

# Appendix B

## Coding 101

### Topics in this Appendix:

- Introduction
- Programming Concepts
- Introduction to C
- Introduction to Debugging
- Introduction to Assembly Language
- Additional Reading



## Introduction

Programming languages are essential to all computers. The electrical components provide the infrastructure and the operating system gives us a framework to play in, but without programs, a computer is just a whirring chunk of plastic and metal. To be understood, a program needs to be written in a some programming language or another, just as a book is written in English or Japanese or Esperanto.

Most programming languages are *imperative* languages. In an imperative language, we give the computer a set of instructions and all the steps necessary to execute those instructions. Programming languages sit on a spectrum that ranges from *low-level* to *high-level*. It's not negative to call something a low-level language—it just means that the lines of code are relatively close to the actual commands being executed at the hardware level. A high-level language has more layers of abstraction. When we program in a high-level language, our code might not look at all like the actual instructions being executed by the computer. A compiler or interpreter takes care of converting our code into instructions the computer can understand. This appendix discusses programming from high-level C to low-level assembly, peeling each layer away like the layers of an onion.

### NEED TO KNOW... LIMITATIONS OF THIS APPENDIX

---



This appendix will not to turn you into a C or assembly language programmer. But it will teach you enough about the structures of these two languages so that you can start to find your way around. Most high-level languages you'll encounter will feel very similar to C and will differ primarily in the specific commands and syntax. If you ever need to learn an *object-oriented* language such as Java, you'll have some extra concepts to study, but many of the basic principles will be the same. If you want to do more advanced programming in these languages, look at the suggestions for further reading at the end of this appendix. You'll find yourself writing complex programs in no time!

---

## Programming Concepts

In this section, we explore some of the essential concepts necessary for any programming language:

- Assignment
- Control structures (looping, conditional branching, and unconditional branching)
- Storage structures (structures, arrays, hash tables, and linked lists)
- Readability (comments, function and variable names, and pretty printing)

These concepts will serve you well for the specific programming languages we're learning here: C and assembly language. But, they are also important general concepts you'll need in any language you might learn, such as C++, Java, or Perl.



## Assignment

*Assignment* occurs when your program stores some information in memory so you can use it later. To get to the information, you need some kind of handle for later access. Frequently we use *named variables*.

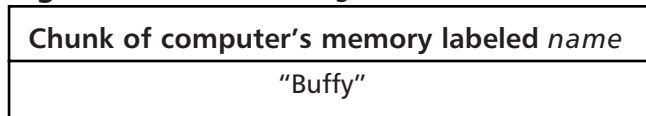
Let's say we would like to greet the user of our program by name. We might write a program that stores the user's name in a variable called `<name>`. The command to print the greeting might look something like:

```
print out "Hello, <name>."
```

When the program runs, the computer will recognize `<name>` as a named variable and will look in its memory in the spot labeled *name* to find the character string. The variable corresponds to the string *Buffy* in the example shown in Figure B.1. The computer will then perform variable substitution and put the string *Buffy* where it saw the name of the variable:

```
Hello, Buffy.
```

**Figure B.1** Variable Assignment



Many programming languages make you *declare* variables before you use them for the first time. To declare a variable is to tell the program "I intend to use a variable with a certain name and of a certain type." Declarations allow the program to set aside enough space in memory to store all your variables.

### NEED TO KNOW... VARIABLE DECLARATIONS IN PSEUDOCODE



Programming languages have different ways of declaring and using named variables, some of which are introduced in this appendix. The `<variable-name>` syntax we use is one you'll often see in *pseudocode*. Pseudocode is a way of presenting coding examples if you aren't sure that all your readers will be using the same programming language or if you don't want to worry about whether or not you are placing the commas and semicolons in exactly the right places. You can't compile or run pseudocode, but if you know a programming language, you can easily convert a pseudocode example into a real example in the programming language you know. Pseudocode works because most imperative programming languages share the same features. The examples in this section are written in pseudocode.



## 544 Appendix B • Coding 101

If our name greeting example were written in C, it would appear as follows:

```
#include <stdio.h>

main()
{
    /* Initialize the user's name */
    char name[] = "Buffy";

    /* Print the user's name */
    printf("Hello, %s\n", name);
}
```

## Control Structures

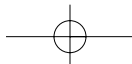
It would be pretty difficult to write a program if we had to tell the computer every single instruction and exactly when we wanted the computer to implement that instruction. Say we wanted to write a program to display the words “Hello, world” on the computer screen. If we had to tell the computer every instruction, our program might look something like this:

```
Print out "H".
Move the cursor right a few pixels.
Print out "e".
Move the cursor right a few pixels.
Print out "l".
...
```

The same program would be written in C as follows:

```
#include <stdio.h>

main()
{
    printf("H");
    printf("e");
    printf("l");
    ...
}
```



Even this simple program assumes that the computer already knows how to display each letter on the screen! To make programming easier, languages use looping, conditional branching, and unconditional branching.

## Looping

*Looping* allows you to execute the same lines of code multiple times. Perhaps you want to say “Hello, world” five times. You could write the line of code *print out ‘hello, world’* five times, or you could use a looping structure to tell your program to run your one line of code multiple times:

five times, print out 'hello, world'

Which would produce the output:

```
hello, world
hello, world
hello, world
hello, world
hello, world
```

This program would be written in C as follows:

```
#include <stdio.h>

main()
{
    int counter;    /* initialize the counter to integer */

    for ( counter = 0; counter < 5 ; counter++ )
        printf("hello, world\n");
}
```

## Conditional Branching

*Conditional branching* allows you to tell your program what to do if certain conditions are met. For example, we might write a program that says goodbye to you at the end of the day. At 5:00 P.M. Monday through Thursday, we want our program to say “Goodnight. See you tomorrow!” But at 5:00 P.M. Friday, we want our program to say “Have a great weekend!” Our pseudocode might look something like:

If today is Monday, Tuesday, Wednesday, or Thursday, then print out "Goodnight. See you tomorrow!"



## 546 Appendix B • Coding 101

Or

If today is Friday, then print out "Have a great weekend!"

This program would be written in C as follows:

```
#include <stdio.h>

main()
{
    char weekday;

    /*
     * Some code goes here to set "weekday" based on the current day
     */

    switch (weekday)
    {
        case 'M', 'T', 'W', 'R':
            printf("Goodnight. See you tomorrow!\n");
            break;
        case 'F':
            printf("Have a great weekend!\n");
            break;
    }; /* Finished with the switch statement */
}
```

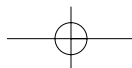
The most common conditional branching structures are *if/then/else* statements, like the one in this example, and conditionals built into *loops* (which execute some lines of code until the following conditions are met).

## Unconditional Branching

*Unconditional branching* allows you to tell your program what to do when a certain line is reached, without any conditions. Some blocks of code might be run many times. Unconditional branching allows you to write that frequently used code in some convenient location outside the main body of your program, storing it as a *procedure* or *function*. When you need to execute that block of code, you can branch to that block wherever it exists.

Here's an example:

```
<planet> = world                [comment: assignment]
for five times
    print out "hello <planet>"
```





```
finish loop                                [comment: looping]
call function <day>                         [comment: unconditional branching]
begin function <day>
    if today is monday
        print out "happy monday"
    else
        print out "aren't you glad it isn't monday?"
    finish if                               [comment: conditional branching]
finish function <day>
```

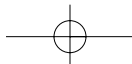
If run on a Monday, this program will display:

```
hello world
hello world
hello world
hello world
hello world
happy monday
```

This might seem a bit complicated. On the other hand, this program would be written in C as follows:

```
#include <stdio.h>
main()
{
    int counter;                            /* declaration */
    char planet[] = "world";                /* declaration & assignment*/
    char today;                              /* declaration */
    void day();                              /* function declaration */
    for ( counter = 0; counter < 5; counter++ )
    {
        printf("hello %s\n", planet);
    }                                        /* finished looping */

    day();                                  /* unconditional branching */
}
```



**548 Appendix B • Coding 101**

```
void day()
{
    /*
     * Some code goes here to set "weekday" based on the current day
     */
    if (today == 'M')
    {
        printf("happy monday\n");
    }
    else /* conditional branching */
    {
        printf("aren't you glad it isn't monday?\n");
    }
};
```

## Storage Structures

When a computer program is running, it is usually processing large amounts of information. It stores that information in the computer's memory. The problem for a computer programmer is how best to store the information. If every piece of information the program needs were just written willy-nilly into the computer's memory, it would be very difficult for the programmer to recall that information when it is needed—not to mention slow for the computer to find it! To solve this problem, programmers use *storage structures*—software components that simplify information storage. These structures are sometimes symbolic, existing primarily in your mind as you write your code, without your programming language being aware of them at all. We'll see more how this works when we look at C in more detail.

Four of the most important storage structures are:

- Structures
- Arrays
- Hash tables
- Linked lists

### NEED TO KNOW... A NOTE ABOUT STORAGE STRUCTURES, C, AND ASSEMBLY LANGUAGE

---



Hash tables and linked lists are high-level data structures and are not built in to any standard implementations of C or assembly language. Many C programs include homegrown implementations of linked lists and hash tables (which are fairly easy to write) because they are so useful. We can't teach you the details of every implementation, but we can teach you enough about the basics to recognize them and to know how to use them when you see them.

