They look similar to function declarations, but actually contain the details of what the function does.

A function definition has five parts:

The return type is a variable type. See chapter 4 for details on variable types. A return type of "void" means the function returns nothing.

- **Its return type**.
- **Its name**.
- **Its parameter list**.
- **Its arguments**. A function's arguments are the actions taken by that function. They can be the creation of variables
- **Its return value**. This is the data that the function sends back to the larger program after the function has run. A function can either return the value of one variable or nothing at all.

The return value is actually part of your function's arguments, but is so vital that it deserves a section of its own.

"Return 0" means the function returns nothing.

Here's an example:

```
int SayHello( void ) {
  printf ( "Hello world!\n" );
 return 0;
}
```

"Int" is the return type, "SayHello" is the name, "void" is the parameter list, "{ ... printf("Hello world"); ... }" is the argument (notice that the arguments are inside a set of curvy brackets), and "return 0" is its return value.

When called in your program, this function will take no inputs and will return no values, but will create a window that says “Hello world” in it.

SECTION 3C: CALLING A FUNCTION

Functions are brought into use with function calls. A function call is what actually allows your program use the function. Function calls have only one part:

- Its name.

Here’s an example:

```
SayHello();
```

“SayHello” is the name. “()” is where you would add any parameters you would want to specify, but “SayHello” has none.

SECTION 3D: CREATING A FUNCTION

Now you know how to declare a function, how to define a function, and how to call a function. This section will walk your through the steps of doing each of these three things.

Here’s how to do it:

1. Open your C compiler.
2. Create a new C program file.
3. Type the following text:

Recall from the previous chapter that this “main” function is the main function for your program. It starts the whole process, either directly or indirectly. All functions must be called in it. Those functions can then call other functions, but it all starts with the “main” function.

```
#include <stdio.h>

int main ( void ) {
   YourFunction ();
 return 0;
}
```

This includes part of the C Standard Library (see chapter 7 about libraries) and the main function that calls your function.

4. Type the following text on the next line:

```
int YourFunction( void )

int YourFunction( void ) {
  printf ( "Your name!\n" );
 return 0;
}
```

This program is very simple. It will display your name on the screen. Notice that YourFunction() is calling another function. This and other functions are built into the C standard library.

5. Compile and run your program!

   And you're done!

   Check it out! Does it display a window that displays "Your name"?

CHAPTER 4: VARIABLES

Variables are the nouns in the story of your program. They are its most basic items, and the building blocks of your empire. They can exist independently of functions (though they don't do very much).

Variables store data. Some of them store a lot of data, some of the store just a little *bit* (har!). They are a foundational part of the C programming language.

Variables come in different types. The type of a variable determines what can be put into it. Just like the collective noun "flock" can refer only to birds (or bats, or dinosaurs), a variable of type "int" can only store integers.

The four basic types of variables are discussed in section A. There are many more complex kinds of variables, but the four basic kinds are enough to get you started.

This chapter discusses the types of variables, how to declare a variable, how to define a variable, how to call a variable, and how to create a function with variables.

It is divided into five sections:

SECTION 4A: TYPES OF VARIABLE

There are four basic types of variable*:

- **char** variables can store a string of characters, like the letters "Hello."
- **int** variables can store integer numbers, like 1, 2, 3, or 31,549.
- **float** variables can store floating point numbers (fractions of integers), like 2.5 or 1.125.
- **double** variables can a wider range of floating point numbers than float variables.

***This is a lie!**

There are many exciting types of variables with many exciting features, but they're complicated and we won't be learning about them.

Each of these variables can store this kind of data and no others. If you try to make the value of a char variable act like an int variable, there is going to be trouble.

Now that we know the four basic types of variable (and to be honest, only three look interesting), we can actually go through the syntax of creating them!

But how?

SECTION 4B: DECLARING A VARIABLE

Variables are introduced with a variable declaration. Just like with functions, you can declare a variable before you define it. This is necessary, because you may have a function use a variable before you get a chance to define it. If you declare all your variables up at the top of your program, this isn't an issue.

A variable declaration has two parts:

- **Its type**. A variable's type is the type of data it will store (see the previous section).
- **Its name**. A variable's name is how it is referenced by the program. This name should be made in accordance with the conventions on page ten.

Here's an example:

```
int variableOne;
```

SECTION 4C: DEFINING A VARIABLE

Variables are defined with a variable definition. This definition assigns an initial value to the equation.

A variable definition has three parts:

- **Its name**.
- **The basic assignment operator ("=")**. The basic assignment operator allows you to assign a value to this variable. For more information on operators, see chapter 5.
- **Its value**. This is the actual data that is stored in the variable.

