Chapter 2: Getting Started in C

YOUR FIRST C PROGRAM

The best way to get started with C is to actually study a program, so load the file named trivial.c and display it on the monitor. You are looking at the simplest possible C program. There is no way to simplify this program or to leave anything out. Unfortunately, the program doesn't do anything.

The word **main** is very important, and must appear once, and only once in every C program. This is the point where execution is begun when the program is run. We will see later that this does not have to be the first statement in the program but it must exist as the entry point. Following the main program name is a pair of parentheses which are an indication to the compiler that this is a function. We will cover exactly what a function is in due time. For now, I suggest that you simply include the pair of parentheses.

The two curly brackets, properly called braces, are used to define the limits of the program itself. The actual program statements go between the two braces and in this case, there are no statements because the program does absolutely nothing. You can compile and run this program, but since it has no executable statements, it does nothing. Keep in mind, however, that it is a valid C program. When you compile this program, you may get a warning depending on how your compiler is set up. You can ignore the warning and we will discuss it later in this tutorial.

A PROGRAM THAT DOES SOMETHING

For a much more interesting program, load the program named wrtsome.c and display it on your monitor. It is the same as the previous program except that it has one executable statement between the braces.

The executable statement is a call to a function supplied as a part of your C library. Once again, we will not worry about what a function is, but only how to use this one named **printf()**. In order to output text to the monitor, the desired text is put within the function parentheses and bounded by quotation marks. The end result is that whatever is included between the quotation marks will be displayed on the monitor when the program is run.

Notice the semi-colon at the end of the line. C uses a semi-colon as a statement terminator, so the semi-colon is required as a signal to the compiler that this line is complete. This program is also executable, so you can compile and run it to see if it does what you think it should. It should cause the text between the quotation marks to appear on the monitor.

You may get two warnings when you compile this program and each of the remaining programs in this chapter. All warnings in this chapter can be ignored.

ANOTHER PROGRAM WITH MORE OUTPUT

Load the program wrtmore.c and display it on your monitor for an example of more output and another small but important concept. You will see that there are four program statements in this program, each one being a call to the function **printf().** The top line will be executed first, then the next, and so on, until the fourth line is complete. The statements are executed in order from top to bottom.

Notice the funny character near the end of the first line, namely the backslash. The backslash is used in the **printf()** statement to indicate that a special control character is following. In this case, the "**n**" indicates that a newline is requested. This is an indication to return the cursor to the left side of the monitor and move down one line. It is commonly referred to as a carriage return/line feed. Any place within text that you desire, you can put a newline character and start a new line. You could even put it in the middle of a word and split the word between two lines. The C compiler considers the combination of the backslash and letter **n** as one character.

A complete description of this program is now possible. The first **printf()** outputs a line of text and returns the carriage. (Of course, there is no carriage, but the cursor is moved to the next line on the monitor. The terminology carries over from the days of teletypes.) The second **printf()** outputs a line but does not return the carriage so that the third line is appended to the end of the second, then followed by two carriage returns, resulting in a blank line. Finally the fourth **printf()** outputs a line followed by a carriage return and the program is complete.

After compiling and executing wrtmore.c, the following text should be displayed on your monitor;

```
This is a line of text to output.
And this is another line of text.

This is a third line.
```

Compile and run this program to see if it gives you this output. It would be a good idea at this time for you to experiment by adding additional lines of printout to see if you understand how the statements really work. Add a few carriage returns in the middle of a line to prove to yourself that it works as stated, then compile and execute the modified program. The more you modify and compile the example programs, the more you will learn as you work your way through this tutorial.

LET'S PRINT SOME NUMBERS

Load the file named oneint.c and display it on the monitor for our first example of how to work with data in a C program. The entry point **main** should be clear to you by now as well as the beginning brace. The first new thing we encounter is line 3 containing `int index;` which is used to define an integer variable named **index**. The word **int** is a keyword in C, and can not be used for anything else. It defines a variable that can store a value in the range from -32768 to 32767 in most C compilers for microcomputers. The variable name, **index**, can be any name that follows the rules for an identifier and is not one of the keywords for C. The final character on the line, the semi-colon, is the statement terminator as discussed earlier.

Note that, even though we have defined a variable, we have not yet assigned a value to it, so it contains an undefined value. We will see in a later chapter that additional integers could also be defined on the same line, but we will not complicate the present situation.

Observing the main body of the program, you will notice that there are three statements that assign a value to the variable **index**, but only one at a time. The statement in line 5 assigns the value of 13 to **index**, and its value is printed out by line 6. (We will see how shortly. Just trust me for the time being.) Later, the value of 27 is assigned to **index**, and finally 10 is assigned to it, each value being printed out. It should be intuitively clear that **index** is indeed a variable and can store many different values but only one value at a time of course.

Please note that many times the words "printed out" are used to mean "displayed on the monitor". You will find that in many cases experienced programmers take this liberty, probably due to the **printf()** function being used for monitor display.

HOW DO WE PRINT NUMBERS ?

To keep our promise, let's return to the **printf()** statements for a definition of how they work. Notice that they are all identical and that they all begin just like the **printf()** statements we have seen before. The first difference occurs when we come to the % character. This is a special character that signals the output routine to stop copying characters to the output and do something different, namely output the value of a variable. The % sign is used to signal the output of many different types of variables, but we will restrict ourselves to only one for this example. The character following the % sign is a **d**, which signals the output routine to get a decimal value and output it. Where the decimal value comes from will be covered shortly. After the **d**, we find the familiar **\n**, which is a signal to return the video "carriage", and the closing quotation mark.