---

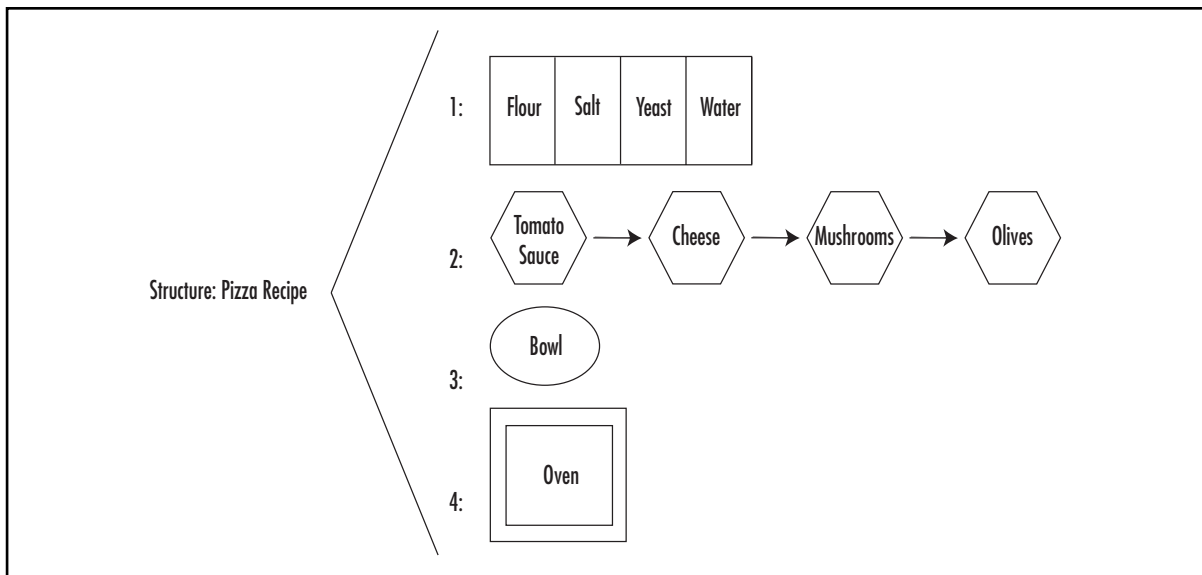


## Structures

Before we go into details about the various storage structures, let's start with the miscellaneous storage structure: the appropriately named *structure* (see Figure B.2). Structures (sometimes called *records*) are conglomerations of different types of data. For example, a pizza recipe structure might hold:

- An array (illustrated in Table B.1) of ingredients to make the crust
- A linked list (as illustrated in Table B.4) of ingredients to make the toppings
- One bowl
- One oven

**Figure B.2** A Pizza Recipe Structure, With Elements We'll Explore in More Detail Shortly



Structures are handy mostly as a logical organization aid. In a large and complex program, you might use a structure to make it easier for you to remember what data should be treated as part of one logical unit.

## Arrays

One way of storing data is in an *array*. An array is like a long row of post office boxes. Each post office box has a unique number on its door and contains mail for one person or family. When a post office customer wants to check her mail, she looks in the box with her number on the door. Her mail is always stored in that box.

In an array, the computer cordons off an area of memory that holds the information being stored, just like the post office wall is filled with post office boxes. Each virtual post office box is called an



## 550 Appendix B • Coding 101

*array element*. Each element stored in the array is indexed by a number. If you want to retrieve the information stored in the fourth chunk of the array, for example, you would request the information telling the computer the array's name and the chunk you want to retrieve. For example, your array might be called *crust* and contain all the different ingredients for pizza crust. An example of array *crust* is shown in Table B.1.

**Table B.1** The Sample Array *crust*

---

### The Array *crust*

---

element 1	flour
element 2	salt
element 3	yeast
element 4	water

---

Now *crust (4)* contains the string *water*. Your pseudocode program might say:

```
print to screen "add " + crust(4)
```

which would produce the output:

```
add water
```

---

### NEED TO KNOW... A NOTE ABOUT NUMBERING

---



In most programming languages, numbering actually starts at 0, not at 1. A list with four elements will have those elements numbered 0, 1, 2, and 3. So the array in Table B.1 will actually look more like the array shown in Table B.2.

---

**Table B.2** Correctly Numbered Array *crust*

---

### Array greetings

---

element 0	flour
element 1	salt
element 2	yeast
element 3	water

---

There are four numbers there, but the first one is numbered 0 and the last one is numbered 3.





## Hash Tables

If you live in a very small town, you might not need post office boxes, because the postmaster knows every resident of the town by sight. Instead of going into the post office, walking up to a large wall full of numbered boxes, and fetching all the mail in the box numbered “303”, you just walk up to the postmaster’s desk and say “Hi, Clark! I’m picking up Chloe’s mail today. Does she have anything?” To which the postmaster replies, “Good morning, Lex! Here’s Chloe’s mail.” Instead of requesting Chloe’s mail by the number of her box, you requested by her name. This is how *hash tables* work. In a hash table, your elements are not indexed by number, as they are in an array, but by a unique name, or *key*. This is illustrated in Table B.3.

**Table B.3** The Sample Hash Table *Greetings*

Key	Output
English	hello
French	bonjour
Russian	zdravstvuite
Spanish	hola

Your pseudocode program might say:

```
print to screen greetings(Spanish) + "world!"
```

Which would produce the output:

```
hola world!
```

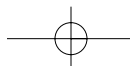
### NEED TO KNOW... HASH TABLES VERSUS ARRAYS



You might ask why we don’t always use hash tables instead of arrays. After all, isn’t it easier to remember that the Spanish word for hello is stored in the box keyed with the word *Spanish* than it is to remember that it’s box number 4? There are two answers to this question. First, in this example, it *is* easier to remember *Spanish* than 4. But more importantly, arrays are nearly always faster than hash tables. This speed of hash table access and array access varies among different language implementations, but it is usually much faster for the computer to find array elements. See the discussion of array implementation in C for an example of why this is usually so.

## Linked Lists

Hash tables and arrays are all well and good if you’re going to be accessing one piece of information at a time, as if you were fetching your mail from a post office box. But maybe you need to get at your stored information in a particular order, first one piece, and then the next. For example, you’ve decided to





## 552 Appendix B • Coding 101

make a pizza from scratch. You need to start with flour, salt, yeast, and water, and then later add tomato sauce, cheese, mushrooms, and garlic. You have to make sure you access your ingredients in order because it won't be a very good pizza if you mix the flour with the mushrooms. A *linked list* makes sure that you access the ingredients in order—a detail that it has in common with arrays. The main difference between an array and a linked list is that in a linked list, you can add or remove containers from your list. Remember, an array is like a wall of post office boxes. If you are storing the ingredients for your pizza in a wall of post office boxes, your pizza recipe might look similar to Table B.4.

**Table B.4** Array of Pizza Ingredients

1	2	3	4	5	6	7	8
Flour	Salt	Yeast	Water	Tomato sauce	Cheese	Mushrooms	Olives

But what if you decide that you don't want mushrooms on your pizza? You can take the mushrooms out of box 7, but there's still an empty post office box between the cheese and the olives. That's both wasteful and confusing and would lead to an arrangement similar to Table B.5.

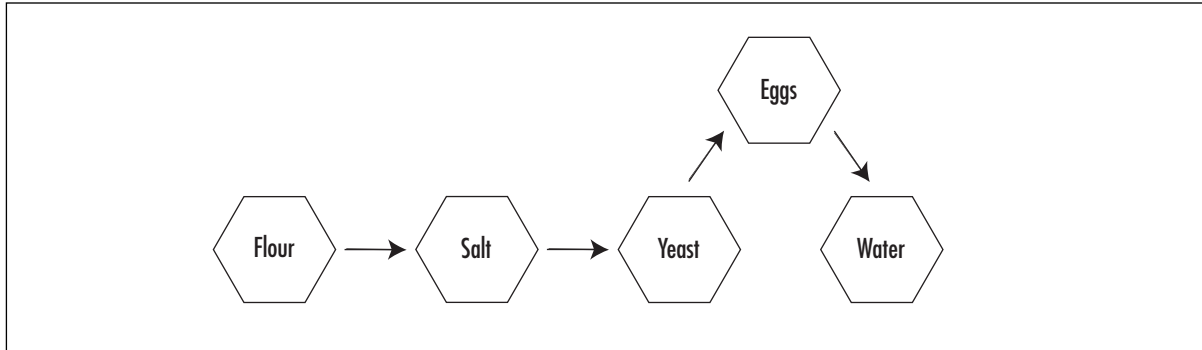
**Table B.5** Modified Array of Pizza Ingredients

1	2	3	4	5	6	7	8
Flour	Salt	Yeast	Water	Tomato sauce	Cheese		Olives

Worse, what if you learn that your pizza crust will be much tastier if you add some egg to the dough? In order to fit in the egg after the water, you'll need to shift the water to box 5, the tomato sauce to box 6, and so on down the line, all just to put the eggs into box 4. Not very practical!

This is why in kitchens we keep our ingredients in ingredient bowls, not in post office boxes. If we have eight ingredient bowls on the kitchen counter, and we decide we don't want mushrooms on the pizza, we can toss out the seventh bowl and move the olives closer to the cheese. If we decide to add eggs, we can squeeze a bowl of eggs between the bowl of yeast and the bowl of water. This arrangement is illustrated in Figure B.3.