Here are a couple of examples:

```
integerOne = 1;
integerTwo = 234;
```

SECTION 4D: CALLING A VARIABLE

Variables are brought into use with variable calls. A variable call is what actually allows your program use the function. Function calls have only one part:

- **Its name**.

Here's an example:

```
integerOne;
```

integerOne is the name of the variable. It does not really do anything without being linked to an action.

SECTION 4E: CREATING A FUNCTION WITH VARIABLES

Now that we know the four basic types of variable (and to be honest, only three look interesting), we can actually go through the syntax of creating them!

Here's how to do it:

1. Open your C compiler.
2. Create a new C program file.
3. Type the following text:

Recall from the previous chapter that this "main" function is the main function for your program. It starts the whole process, either directly or indirectly. All functions must be called in it. Those functions can then call other functions, but it all starts with the "main" function.

```
#include <stdio.h>
 int main ( void ) {
   YourSecondFunction ();
 return 0;
}
```

This includes part of the C Standard Library (see chapter 7 about libraries) and the main function that calls your function.

4. Type the following text on the next line:

Notice that yourName is a char type variable

```
int YourSecondFunction( void );
char yourName;

yourName = "Whatever your name is";

int YourSecondFunction( void ) {
  printf ( yourName );
 return 0;
}
```

This program is very simple. It includes the main function calling another function. This other function calls the variable yourName, which is declared and defined in the two rows above.

Notice that YourSecondFunction() is calling a number of other functions. These other functions are built into the C standard library.

6. Compile and run your program!

   And you're done!

   Check it out! Does it display a window that displays whatever you typed into the yourName definition line?

CHAPTER 5: OPERATORS

Operators are things like +, -, and =. They act on variables and change them. **Basically, operators allow variables to talk to one another**.

Your computer already knows what they are. They do not need to be declared or defined. They only need to be used in the appropriate place and for the appropriate purpose.

There are many kinds of operator in C. The simplest kinds are arithmetic operators and relational operators. **Arithmetic operators** preform arithmetic functions on variables. **Relational operators** compare variables to constants and other variables. There are many other different kinds of operator available in C. Many of them do strange and exotic things, like changing a variable's type.

We won't be learning about those.

This chapter deals only with mathematical and relational operators. It lists these operators and describes what they do, and then gives you a chance to play with them on your own.

It is divided into two sections:

SECTION 5A: A TABLE OF DIFFERENT OPERATORS

This section introduces the basic kinds of arithmetic and relational (or conditional operators).

The basic arithmetic operators are:

Operator Name	Syntax	Description
Basic Assignment	a = b	Assigns variable or constant to variable
Addition	a + b	Adds variable or constant to variable
Subtraction	a – b	Subtracts variable or constant from variable
Multiplication	a * b	Multiplies variable by variable or constant
Division	a / b	Divides variable by variable or constant
Increment	a++	Increases value of variable by 1
Decrement	a--	Decreases value of variable by 1

These operators change the value of variables. You can use these in conjunction with each other: b = a + 1, etc.

The basic relational (or conditional) operators are:

Operator Name	Syntax	Description
Equal to	a == b	0 if a equals b; 1 if a does not equal b
Not equal to	a != b	0 if a does not equal b; 1 if a equals b
Greater than	a > b	0 if a is greater than b; 1 if b is not greater than a
Less than	a < b	0 if a is less than b; 1 if b is not less than a
Greater than or equal to	a >= b	0 if a is greater than or equal to b; 1 if b is not greater than or equal to b
Less than or equal to	a <= b	0 if a is less than or equal to b; 1 if b is not less than or equal to b

These operators check the value of variables against constants and other variables. If the condition is true, then the operation assigns a value of 0 to the result. If the condition is false, then the operation assigns a value of 1 to the result.

This ability comes in useful in the next chapter, statements.

SECTION 5B: CREATING A FUNCTION WITH OPERATORS

Now that we know about operators, we can use them to create a series of neat functions. (Not really, but neat stuff is coming in the next chapter, I swear).

Here's how you do it:

1. Open your C compiler.
2. Create a new C program file.
3. Type the following text:

Recall from the previous chapter that this "main" function is the main function for your program. It starts the whole process, either directly or indirectly. All functions must be called in it. Those functions can then call other functions, but it all starts with the "main" function.

```
#include <stdio.h>
 int main ( void ) {
   YourThirdFunction ();
 return 0;
}
```

This includes part of the C Standard Library (see chapter 7 about libraries) and the main function that calls your function.

4. Type the following text on the next line:

```
int YourThirdFunction( void );

int myInt;
int myInt2;
int myInt3;

myInt = 1;
myInt2 = 2;
myInt3 = 3;

int YourThirdFunction( void ) {
  printf ( myInt + myInt2 – myInt3 );
 return 0;
}
```

This program is very simple. It includes the main function calling another function. This other function calls the three variables declared and defined in the rows above, adds two of them together, and subtracts the third.