All of the characters between the quotation marks define the pattern of data to be output by this statement. Following the output pattern, there is a comma followed by the variable name **index**. This is where the **printf()** statement gets the decimal value which it will output because of the **%d** we saw earlier. We could add more **%d** output field descriptors anywhere within the brackets and more variables following the description to cause more data to be printed with one statement. Keep in mind however, that the number of field descriptors and the number of variable definitions must be the same or the runtime system will generate something we are not expecting.

Much more will be covered at a later time on all aspects of input and output formatting. A reasonably good grasp of these fundamentals are necessary in order to understand the following lessons. It is not necessary to understand everything about output formatting at this time, only a fair understanding of the basics.

Compile and run oneint.c and observe the output. Two programming exercises in this chapter are based on this program.

HOW DO WE ADD COMMENTS IN C ?

Load the file named comments.c and observe it on your monitor for an example of how comments can be added to a C program. Comments are added to make a program more readable to you but represent nonsense to the compiler, so we must tell the compiler to ignore the comments completely by bracketing them with special characters. The slash star combination is used in C for comment delimiters, and are illustrated in the program at hand. Please note that the program does not illustrate good commenting practice, but is intended to illustrate where comments can go in a program. It is a very sloppy looking program.

The first slash star combination introduces the first comment and the star slash at the end of the first line terminates this comment. Note that this comment is prior to the beginning of the program illustrating that a comment can precede the program itself. Good programming practice would include a comment prior to the program with a short introductory description of the program. The comment in line 3 is after the main program entry point and prior to the opening brace for the program code itself.

The third comment starts after the first executable statement in line 5 and continues for four lines. This is perfectly legal because a comment can continue for as many lines as desired until it is terminated. Note carefully that if anything were included in the blank spaces to the left of the three continuation lines of the comment, it would be part of the comment and would not be compiled, but totally ignored by the compiler. The last comment, in line 11, is located following the completion of the program, illustrating that comments can go nearly anywhere in a C program.

Experiment with this program by adding comments in other places to see what will happen.

Comment out one of the **printf()** statements by putting comment delimiters both before and after it and see that it does not get executed causing a line of printout.

Comments are very important in any programming language because you will soon forget what you did and why you did it. It will be much easier to modify or fix a well commented program a year from now than one with few or no comments. You will very quickly develop your own personal style of commenting.

Some C compilers will allow you to "nest" comments which can be very handy if you need to "comment out" a section of code during debugging. Since nested comments are not a part of the ANSI-C standard, none will be used in this tutorial. Check the documentation for your compiler to see if they are permitted with your implementation of C. Even though they may be allowed, it is a good idea to refrain from their use, since they are rarely used by experienced C programmers.

A VERY USEFUL EXTENSION

Load the program named cppcoms.c for an example program with a very useful extension to the C programming language in recent years. Comments in C++, a language that was based on C, begin with a double slash and continue to the end of the line. Many compiler writers are adding this comment style to their compilers in addition to the slash-star delimiter defined already.

Using the older style slash-star comment, it is fairly easy to begin a comment, and forget to terminate it, inadvertently commenting out a block of code. In some circumstances, this can be very difficult to track down, so the designer of C++ added the single line comment. If your compiler supports this style of comment, you may choose to use it, but you must keep in mind that if you ever want to use a different compiler that doesn't support this extension, you may have a lot of code to modify. Of course, that assumes you add lots of comments to your code.

GOOD FORMATTING STYLE

Load the file goodform.c and observe it on your monitor. It is an example of a well formatted program. Even though it is very short and therefore does very little, it is very easy to see at a glance what it does. With the experience you have already gained in this tutorial, you should be able to very quickly grasp the meaning of the program in it's entirety. Your C compiler ignores all extra spaces and all carriage returns giving you considerable freedom in formatting your program. Indenting and adding spaces is entirely up to you and is a matter of personal taste. Compile and run the program to see if it does what you expect it to do.

Now load and display the program uglyform.c and observe it. How long will it take you to figure out what this program will do ? It doesn't matter to the compiler which format style you use, but it will matter to you when you try to debug your program. Compile this program and run it. You may be surprised to find that it is the same program as the last one, except for the formatting. Don't get too worried about formatting style yet. You will have plenty of time to develop a style of your own as you learn the C language. Be observant of styles as you see C programs in magazines and books.

This should pretty well cover the basic concepts of programming in C, but as there are many other things to learn, we will forge ahead to additional program structure. It will definitely be to your advantage to do the programming exercises at the end of each chapter. They are designed to augment your studies and teach you to use your compiler.

Programming Exercise:

---

1. Write a program to display your name on the monitor.

2. Modify the program to display your address and phone number on separate lines by adding two additional **printf()** statements.
3. Remove line 5 from oneint.c by commenting it out, then compile and execute the resulting program to see the value of an uninitialized variable. This can be any value within the allowable range. If it happens to have the value of zero, that is only a coincidence, but then zero is the most probable value to be in an uninitialized variable because there are lots of zero values floating around in a computers memory.
4. Add the following two lines just prior to the closing brace of oneint.c to see what it does. Study it long enough to completely understand the result.

```
printf("Index is %d\n it still is %d\n it is %d",

              index, index, index);
```

Index Previous Next . . . . . *C Tutorial*