**Figure B.3** Linked-List Pizza Crust Ingredients

This is how linked lists work. When we need a new container for information, we can slip one in between two prior containers. When we need to delete an information container, we can do that, too. The disadvantage of linked lists is that we can't directly access a container: "Fetch me the fourth ingredient." We have to say "Fetch me the next ingredient" or (in some implementations) "Fetch me the previous ingredient." Pseudocode for baking a pizza might look like this:

```
while there is a next bowl after this one,  
    fetch me the current ingredient;  
    empty the bowl, and move to the next bowl;  
when there isn't a next bowl after this one,  
    empty the current bowl, and put the pizza in the oven.
```

## Readability

In order to make code maintainable—to fix its bugs and update it as time passes—you should make sure your code is as readable as possible. Programmers joke about code that is *WORN* (Write Once, Read Never). It's easy to write a computer program that is completely unreadable. But unless you're trying to win the annual International Obfuscated C Contest (a genuine contest, run since 1984, which gives prizes to the most unreadable and bizarre C program entered—archives available at [www.ioccc.org/years-spoiler.html](http://www.ioccc.org/years-spoiler.html)), you probably want to make sure that you can read your own code after you've written it.

## Comments

All computer languages give you the ability to write *comments* in the code. Comments are blocks of the program text that the computer ignores. Comments are intended for you, the programmer, and anyone else who might need to read the code. Since computer languages rarely look much like English, it can be difficult to look at a piece of code you've written after some time has passed and understand exactly what it does. If you include comments explaining the intent of each significant block of code, you'll always be able to understand the original intent of those lines. Each language has

**554 Appendix B • Coding 101**

a different way of telling the computer that some lines of text are comments and not program code, such as the symbols #, /\*, or //, but it is usually pretty easy to recognize them. An example of commenting is shown in Figure B.4.

**Figure B.4 Pseudocode With Comments**

---

```
while nextBowl exists      /* if the next bowl isn't empty */
    fetch Ingredient       /* take the ingredient from the current bowl */
    nextBowl              /* move to the next bowl */
    delete prevBowl      /* put the previous, empty bowl in the sink */

when nextBowl doesn't exist /* when you're done, */
    delete Bowl          /* put the last bowl in the sink */
    bake pizza           /* and put the pizza in the oven! */
```

---

## Function and Variable Names

When you're writing a computer program, you'll probably have the opportunity to assign lots of arbitrary names to variables and functions. It's easy to get lazy and assign function and variable names that are very short, so you don't have to type very much. But take a look at the program from Figure B.4 if we replace all the variable names with something very short and easy to type, as shown in Figure B.5.

**Figure B.5 Pseudocode With Confusing Variable Names**

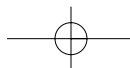
---

```
while i exists            /* if the next bowl isn't empty */
    fetch k               /* take the ingredient from the current bowl */
    i                    /* move to the next bowl */
    delete m             /* put the previous, empty bowl in the sink */

when i doesn't exist     /* when you're done, */
    delete j             /* put the last bowl in the sink */
    bake n               /* and put the pizza in the oven! */
```

---

This piece of pseudocode means the same thing as far as the computer is concerned, but it doesn't really make any sense to you or me. Whenever possible, use variable and function names that have meaning to you in the context of your program. Doing so might involve a little bit more typing now, but it will make your life much, much easier later, when you have to fix a bug in your code.



## White Space

In most modern programming languages, an excess of white space is ignored by the computer. This means that you can use as many—or as few—tabs and space characters as you need. Your program will be easier to read later if you format it so that it is clear how the program flows. When it comes to the nitty-gritty details of formatting, there are as many preferences as there are programmers. However, a couple of broad conventions have been agreed on as generally useful:

- Use a new line to indicate a new command. Figure B.4 could have been written in just two lines, but it would have been much harder to read:

```
while nextBowl exists; fetch Ingredient; nextBowl; delete prevBowl.  
when nextBowl doesn't exist; delete Bowl; bake pizza.
```

- If a block of text is part of a loop, function, or conditional structure, use leading white space to show the lines of code that are being evaluated similarly. Here is the psuedocode from Figure B.4 without leading white space for the loop and conditional statements. This is much harder to read than the sample with white space:

```
while nextBowl exists  
  fetch Ingredient  
  nextBowl  
  delete prevBowl  
when nextBowl doesn't exist  
  delete Bowl  
  bake pizza
```

## Introduction to C

C is a runtime environment that exists on nearly every computer platform. C is a platform-independent compiled language, but it has a large library of hardware-specific, low-level system calls available to help us access the hardware that we are programming for. On its own, C is a very small language; it doesn't even know how to display text to the screen! But every C installation comes with the *C standard libraries*, which provide the programmer with a host of handy functions.

C is a *compiled* language. This means that we write the program in the English-like language that you're learning here and then use another program (a *compiler*) to convert it into commands the computer can understand and execute.



## NEED TO KNOW... YOUR COMPILER

---



Many different C compilers are available. You might be using a command-line compiler, for which you write your program in a text editor such as Notepad, Emacs, or vi and then compile your program with a command such as `cc myprogram.c -o myprogram.exe`. You might be using a graphical programming environment, where you write and compile your program in an easy-to-understand window, such as Visual C or CodeWarrior. We can't teach you the ins and outs of the compiler you'll be using, because they're all different. Refer to your compiler manual for instructions on how to compile your C program.

---

## History and Basics of C

C was invented by Dennis Ritchie (based on work done by Kenneth Thompson) in the early 1970s as a language intended for programming on Thompson's brand-new UNIX operating system. C was standardized into what we now know as *ANSI C* in the mid-1980s. For many years, C was primarily used for programming on UNIX and its variants, but it is now a widespread standard. C and its descendents (including C++ and C#) are among the most commonly used programming languages.

## Printing to the Screen

A C program is just a sequence of commands. Let's start with our first program, the ubiquitous "hello, world," which is the first program you will learn to write in almost any programming language (see Figure B.6).

**Figure B.6** The Hello, World Program

---

```
1    #include <stdio.h>
2
3    main()
4    {
5        printf("hello, world\n");
6    }
```

---

Let's break down this program. The meat of any C program, the part that runs when you execute the program, is the *main* block. This main block is a special-purpose *function* that tells your program to begin its work here. You can see the declaration of the main block on line 3 of Figure B.6. Those parentheses after the word *main* are required after any function and are used to pass arguments to the function if you need any (we'll get to some details of function calls and argument passing later). In this program, we aren't passing any arguments to the main function, so the parentheses are present but empty.





After a function's initial line, all statements that belong to that function are grouped together with curly braces `{}`. In this program, those curly braces are on lines 4 and 6. Anything between those curly braces (in this case, line 5) is part of the function. So, in this program, the heart of the main block is the command on line 5:

```
printf("hello, world\n");
```

The command *printf*, like *main*, is a function. Notice that it begins with the function name (*printf*) followed by zero or more arguments in closing parentheses (in this case, one argument, which is equal to the string `"hello, world\n"`). *printf* is the formatted print command. Here it is printing to the screen the contents of its argument: the characters *hello world* followed by *\n*. In a C character string, a single character preceded by the backslash character (`\`) has a special meaning. *\n* is the C notation for printing a new line to the screen. You can't put a new line directly in a quoted string, for example:

```
"here is my first line
here is my second"
```

This is not valid C. To write those lines to the screen, your command would have to be:

```
printf("here is my first line\nhere is my second\n")
```

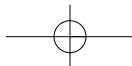
or a variant:

```
printf("here is my first line\n");
printf("here is my second\n");
```

or:

```
printf("here is ");
printf("my first line\nhere is my second\n");
```

The separate *printf* commands don't change where a new line begins. Only the *\n* characters create new lines.





## NEED TO KNOW... SOME INTERESTING CHARACTER STRING ESCAPE SEQUENCES



Several similar sequences cause *printf* to display something special to the screen. Some are very special types of characters, such as the audible alert bell that *printf* sounds when given the character sequence `\a`. Most are designed simply to escape the meaning of some other character (hence the name *escape sequences*), to allow *printf* to print the literal character instead of trying to interpret the meaning. For example, if we need to display a double quote mark (") on the screen, we need to prevent *printf* from parsing the special meaning of the double quote mark as "here is the beginning or end of a character string." Some of the more interesting escape sequences include:

- `\n` newline
- `\t` horizontal tab
- `\?` question mark
- `\'` single quote
- `\"` double quote
- `\a` alert bell

Earlier we mentioned that C doesn't really have much complex functionality of its own and doesn't even know how to output characters to the screen in any simple way. Well, that's where line 1 of Figure B.6 comes in. C has standard libraries that provide that basic functionality that is not built into the language. To make your final program as small as possible, you include only the standard libraries you need into your program. Line 1 includes the standard library *stdio.h*, which is responsible for standard input and output functionality. The included library provides us with the *printf* function.