Notice that YourThirdFunction() is calling a number of other functions. These other functions are built into the C standard library.

7. Compile and run your program!

And you're done!

Check it out! Does it display a window that displays the value 0?

CHAPTER 6: STATEMENTS

Statements in C can control the **order in which your functions are run**, **how your functions interact**, and whether your program is a **stable, ongoing concern or something that's run just once**.

Consider the following example: a man walks up to a door and tries to open it. If it is unlocked, he walks into the house. If it is locked, he waits outside for ten minutes and then tries again.

Statements in C can duplicate this scenario: the function "walk into the house" is called only if the "door" variable is defined as "unlocked." Otherwise, the function "wait outside for ten minutes until trying again" is called.

This chapter talks about the different kinds of statements, and then walks you through the process of adding each kind to your code.

This chapter is divided into two sections:

SECTION 6A: TYPES OF STATEMENT

There are three basic types of statement about which we're going to learn:

- **IF statements**. IF statements take the value of a variable and check it against a condition. If the check is true, the IF statement runs one function. If the check is false, the IF statement runs another function or does nothing.
- **FOR statements**. FOR statements take the value of a variable, set a condition against which that value is checked, and then change the value of the initial variable after running through a series of functions. When the initial condition is met, the FOR statement stops.
- **WHILE statements**. WHILE statements are very similar to FOR statements. They establish a condition using the value of a variable. As long as that condition is met, they will call a function or series of functions. One of those functions changes (or should change) the value of the variable. When the condition is no longer met, the WHILE statement stops.

You can nest these statements inside each other from here to infinity, if you want.

There are other kinds of statement that do things differently in subtle ways, but we're going to pretend like they don't exist for the rest of this manual.

And if you're thinking that statements look like a good way to create programs that run in circles forever, you're right. They are the most powerful tool we've introduced to date. We are not become Death, Destroyer of Worlds, but we might become Annoyance, Annoyer of Computers.

So let's go through how each of them can be integrated into your program!

SECTION 6B: CREATING A FUNCTION WITH STATEMENTS

Don't panic! Remember, you can always set this manual aside and do something wholesome instead.

Now here's the section where we create a function with statements. To save room, we're going to create a function with all three kinds of statement in it at once.

Here's how you do it:

1. Open your C compiler.
2. Create a new C program file.
3. Type the following text:

```
#include <stdio.h>
 int main ( void ) {
   YourFourthFunction ();
   YourFifthFunction ();
   YourSixthFunction ();
 return 0;
}
```

Recall from the previous chapter that this "main" function is the main function for your program. It starts the whole process, either directly or indirectly. All functions must be called in it. Those functions can then call other functions, but it all starts with the "main" function.

This includes part of the C Standard Library (see chapter 7 about libraries) and the main function that calls your function.

4. Type the following text on the next line:

```
int YourFourthFunction( void );

int myInt;
int myInt2;
int myInt3;

myInt = 1;
myInt2 = 2;
myInt3 = 3;

int YourFourthFunction( void ) {
  if (myInt > 2)
   printf( “This integer is less than two!” );
  else
   printf( “This integer is more than two!” );
 return 0;
}

int YourFifthFunction( void ) {
 for (myInt2; myInt2 < 5; myIn2++) {
  printf( “For looping: %d\n”, myInt2 );
 }
 printf( “This loop has been passed!\n” );
 return 0;
}

int YourSixthFunction( void ) {
 while (myInt3 < 5) {
  printf( “While looping: %d\n”, myInt3 );
  myInt3 = myInt 3 + 1;
 }
 printf ( “This loop has been passed!\n” );
 return 0;

}
}
```

This program is very simple. It includes the main function calling another function. This other function calls the three variables declared and defined in the rows above, adds two of them together, and subtracts the third.

Notice that YourThirdFunction() is calling a number of other functions. These other functions are built into the C standard library.

8. Compile and run your program!

And you’re done!

Check it out! Does it display three groups of numbers along with cutesy sayings?

CHAPTER 7: LIBRARIES AND FURTHER INFORMATION

Congratulations! You've reached the last chapter of this manual, and (surprise!) it's an easy one.

You already know exactly what to do to create masterful programs. You can already create simple games (albeit mathematical games), and you now have the blocks to build great things.

Many people before you have wanted to build great things, and so many tools have come into being to help them do it **more easily**. Foremost among these tools are collections of data called "libraries," but there are many more.

This chapter talks about libraries, how to connect them to your programs, and how to go further in your education in C programming.