One last character we haven't covered: that semicolon (;) at the end of line 5. C commands are separated by semicolons, not by white space, so the following commands are legal:

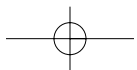
```
printf("here is ");
printf("my first line\nhere is my second\n");
```

But this next example isn't:

```
printf("here is ")
printf("my first line\nhere is my second\n")
```

## Data Types in C

C is a *strongly typed* language. This means that the language distinguishes among the different types of data it can process. It's important to recognize data types for many reasons. For one thing, your programming language needs to allocate storage for any information you intend to store. To store the integer 8 is relatively simple; you need as much space as the computer will take to store that integer. But what if you want to store the real number 8? (To program efficiently, you're going to use things you learned in math class! Remember that integers are only the numbers  $-\infty, \dots, -3, -2, -1, 0, 1, 2, 3, \dots, \infty$ , but that real numbers also include numbers like 3.759.) If you want to store the real number 8 to, say, three points of precision (that is, so you can distinguish between 8.000 and 8.003), you'll need a lot



more storage space in the computer. And if you want to be able to distinguish between 8.000 and -8.000, you'll need even more space. So it's important to use the right data type for your variable, or you can rapidly run out of memory for your program.

C has only a few data types:

- **int** An integer.
- **float** A single precision floating-point number (basically, a real number).
- **double** A double precision floating-point number (basically, a real number with extra precision).
- **char** One character.

These data types can optionally be used with the following modifiers:

- **short** If you aren't using very large numbers and want the program to allocate space effectively.
- **long** If you are using very large numbers.
- **signed** If it matters to you whether the numbers are positive or negative.
- **unsigned** If you're not going to be using negative numbers and you want the program to allocate space effectively.

For entry-level programming, you won't use any of these modifiers, but you'll want to be able to recognize them if you see them in somebody else's code.

If you've done some programming before, you might notice two types that are missing here: Booleans and character strings. Booleans (the values *true* and *false*) are usually represented in C as a special case of integers. Character strings are arrays of characters. We'll talk more about how to implement character strings later.

## Mathematical Functions

You know what's really great about computers? They know how to do arithmetic, so we don't have to. Many basic mathematical functions are included in C, and to use them you don't need to include any standard libraries. An additional library called *math.h* provides more complex mathematical functions such as sines, cosines, logarithms, and powers. Figure B.7 displays a program that calculates the number of minutes in a day.



## 560 Appendix B • Coding 101

**Figure B.7** Mathematical Example

---

```
1  #include <stdio.h>
2
3  main()
4  {
5      /* variable declarations */
6      int seconds, minutes, hours;
7      int total;
8      seconds = 60;          /* number of seconds in one minute */
9      minutes = 60;         /* number of minutes in one hour */
10     hours = 24;           /* number of hours in one day */
11
12     total = seconds * minutes * hours; /* calculate total */
13     printf("there are %d seconds in one day.\n", total);
14 }
```

---

When you run this program, your computer should print out the line:

**there are 86400 seconds in one day.**

Let's step through this program to see what we did. You recognize line 1—it includes the standard input and output library. We're using this library to get the *printf* command, which will print the results of our mathematical equation. Line 3 begins the main function, and line 4 provides the curly brace that tells the program “the lines between here and the matching curly brace belong in the main function.”

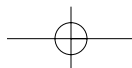
The first new line we've seen in this program is on line 5: */\* variable declarations \*/*. This is a C comment: a line of the program that is there for your benefit only but is ignored by the compiler. Anything between the initial */\** and the closing *\*/* is a *comment* and not part of the program. The comment on line 5 lets us know that we are about to *declare variables*.

Variables in C are declared before use. A declaration, which consists of a data type and some number of variable names, tells the program the sort of information that is going to be stored in that variable. In Figure B.7, the variables are defined in two lines (6 and 7):

```
int seconds, minutes, hours;
int total;
```

Because they're all of the same type (*int*—that is, integers), we could have declared them all in one line:

```
int seconds, minutes, hours, total;
```



or on four separate lines as follows:

```
int seconds;  
int minutes;  
int hours;  
int total;
```

C doesn't care how you lay it out, so you should use whichever method makes your code most readable for you. You might split conceptually—variables that all refer to one function on one line and to another function on a second line—or by any other method you like.

After you've declared your variables, you can assign them. *Assignment* gives a value of the appropriate type to the variable you have even a declaration. In this case, the appropriate type is integer, so we assign each variable name its initial value as follows (lines 8, 9, and 10):

```
seconds = 60;  
minutes = 60;  
hours = 24;
```

This way before we begin the computation, the variables contain meaningful values.

On line 12, the actual calculation occurs:

```
total = seconds * minutes * hours;
```

Most of these characters should be fairly familiar:

- Equals sign (=) is the *assignment operator*, which places the results of the calculation to the right of the equals sign into the variable on the left.
- The asterisk (\*) says to multiply, just like you would use × in a written calculation: seconds × minutes × hours.
- To add and subtract you would, predictably, use the plus sign (+) and the minus sign (-), and to divide, you would use the slash (/).

The statement on line 13 is a *printf* statement, but this one looks a little different. For one thing, it has two arguments separated by a comma: a quoted character string and a variable name.

```
printf("there are %d seconds in one day.\n", total);
```

The *printf* function does more than just output simple character strings to the screen. It can do complex output formatting. The first argument to the *printf* function is always a character string. That character string can contain some number of substitution characters, each one a letter prefaced by %. For each substitution character, the *printf* function takes an argument explaining which variable will have its contents substituted into the character string.



## 562 Appendix B • Coding 101

In this example, the substitution character is `%d`. This is C for “take the value of the variable for the corresponding argument and display it as a decimal integer.” The argument that corresponds to `%d` is `total`. In the preceding calculation, the value of `total` was set to  $60 * 60 * 24$ , or 86,400. Thus, the function’s output will be:

```
there are 86400 seconds in one day.
```

### NOTE



This non-intuitive removal of the variable from the printing string isn’t present in some higher-level languages. In Java, for example, the preceding statement would be:

```
System.out.println("there are " + total + " seconds in one day.");
```

You might ask why we set the number of seconds in a minute, the number of minutes in an hour, and the number of hours in a day as variable values. After all, aren’t variables supposed to be, well, variable? But there are always 60 seconds in one minute, always 60 minutes in one hour, and always 24 hours in one day. And in fact, there is a way to create a *symbolic constant* to hold this kind of information that will never change. Instead of declaring and assigning the following variable:

```
int seconds;
seconds = 60;
```

you can define a symbolic constant:

```
#define SECONDS 60
```

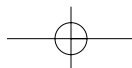
At compilation time, every occurrence of `SECONDS` will be replaced with your replacement text, `60`. Note that there is no semicolon completing a `#define` line. By convention, symbolic constants are written in all capital letters to distinguish them from variable names, which are conventionally some combination of upper- and lowercase letters.

## Control Structures

Remember all those control structures we learned about the beginning of the appendix? Well, C can do all of those. We’ll look at two forms of looping (*for* loops and *while* loops) and two forms of conditional branching (*if/then/else* statements and *switch* statements). Unconditional branching in C is implemented with function calls, which we’ll deal with in the next section.

### *For* Loops

The *for* statement is a loop that operates until a certain condition has been met. This concept is shown in Figure B.8.





### Figure B.8 A Sample *for* Loop

---

```
int i;

for ( i = 1; i <= 10; i++ )
{
    ...
}
```

---

There are three components to the loop's control mechanism, all stored within the parentheses. Look at the three parts of the statement in Figure B.8, separated by semicolons. First:

```
i = 1
```

This part of the loop initializes any variables that will be used during the loop's control. Here we are taking a variable *i* (which has been declared beforehand as an integer—*int i*;) and initializing it to 1. The second part of the *for* loop's control gives a test condition:

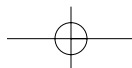
```
i <= 10
```

This test is evaluated during program operation. In this case it is asking whether or not the variable stored in *i* is less than or equal to 10. If it isn't, the program will exit this *for* loop and continue on with whatever it was doing before the loop was entered. If it is, the body of the loop will be executed. Before we re-enter the loop and perform all this once more, we do the third step of the *for* loop:

```
i++
```

This step increments the counter variable we are using in the loop. This command tells C to add 1 to the variable stored in *i*.

The first time this program runs, the variable *i* will be initialized to 1, the program will test to see if 1 is less than or equal to 10, and it will discover that it is. The body of the loop will be executed, the variable *i* will be incremented by the statement *i++* to 2, and the process will begin again. After the tenth time this program runs, the variable *i* will be incremented to 11, and the loop will stop as *i* no longer meets the test condition.





## 564 Appendix B • Coding 101

## Comparison Operators and Increment/Decrement Operators

In this section you were introduced to two new operators:  $\leq$ , a comparison operator, and  $++$ , an increment operator.

*Comparison operators* test some relation between the value on the left and the value on the right:

- $<$  Is less than.
- $\leq$  Is less than or equal to.
- $>$  Is greater than.
- $\geq$  Is greater than or equal to.
- $==$  Is equal to.
- $!=$  Is not equal to.

### NOTE



To test if two values are equal, the comparison operator has two equals signs ( $==$ ). To assign a value to a variable, the assignment operator has one equals sign ( $=$ ). Don't get them confused! If you accidentally write a comparison statement like  $i = 10$ , your statement won't test to see if the variable  $i$  is equivalent to 10; it will assign the value 10 to your variable.

*Increment and decrement operators* provide shorthand for adding or subtracting one to a variable:

- $i++$ ,  $++i$ , and  $i = i + 1$  all add 1 to the value of  $i$ .
- $i--$ ,  $--i$ , and  $i = i - 1$  all subtract 1 from the value of  $i$ .

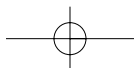
### WARNING



Actually, the three forms do have subtly different meanings having to do with timing and precedence. These distinctions shouldn't matter at this level of programming, but be aware that they exist as you move on to more advanced programming tasks.

## While Loops

A *while* loop is very similar to a *for* loop, but rather than having the variable initialization and incrementation controlled by the loop itself, they happen elsewhere. We *initialize* the variable before we ever enter the *while* loop, *test* the variable value in the loop control, and take care of any *variable modification* inside the body of the loop. The *for* loop in Figure B.8 can be implemented with a *while* loop as shown in Figure B.9.





### Figure B.9 A Sample *while* Loop

---

```
int i;
i = 1;

while ( i <= 10 )
{
    ...
    i++;
}
```

---

The counter variable is set before we begin the loop. When we enter the loop, we perform the test: Is the variable less than or equal to 10? If it is, we enter the loop, perform some code in the block, and finish incrementing the counter variable as part of the block.

### If/Else

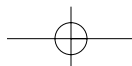
An *if* statement performs conditional branching. In an *if* statement, we test to see if the condition is true, do something if it is, and possibly do something else if it isn't. We've done tests as part of the *while* loops and *for* loops, but there is no looping built into *if* statements. An *if* statement might look similar to that shown in Figure B.10.

### Figure B.10 A Sample *if/else* Statement

---

```
int i;
...
if ( i == 1 )
{
    [A: some lines of code here]
}
else if ( i == 2 )
    [B: only one statement can go here]
else
{
    [C: some lines of code here]
}
/* end if statement */
```

---





## 566 Appendix B • Coding 101

First, our *if* statement tests to see if the variable *i* is equivalent to 1. If it is, it executes the lines of code enclosed in the braces and terminates the statement (that is, no code in the *else* clauses of this statement will be executed). If it isn't, it looks to see if there is an *else* clause, and there is. The first *else* condition says to run another test: Is the variable equivalent to 2? If it is, we enter that block of code (notice that there are no braces around that next section of code; this is permissible as long as there is only one semicolon-terminated statement in the block) and don't execute any other *else* clauses in this *if* statement. If the variable isn't equivalent to 2, we move on to the final clause. Because there is no *if* after this final *else*, all other cases execute this block of code:

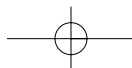
- If we enter the *if* statement when the variable *i* is equal to 1, we will execute only the line of code labeled *A*.
- If we enter the *if* statement when the variable *i* is equal to 2, we will execute only the line of code labeled *B*.
- If we enter the *if* statement when the variable *i* is equal to any number other than 1 or 2, we will execute only the line of code labeled *C*.

## Switch

A *switch* statement is like a special case of a multi-tiered *if/else* statement. In each test of an *if* statement, you can test for something different. For example, you could write a program similar to Figure B.11.

**Figure B.11** A Complex *if/else* Statement

```
if ( foo == 1 )
{
    ...
}
else if ( bar <= 39 )
{
    ...
}
else if ( baz == 's' )
{
    ...
}
```



But often your tests are much simpler than this and you just want to test for assorted values of a single variable (which is, in fact, what we did in Figure B.10 to learn *if* statements). *Switch* statements deal with this special case of testing simply to see if one expression matches one of a number of values (see Figure B.12).

**Figure B.12** A Sample *switch* Statement

```
switch (foo)
{
    case 1:
        [A: some lines of code]
    case 2: case 5:
        [B: some lines of code]
        break;
    default:
        [C: some lines of code]
        break;
}
```

This code is running a test on the variable named *foo*. If the variable *foo* is equivalent to 2 or to 5 (the line *case 2: case 5:*), it will execute the lines of code we've marked *B* and then break out of the *switch* statement. If the variable *foo* is equivalent to 1 (the line *case 1:*), it will execute the lines of code we've marked *A*, but because there is no *break;* statement, it will also execute the lines of code labeled *B*. Be careful of this; remember to use *break!* If the variable *foo* is equivalent to any number other than 1, 2, or 5, it will execute the code labeled *default*, the code we've marked *C*.

#### NOTE



The lines of code after a "case" in a *switch* statement do not need curly braces around them. The *switch* statement itself does need curly braces.

## Storage Structures

### Arrays, Pointers, and Character Strings

A *pointer* is a special kind of variable. Its job is to contain the address of another variable. The address is the location in the computer's memory where the second variable lives. Your house address is a pointer to where on your street, in your town, you live. Knowing your address, we can come find you. Similarly, the variable's address tells the computer program how to find that variable in memory.



## 568 Appendix B • Coding 101

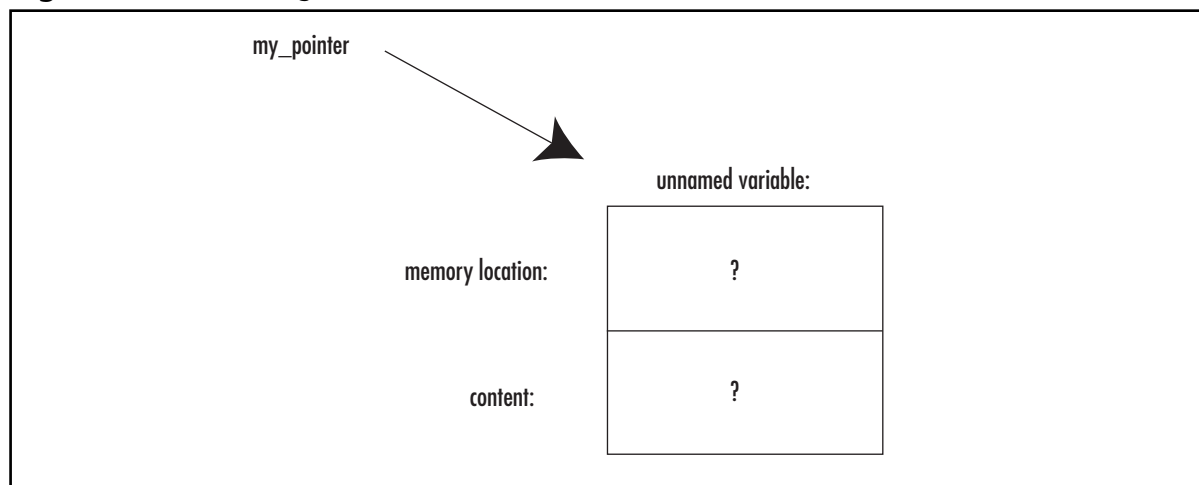
Pointers, and their cousins *address operators* and *arrays*, can be extremely powerful, but they can also be very confusing. Understanding pointers and dereferencing are the biggest hurdle in learning C. They won't make sense all at once; don't worry, it will sink in over time! Once you master this concept, you'll be well on your way to becoming a great programmer.

To begin, let's imagine that we have an integer variable called *my\_variable*. If we want a pointer to it, we can declare one using the ampersand (&) operator to find the address of *my\_variable*. We begin by declaring the two variables, one integer and one pointer to integer:

```
int my_variable;
int *my_pointer;
```

The asterisk (\*) in front of *my\_pointer* defines *my\_pointer* as a pointer to some other value. In this case, since the declaration begins *int \**, we know it's a pointer to a value of type *integer*. The pointer refers to some location in memory, with no value yet assigned (see Figure B.13).

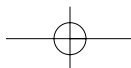
**Figure B.13** Declaring a Pointer

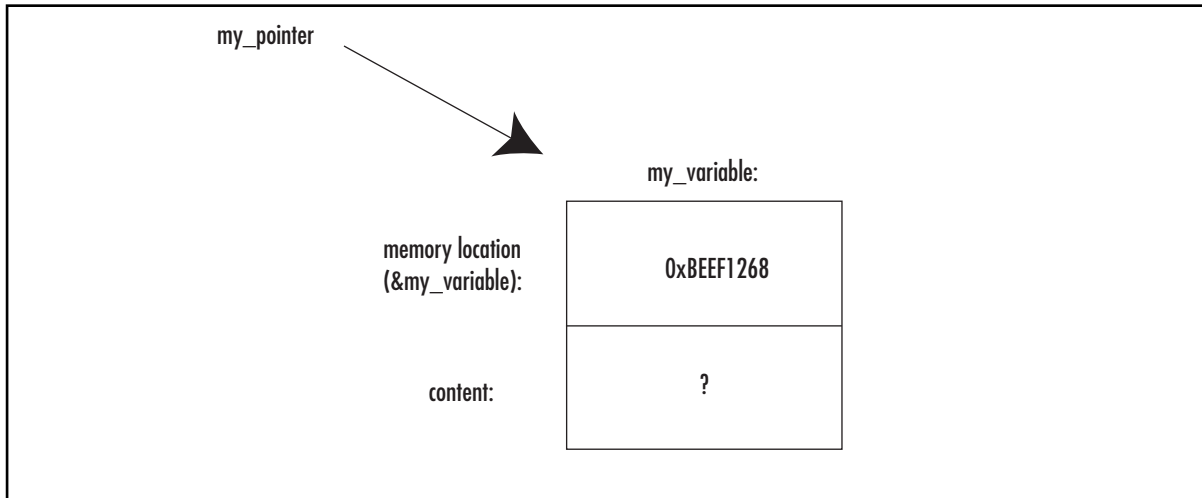


Now we follow the declaration with an assignment:

```
my_pointer = &my_variable;
```