It is divided into three sections:

SECTION 7A: LIBRARIES

C can reference other files to get more data for a program. The most common group of files it references is called the Standard Library. This library provides many common, built-in functions that allow you to communicate with your computer in diverse ways. You don't have to build functions to create windows that display text or receive user input (though you certainly can, with enough patience), you have them already built for you. The function that we've called in all our functions, printf(), is part of the Standard Library.

SECTION 7B: ADDING LIBRARIES TO A PROGRAM

But what if you actually want to add a library to your program?

Here's how to do it:

1. Open your C compiler.
2. Create a new C program file or open an existing program.
3. Type the following text into the beginning of your program:

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

The "#" sign tells your program to search for this library in all standard folders. The "include" command tells your program to include this in the program before compiling and running it. The less than and greater than signs tell your program that the name of the library is within, and "stdio.h" is the name of the library!

stdio.h is the part of the Standard Library for all C programs. There are many other files that are part of the Standard Library, and all of them are useful.

And that's all! All objects, variables, functions, and data from that library will be added to your program, ready for your functions to use.

SECTION 7C: GOING FURTHER

There are **many more things** to learn before you become a C master, but you have already encountered most of the basics.

The next steps, if you are interested, are learning about options called pointers and parameters (they extend the abilities of variables and functions), learning about advanced variable types (like Boolean variables, where the values are true or false) learning more operators (including logical and assignment operators) and more about preprocessing directives. You also can learn additional functions in the Standard Library. If you're interested in learning more, check out the acknowledgements section on the next page.

The basics, though, are all here. And now **they're all in your mind**.

Good day and good luck to you all.

ACKNOWLEDGEMENTS AND FURTHER READING

This manual used a **number of resources** to create itself. It recommends them all if you're interested in **learning more**:

- Allain, A., Hoffer, A., & Kern, M. (2008, February 21). ***C programming: Programming tutorials – C, C++, OpenGL, STL***. Retrieved from: http://www.cprogramming.com/tutorial.html

  This is an open, HTML-based tutorial for programming in C. It is also used to introduce concepts in C-related languages, and is an excellent resource for those who want to go all the way.

- King, K. N. (2008). ***C programming: A modern approach***. New York, NY: W. W. Norton & Company.

  This is a book designed for C programmers of Windows operating systems, though it is open to all programmers. It provides material in a thorough, conservative way.

- Lahtinen, E., Ala-Mutka, K., & Järvinen, H. M. (2005). **A study of the difficulties of novice programmers**. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*. New York, NY: ACM.

  This book helped the authors learn ways to effectively communicate concepts to beginning programmers, as well as devise strategies for teaching them known problematic concepts.

- Mark, D. (2010). ***Learn C on the Mac***. New York, NY: Apress.

  I'm not going to lie. This book has totally saved my ass. It is a comprehensive examination of basic C concepts from start to finish, and is designed for people who want to code on the Macintosh

- Milne I. & Rowe, G. (2002). **Difficulties in learning and teaching programming – Views of students and tutors**. *Education and Information Technologies, 7*(1), 55-66.

  This article helped create learning strategies for beginning programmers, much like the previous study.

- Young, R. (2010, June 3). **How computers work**. Retrieved from: http://www.fastchip.net/howcomputerswork/p1.html

  This website clarified how computer programming techniques on the processor and bit level. It was very helpful in explaining how computer chips work in the most basic way. Quite entertaining, too.

APPENDIX: SETTING UP YOUR COMPUTER

A compiler turns the C code you write into a language that your computer can understand and execute (as discussed in chapter 1).

Compilers are not standard on computers; they must be downloaded and installed.

There are a number of really great (free) compilers trying to attract your attention for all kinds of operating systems. We're going to go with the most official of all versions for Macintosh computers, the one created by Apple.

This works because the code you write is translated by different compilers into different languages, depending on the operating system.

This manual uses the Macintosh version for all examples, but **the steps and the code are the same for C compilers of all operating systems**, and your code will run the same.

Here's how to get your Macintosh compiler up and running:

1. Go to http://developer.apple.com/mac.
2. Click through the Macintosh developer program checklist.
3. Download the file.
4. Open the .dmg file that downloads and say yes to anything reasonable to which it says you must agree.

   If you are unsatisfied with the terms of the end-user license agreement, contact Apple. They may be willing to make an exception.
5. Once the program has installed, open it.
6. Choose "New Project" from the file menu.
7. Type the following code into the big window and save it with an appropriate name:

   Printf ("This is a Macintosh C compiler!")

   Congratulations! You have just written your first program! Unless you did chapter 2 first.
8. Click the "Build and Go" button near the top toolbar.
9. Confirm that the end result is a window that says "This is a Macintosh C compiler!"

   If it is not, consult the error window that appears instead and take appropriate action.

   And you're done!