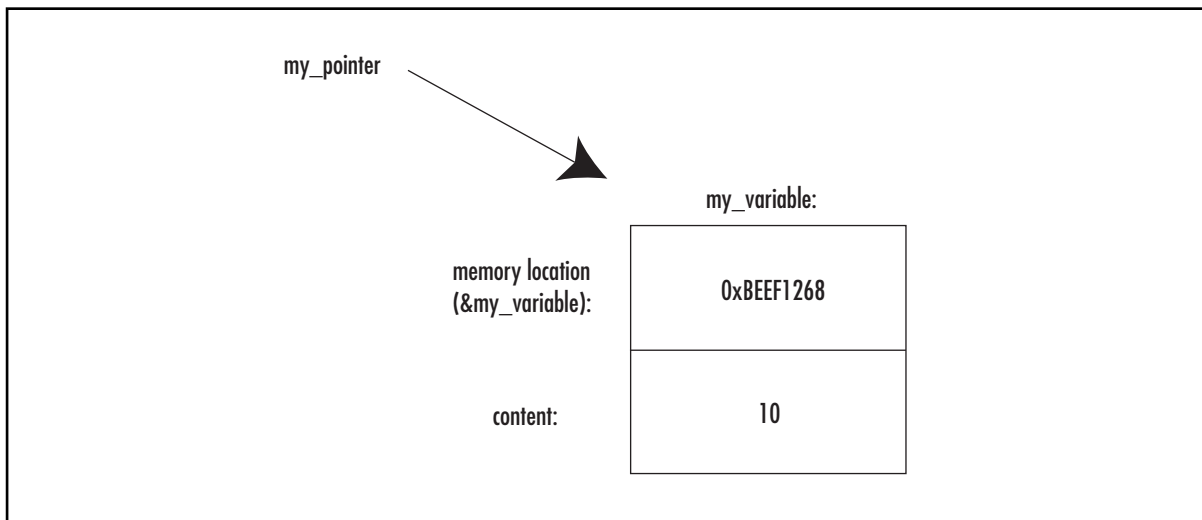
Figure B.14 illustrates this assignment. Though the integer named *my\_variable* has no value yet, it does have an assigned memory location that's large enough to hold an integer value. The variable *my\_pointer* points to *my\_variable*. The ampersand (&) character in front of *my\_variable* sends out the address of the memory location that has been set aside to store the variable's contents, in this example, 0xBEEF1268. The assignment of this value to *my\_pointer* means that *my\_pointer* always knows the memory location of the variable held in that location. That is, *my\_pointer* points to *my\_variable*.



**Figure B.14** Assigning a Pointer to an Address of Another Variable

The integer stored in `my_variable` (and pointed to by `my_pointer`) can then be assigned into the location pointed at by `my_pointer`. Figure B.15 illustrates the pointer once the value has been assigned with the command:

```
*my_pointer = 10;
```

**Figure B.15** Pointer Assignment

Note that when we are giving the address of the integer to `my_pointer` (`my_pointer = &my_variable`), we don't need to use the asterisk (`*`); we are assigning an address (obtained through the amper-



## 570 Appendix B • Coding 101

sand [&] operator) to a pointer variable, which takes an address without transformation. But when we give an actual integer value to *my\_pointer* in with the command `*my_pointer = 10`, we don't want to change the address of the thing that is being pointed to, but the thing itself, its content—so we need to use the asterisk (\*) to show that extra level of indirection. The \* character is called the *dereferencing* operator because it tells our code not to look at the address reference but at the object it points to, or references.

Arrays in C can be thought of as a special case of pointers. When you declare an array that contains 10 units in C, you're creating your array (like a row of post office boxes) in 10 consecutive chunks of memory.

As you can see in Table B.6, the elements are referenced by the array name (*my\_array*) and their locations in the array: *my\_array[0]*, *my\_array[1]*, ..., *my\_array[9]*.

**Table B.6** An Array Split up into Individual Pointers

Array <i>my_array</i>				
<i>my_array[0]</i>	<i>my_array[1]</i>	<i>my_array[2]</i>	<i>my_array[3]</i>	<i>my_array[4]</i>
<i>my_array[5]</i>	<i>my_array[6]</i>	<i>my_array[7]</i>	<i>my_array[8]</i>	<i>my_array[9]</i>

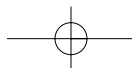
This is the most straightforward way to deal with arrays. But when you're looking at code other people have written, you might notice that they have declared arrays with pointers. See, there's a tricky little side effect in the direct memory addressing that pointers give you. We set up a pointer to the new array:

```
my_array_pointer = &my_array[0];
```

(which, in C shorthand, can also be written as `my_array_pointer = my_array`;—the name of the array is synonymous with the address of the first element). *my\_array\_pointer* is an address in memory. That address in memory points to the first element in 10 consecutive chunks of memory that comprise my array. So in a nifty and incredibly confusing operation called *pointer arithmetic*, you can access the second element of the array by saying:

```
*(my_array_pointer+1)
```

Confusing? As though that weren't bad enough, here's a new wrinkle: Declaring a pointer doesn't actually allocate enough memory for the entire array. If you decide to declare your arrays by using pointers instead of array notation, you'll have to learn how to allocate memory using the library function *malloc*. For entry-level C programming, we recommend sticking to array notation. It's bulkier but much less error prone.



## Strings

Strings are any quoted series of characters such as:

```
"Hello, World!"
```

Character strings are a special case of array. Specifically, a string is an array of characters terminated with a null character, which is notated `\0`.

The string in Table B.7, *my\_string*, is an array with 14 elements. The 14<sup>th</sup> element, which we access using *my\_string[13]*, is the null character. It's an important part of the string we must not forget, even though we never see it!

**Table B.7** Character String as an Array

String <i>my_string</i>													
0	1	2	3	4	5	6	7	8	9	10	11	12	13
H	e	l	l	o	,	space	W	o	r	l	d	!	\0

As with any other array, a character string can be declared either with a pointer or with array notation.

```
/* declares a pointer to unallocated space of a string of unknown length */
char *pointer_to_string;

/* creates an array for a string of 10 characters and allocates the space */
char array_of_string[10];
```

The correct amount of space is allocated if you assign a value to the string at the same time that you declare it:

```
char *pointer_to_string = "Hello, World!";
char array_of_string[] = "Hello, World!";
```

Both of these declarations allocate enough space for 14 character strings and populate the strings with the assigned value. As with any other array, we strongly recommend using array notation rather than pointer notation to deal with strings. It is very bulky to guess ahead of time how many characters you would like to allocate, and when you become more comfortable with pointer notation you will probably switch to that style because it is more space efficient. But for now, you will find your code much easier to debug if you stick with array notation.



## NEED TO KNOW... WARNING ABOUT UNALLOCATED MEMORY



C will not stop you from accessing data you have not allocated. For example, if you allocate an array large enough to hold five integers,

```
int my_array[5];
```

There is nothing in the language preventing you from later trying to grab the 23rd element of the array.

```
int my_number;
my_number = my_array[23];
```

But since you haven't reserved that memory for the array, you have no control over what information might be present in it. It might be empty, or it might be filled with garbage. Or, if that memory doesn't belong to your program, it might crash.

Attempting to read or write unallocated memory might well be the number-one cause of debugging frustration and cursing at computers for a C programmer. A program that looks perfectly valid will suddenly crash, presenting a message similar to:

```
Segmentation violation, core dumped.
```

This happens so frequently that way back in 1980, Greg Boyd at Digital Equipment Corporation wrote a song about it: "The segmentation violation core dumped blues" (see the lyrics at [www.netSPACE.org/~dmacks/internet-songbook/core-dump-blues.html](http://www.netSPACE.org/~dmacks/internet-songbook/core-dump-blues.html)). Nearly 25 years later, it still happens. Just make sure that you allocate unassigned memory before you read it!

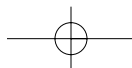
## Structures

After all the complexity of arrays and pointers, structures are mercifully simple. A structure, or *struct*, contains some number of other data types, all conveniently grouped together. In fact, a *struct* can contain other *structs*:

```
/* struct to hold some important info about a tv show */
struct tv_show {
    int channel;
    char show_name[50];
    char favorite_actor[50];
};

struct show_to_record {
    struct tv_show show_I_like;
    long time;
};
```

Now we can declare and assign a variable of type *show\_to\_record*:





```
struct show_to_record IronChef;
struct show_to_record Friends;

IronChef.time = 4;
IronChef.show_I_like.channel = 102;
IronChef.show_I_like.show_name = "Iron Chef";
IronChef.show_I_like.favorite_actor = "Fukui-San";
```

See how you access the elements of the *struct*? If you call the *struct* by its declared name and follow with a dot (.) and the name of the member, you can assign a value to that member. When a *struct* contains a *struct*, you can add another dot, followed by the name of that *struct*'s member, and so on. You have to allocate space for all members of a *struct*. If you're using pointers or arrays of unspecified size, you'll need to explicitly allocate the space for them.

## Function Calls and Variable Passing

A *function* is a piece of code that is separately defined and can be run as many times as you like. It is essentially a subroutine. The C function *printf* we've been using is an example of a system library-provided function.

Once you've written a function, you need some way to pass your variable to it. There are two ways of dealing with variable data in C: *call by reference* and *call by value*. A variable that has been called by value has a *copy* of its data passed to the function, not the data itself. If you make changes to the variable in the function, you have *not* made those changes to the variable in the main program.

So how do we use a function to make changes that persist in the main program? The first method is simple if you are only changing one variable. A function returns a value (just like any other variable, it can return a value of standard data type such as *int*, *char*, or the like). If you're only modifying, say, one string, you can have a function that returns a value of type *string* (that is, a pointer to character, or *char \**), as shown in Figure B.16.

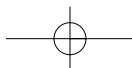
**Figure B.16** A Sample Function Declaration

---

```
/*
 * this function returns a value of type pointer to char,
 * or "char *"
 */
char *my_function();
```

---

This solution isn't without drawbacks. For one thing, you might want to modify several variables in one function. For another thing, there is a convention that many functions follow which return integers containing their success status (0 if the function succeeded or 1 if there was an error). If you want that functionality, you can't return both a status integer and a modified variable, but only one.





## 574 Appendix B • Coding 101

There's an interesting little side effect to C's use of pointers that gives an excellent workaround. Let's say the function in Figure B.16 is passed a character string:

```
char *my_function(char *);
```

This format means that *my\_function* is a function that takes one variable, a pointer to *char* (presumably a character string), and that returns one variable, also a pointer to *char*. Let's call this function:

```
/* since "char *" is another way of referring to "char[]", */  
/* either syntax can be used for declaration */  
char some_string[] = "here is my string";  
  
/* now pass it to the function */  
my_function(some_string);
```

We've learned the C uses call by value passing in function calls. So what is being passed to *my\_function* here? Is it a copy of the entire character array passed—"here is my string"? No, in fact it's a copy of the *pointer* that points to the character string *some\_string*. It's not the same pointer, but they both point to exactly the same place. So if you modify *some\_string*, you actually are modifying the string itself. This is how C approximates *call by reference*. The function is being passed a value but that value is a copy of a reference. In this way, you can modify any information that lives outside a function from inside a function. All you need to do is pass a pointer to the variable.

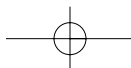
## System Calls and Hardware Access

Sometimes your program needs to interact directly with the operating system or hardware. For this we have *system calls*. These will be different on every operating system you use, because they depend on the abilities of the individual operating system. You will probably need to include a system call-specific library file at the beginning of your program; check the documentation for your particular operating system and hardware platform.

System calls are usually necessary for the kinds of low-level hardware access you need if you're writing a device driver. For example, you probably have access to the calls *read* and *write*, which read and write bytes directly from some file descriptor. To properly use these functions, you need some information about the hardware. For example, you probably need to know the physical device's *block size*—a bit of information about the physical device's logical storage mechanism.

You may also have access to some basic file systems calls: *open*, *creat* (yes, that's spelled *creat*, with no *e* on the end), *close*, and *unlink*. These allow you to manipulate files at a level very close to the operating system, instead of in the higher-level functions that are part of the standard library `<stdio.h>`.

You'll need to know a little bit about the structure of your file system and the devices you mean to access if you'll be using system calls. Since system calls are the only way to get close access to the hardware in C, you'll almost certainly need them if you'll be writing any programs that access hardware components directly, such as a device driver to control a sound card.



## Summary

C is a powerful language, and we have only introduced a small amount of what it can do. Most of what we've introduced here we've only touched on lightly, and there are many C features we haven't had time to discuss, including such important topics as:

- Enums
- Pointer arithmetic
- Bitwise operators ( <<, >>, &, ^, | )
- Logical operators ( &&, ||, ! )
- Order of precedence
- The standard libraries (primarily string and file functions)
- Variable scope
- Void types
- Explicit type casting

If you plan to write a lot of C, we strongly recommend the books in the "Additional Reading" section of this appendix.

## Debugging

Chances are, it won't take you long after you've written your first program to discover your first bug. Everybody, from curious hackers to professionals with decades of experience, makes programming errors. Although it might be easy to find the bug in a five-line program, it can be a lot harder as your programs get more complex. So how do you track down your bugs?

## Debugging Tools

Many integrated development environments come with built-in graphical debuggers. These tools allow you to track exactly what your program is doing at any given point in time. Do you think your program is having problems entering your function *make\_everything\_work()*? Then drag the *stop here* icon, which might look like a little stop sign or an exclamation mark (check your program's documentation), to the line of the program right before it enters that function. When you run the program in your graphical debugger, it will run to that point and then stop and wait for you. You can tell the debugger to step through one line at a time, reporting the contents of variables to you as it goes. This can be a very easy way to discover the reason why your program is crashing.

If you don't have a built-in debugger, or if you prefer the command line, there are tools that you can use. The GNU debugger, or *gdb* ([www.gnu.org/directory/gdb.html](http://www.gnu.org/directory/gdb.html)), is open source and freely available on a very large number of hardware and software platforms. The GNU project also supplies a

**576 Appendix B • Coding 101**

graphical front-end to gdb and other command-line debuggers, called DDD ([www.gnu.org/software/ddd](http://www.gnu.org/software/ddd)). Some programmers find command-line tools far more powerful, because they can quickly type any command they need rather than looking in a menu; others find it frustrating not to have the visual aid of a graphical tool while doing complex debugging.

## The *printf* Method

Sometimes you have a very short program, and you're pretty sure you know where the bug is. Starting up a debugging program is cumbersome for you, and you don't really want to bother or you might not have a debugger available—all you need to know is the value of a variable before, during, and after you enter your function. This is where homemade debugging comes in.

Just tell your program to print the values of the questionable variable at various points during your program's run. This doesn't work particularly well if you're programming graphics, but for straight text output, it's reasonably effective.

For example:

```
int foo;

printf("before I enter the function, foo is %d\n", foo);
/* Enter the function my_function_works */
my_function_works();
printf("after the function, foo is %d\n", foo);

int my_function_works ()
{
    printf("when I am in the function, foo is %d\n", foo);
    ...
    /* Do some stuff here */
}
```

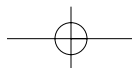
### NEED TO KNOW... AN INTERESTING NOTE ABOUT *printf* AND UNALLOCATED MEMORY



---

If the reason your program is crashing is that you are accessing data from an unallocated pointer, trying to print the data pointed to can crash your program, too! After all, if the data doesn't exist, it's invisible to the *printf* you're using for debugging.

---





What if you want to leave your debugging information in the program, but for now, you just want it to run without output? Here's a quick and dirty way to turn your debugging on and off. It relies on the C preprocessor *#define* command. We'll also use two new commands: *#ifdef* and *#endif*. These are preprocessor commands (which is the reason for the different syntax; don't worry for now about the distinction between normal commands and preprocessor commands) and they act as simple tests. If the string after an *#ifdef* statement has been defined with a *#define* statement, all lines of code between the *#ifdef* and the *#endif* will be included in the program. If the string has not been defined with a *#define* statement, those lines of code will not be compiled nor included in the program. This is a little confusing, so an example could prove helpful:

```
#define DEBUG    /* when this line exists, print out debugging information */

int foo;

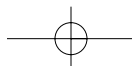
#ifdef DEBUG
    /* Only print this error if we are in debug mode */
    printf("before I enter the function, foo is %d\n", foo);
#endif

/* Enter the function my_function_works */
my_function_works();

#ifdef DEBUG
    /* Only print this error if we are in debug mode */
    printf("after the function, foo is %d\n", foo);
#endif

int my_function_works ()
{
#ifdef DEBUG
    /* Only print this error if we are in debug mode */
    printf("when I am in the function, foo is %d\n", foo);
#endif
    ...
    /* Do some stuff here */
}
```

Those lines that are between the *#ifdef* and *#endif* statements won't be evaluated unless *DEBUG* is defined at the beginning of the program. When you want debugging lines in your program, define *DEBUG*. When you want to program to work without debugging, just remove that *#define* statement.





## NEED TO KNOW... A NOTE ABOUT PREPROCESSOR COMMANDS



Lines that begin with a `#` in C are preprocessor commands, which means that they are parsed by the compiler before anything else. Because these lines are processed before any other parts of the code, they are evaluated in order, from top to bottom. The preprocessor ignores function declarations and other control structures that affect the order in which your code is run.

One last note for the sake of completeness: Operating systems generally have concepts of *output streams*, primarily *standard error* (*stderr*) and *standard output* (*stdout*). The theory is that all normal output should go to the standard output stream, and all errors should go to the standard error stream. Usually both standard output and standard error end up on your computer monitor, and when you see the text appear, you don't know—nor do you care—which stream you're seeing. But if you use the appropriate output stream, it's very easy to treat the streams differently. Perhaps you want to pipe all the output of the program into a text file for later analysis, but you want error messages to appear on your screen, not in the text file. Perhaps you don't want to see errors at all. To accommodate this kind of after-the-fact output manipulation, you can use a modified version of the *printf* function to send your errors directly to standard error:

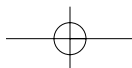
```
fprintf(stderr, "when I am in the function, foo is %d\n", foo);
```

All this debugging will be much simpler if you have used meaningful variable names and commented your code extensively. Debugging a program that crashes for no apparent reason is much more annoying than writing a few extra lines of comments.

## Introduction to Assembly Language

Sometimes, even a relatively low-level language such as C doesn't get us close enough to the hardware. A C program can be portable between different platforms and as such it loses something in efficiency. In assembly language, though, every machine instruction possible on that hardware has an assembly language translation. We use assembly language rather than writing directly in machine language, because it is easier to say *ADD address\_1 address\_2* than to say *0xbe 0x1234 0xf337*. Some would say that a pure assembly language has no instructions that don't map directly to a machine instruction, but we shall stay out of that philosophical battle.

Because of this strong correlation between a particular piece of hardware's instruction set and the assembly language usable on that hardware, assembly language programs aren't particularly portable. A program you write on one system might not be in the slightest bit usable on another system. On the other hand, assembly language programs run extremely quickly and efficiently. Instead of trusting a compiler to lay out the instructions in the most efficient manner, you can guarantee efficiency by writing the instructions exactly as they will be run by the CPU. Moreover, your hardware may have special features that are not accessible to you from a higher-level language such as C.



## NEED TO KNOW... LIMITATIONS OF THIS APPENDIX



As you can probably guess from the previous paragraphs, which assembler you use will vary based on your operating system and hardware platform. But even on any given platform, there are many different assembly language implementations you can use. On Intel, for example, you can use such assembly languages as A386, GNU *as*, HLA, SpAsm, and MASM. Some of these are relatively high-level, offering features that we think of as belonging to high-level programming languages, such as *if/else* statements and *while* loops. Others are very simple, offering not much above the level of the hardware. For this appendix, we focus on simple features and give examples using the low-level GNU assembler, *as*, for the Intel 80386 processor.

## Components of an Assembly Language Statement

An assembly language statement has four components:

- The label
- The operation
- The operands
- The comments

We examine all these concepts in detail in the subsequent sections.

### Labels

Have you ever done any BASIC programming with *GOTO*s? If you have, did somebody give you a supercilious sneer and say, “*Real* programmers don’t use *GOTO*s”? Well, now you can sneer right back—because anybody who can program in assembly language *is* a real programmer, and in assembly language, you use *GOTO*s. Oh, we call them *labels*, but don’t let that fool you.

If you haven’t used labels or *GOTO* statements before, don’t worry. The concept is very simple. A label records the memory address of the line of code that contains the label. At any point in the code, your program can *jump* to the memory address of the label:

```
/*
 * Some assembly language code goes here.
 *
 * Do you recognize these lines? They're comments. In GNU as,
 * any text between "/*" and the next "*/" is a comment,
 * even if it appears in the middle of a line. The comment character
 * may differ (sometimes it is a semicolon (;) or a pound sign (#),
 * for example), but the general format is the same. These lines
 * will not be translated into machine language.
 */
```



## 580 Appendix B • Coding 101

```

my_label:          /* The label is the word and colon at line's start */
    movl $1, %eax  /* Don't worry about what this assembly language */
                   /* command means for now. */

/* More assembly language code goes here. */

jmp my_label      /* now the program will loop back to that label, so */
                 /* the next line of code it executes will be */
                 /* "movl $1, %eax" */

```

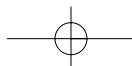
If all you do is loop under any condition back to the label, this program will just make endless circles back and forth between *my\_label* and the command *jmp my\_label*. But even low-level assembly languages provide simple conditionals. In GNU *as*, you can base the decision whether or not to make the jump based on the result of a comparison (Table B.8).

**Table B.8** GNU *as* Jump Commands

GNU as Command	Function
<code>cmpl value_1 value_2</code>	This <i>as</i> statement compares two values and stores a comparison based on the result. It can be followed by one of the <i>jump</i> commands, which will make a decision based on the results of the comparison.
<code>je label</code>	Jump to label if <i>value_1</i> equals <i>value_2</i> .
<code>jg label</code>	Jump to label if <i>value_2</i> is greater than <i>value_1</i> .
<code>jge label</code>	Jump to label if <i>value_2</i> is greater than or equal to <i>value_1</i> .
<code>jl label</code>	Jump to label if <i>value_2</i> is less than <i>value_1</i> .
<code>jle label</code>	Jump to label if <i>value_2</i> is less than or equal to <i>value_1</i> .
<code>jmp label</code>	Jump to label no matter what. This statement does not need to be preceded by the comparison.

## Operations

Two sorts of operations are possible in assembly language. The first maps directly to machine language instructions (*opcodes*) and is translated directly into machine language by the assembler. The second is a meta-command, a command that tells the assembler to do something, instead of telling the computer to do something. In GNU *as*, a meta-command is always preceded by a dot (.) character. This is good shorthand to remember. Even though we haven't introduced either command to you, you know that *.int* is a command directly to the assembler and *popl* translates to a machine instruction. (Just so you know, *.int* reserves storage for some number of integers, and *popl* pops off the top value of the stack.)





## Operands

First, a bit of terminology: An *operand* is the object of an operation. In the following equation the numbers 3 and 5 are both operands (and the + is the *operator*):

3 + 5 = ?

The C variables often act as special cases of operands. In the C statement, the number 4.0 and the variable *my\_number* are both operands:

```
my_answer = my_number / 4.0;
```

When you read about operands with assembly languages, for all practical purposes you're reading about variable assignment.

We learned in the C section of this appendix that different kinds of data take up different amounts of space. In GNU *as*, we declare the data by type in order to guarantee enough space.

The possible data types are:

- **byte** For a single byte of computer memory.
- **int** For an integer between 0 and 65535.
- **long** For an integer between 0 and 4,294,967,295.
- **ascii** For one or more characters.

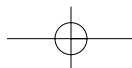
The data storage is declared in the special section of the assembly language program, which is initiated with a statement to the assembler:

```
.section .data
```

Next then the storage itself is declared, with another command to the assembler:

```
.ascii "Hello, world\0"
```

In C, character strings were automatically terminated with the null character (`\0`). In assembly language, you'll need to add that terminating character by hand.



**582 Appendix B • Coding 101**

## Sample Program

The best way to understand assembly language is to see a little bit of it. Here's a simple program that does a little bit of addition:

```
/* addition.exe */
/*
 * sample assembly language program in GNU as
 * adds together the numbers "3" and "17"
 */

/* Data section. We're not using any variables here - just holding
 * the arguments from the command line in registers, so this
 * section is blank. */

.section .data

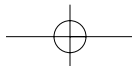
/* Text section. This section contains the actual program. */

.section .text

/* .globl defines a label which has to exist even from outside the program.
 * "_start" is a special-purpose label which tells the program that this is
 * the beginning, similar to "main()" in C. */

.globl _start
_start:

pushl $17          /* push the number 17 onto the top of the stack */
                  /* the stack is the part of memory which is currently */
                  /* being used. Think of it like a stack of cafeteria trays.*/
pushl $3           /* push the number 3 onto the top of the stack. */
                  /* now 3 is on top, with 17 beneath it. */
```



```

popl %eax          /* pop the top of value on the stack (3) -- that is, */
                  /* remove it from the stack, and put it in the */
                  /* temporary register "%eax" */

popl %ebx          /* pop the top of value on the stack (17) -- that is, */
                  /* remove it from the stack, and put it in the */
                  /* temporary register "%ebx" */

addl %eax, %ebx    /* add together the two numbers, and store the result */
                  /* in the second temporary register, %ebx */

movl $1, %eax      /* in Linux, this is the kernel's system call */
                  /* to exit the program.  When the program exits, */
                  /* whatever value is stored in register %ebx */
                  /* will be the return value of the program. */
                  /* because of our addl command, the value stored */
                  /* in %ebx is the sum of 3+17 */

int $0x80          /* this runs the software interrupt responsible */
                  /* for telling the kernel to exit */

```

After we run this program, we can test the return value of the program to find out the sum of the two numbers. As you might guess from looking at the program, we made this somewhat more complex than it needed to be. We didn't need to push 3 and 17 onto the stack, then pop them both off again in order to add them together. We could have just stored the two numbers directly into the temporary registers. But the purpose of the example was to give you a taste of assembly language.

### NEED TO KNOW... STACK TRICKINESS



The top of the stack is in reality the bottom of the stack. Yes, we know, that makes no sense. After all, both "top" and "bottom" are just fake names—what do they really mean in a computer's memory? It's not like there is gravity in the computer defining what is "top" and what is "bottom." What these terms mean is that if you think of memory addresses as having higher numbers at the top and lower numbers at the bottom, the stack grows downward, as illustrated in Table B.9.

**Table B.9** Stack Direction

Memory That Holds the Stack	Stack Sitting in Memory
Address 22	First entry placed into stack
Address 18	Second entry placed into stack
Address 16	Third entry placed into stack
Address 12	Current top of stack
Address 8	X
Address 4	X
Address 0	X

If we now choose to place another entry in the stack, it will go to address 8. So if we need to manually manipulate the *stack pointer* (stored in register `%esp`), adding a new entry to the stack means *subtracting* from the value of the stack pointer.

## Summary

The basis of assembly language is simple to learn. However, learning how to do something with it—that is a whole new kettle of fish.

In general, assembly language is only used to manipulate hardware we can't access with high-level languages or to accelerate a particularly slow section of code. However some coders prefer using assembly over a high-level language, and it is particularly useful for hardware hacking.

## Additional Reading

If you are interested in learning more about any of the topics in this appendix, we recommend the following books:

- *Structured Computer Organization*, fourth edition, by Andrew S. Tannenbaum (Prentice-Hall, 1998)
- *A Book on C*, by Al Kelley and Ira Pohl (The Benjamin/Cummings Publishing Company, 1995)
- *C Programming Language*, second edition, by Brian W. Kernighan and Dennis Ritchie (Prentice Hall, 1988)
- *The Art of Assembly Language*, by Randall Hyde (No Starch Press, 2003 or <http://webster.cs.ucr.edu>)
- *Programming from the Ground Up*, by Jonathan Bartlett (<http://savannah.nongnu.org/projects/pgubook>)

