Chapter 3: Program Control

THE WHILE LOOP

The C programming language has several structures for looping and conditional branching. We will cover them all in this chapter and we will begin with the while loop.

The while loop continues to loop while some condition is true. When the condition becomes false, the looping is discontinued. It therefore does just what it says it does, the name of the loop being very descriptive.

Load the program while.c and display it for an example of a while loop. We begin with a comment and the program entry point **main()**, then go on to define an integer variable named **count** within the body of the program. The variable is set to zero and we come to the while loop itself. The syntax of a while loop is just as shown here. The keyword **while** is followed by an expression of something in parentheses, followed by a compound statement bracketed by braces. As long as the expression in the parenthesis is true, all statements within the braces will be repeatedly executed. In this case, since the variable **count** is incremented by one every time the statements are executed, it will eventually reach 6. At that time the statement will not be executed because **count** is not less than 6, and the loop will be terminated. The program control will resume at the statement following the statements in braces.

We will cover the compare expression, the one in parentheses, in the next chapter. Until then, simply accept the expressions for what you think they should do and you will be correct for these simple cases.

Several things must be pointed out regarding the while loop. First, if the variable **count** were initially set to any number greater than 5, the statements within the loop would not be executed at all, so it is possible to have a while loop that never is executed. Secondly, if the variable were not incremented in the loop, then in this case, the loop would never terminate, and the program would never complete. Finally, if there is only one statement to be executed within the loop, it does not need delimiting braces but can stand alone.

Compile and run this program after you have studied it enough to assure yourself that you understand its operation completely. Note that the result of execution is given for this program, (and will be given for all of the remaining example programs in this tutorial) so you do not need to compile and execute every program to see the results. Be sure to compile and execute some of the programs however, to gain experience with your compiler. Once again, you may get compiler warnings in this chapter which can be ignored temporarily.

You should also modify any programs that are not completely clear to you until you understand them completely. The best way to learn is to try various modifications yourself.

THE DO-WHILE LOOP

A variation of the while loop is illustrated in the program dowhile.c, which you should load and display. This program is nearly identical to the last one except that the loop begins with the keyword **do**, followed by a compound statement in braces, then the keyword **while**, and finally an expression in parentheses. The statements in the braces are executed repeatedly as long as the expression in parentheses is true. When the expression in parentheses becomes false, execution is terminated, and control passes to the statements following this statement.

Several things must be pointed out regarding the do-while loop. Since the test is done at the end of the loop, the statements in the braces will always be executed at least once. Secondly, if the variable **i** were not changed within the loop, the loop would never terminate, and hence the program would never terminate.

It should come as no surprise to you that these loops can be nested. That is, one loop can be included within the compound statement of another loop, and the nesting level has no limit. This will be illustrated later.

Compile and run this program to see if it does what you think it should do.

THE FOR LOOP

The for loop is really nothing new, it is simply a new way to describe the while loop. Load and display the file named forloop.c on your monitor for an example of a program with a for loop. The for loop consists of the keyword **for** followed by a rather large expression in parentheses. This expression is really composed of three fields separated by semi-colons. The first field contains the expression `"index = 0"` and is an initializing field. Any expressions in this field are executed prior to the first pass through the loop. There is essentially no limit as to what can go here, but good programming practice would require it to be kept simple. Several initializing statements can be placed in this field, separated by commas.

The second field, in this case containing `"index < 6"`, is the test which is done at the beginning of each pass through the loop. It can be any expression which will evaluate to a true or false. (More will be said about the actual value of true and false in the next chapter.)

The expression contained in the third field is executed each time the loop is exercised but it is not executed until after those statements in the main body of the loop are executed. This field, like the first, can also be composed of several operations separated by commas.

Following the **for()** expression is any single or compound statement which will be executed as the body of the loop. A compound statement is any group of valid C statements enclosed in braces. In nearly any context in C, a simple statement can be replaced by a compound statement that will be treated as if it were a single statement as far as program control goes. Compile and run this program.

You may be wondering why there are two statements available that do exactly the same thing because the **while** and the **for** loop do exactly the same thing. The **while** is convenient to use for a loop when you don't have any idea how many times the loop will be executed, and the **for** loop is usually used in those cases when you are doing a fixed number of iterations. The **for** loop is also convenient because it moves all of the control information for a loop into one place, between the parentheses, rather than at both ends of the code. It is your choice as to which you would rather use.

THE IF STATEMENT

Load and display the file ifelse.c for an example of our first conditional branching statement, the if. Notice first, that there is a for loop with a compound statement as its executable part containing two if statements. This is an example of how statements can be nested. It should be clear to you that each of the if statements will be executed 10 times.

Consider the first **if** statement. It starts with the keyword **if** followed by an expression in parentheses. If the expression is evaluated and found to be true, the single statement following the **if** is executed, and if false, the following statement is skipped. Here too, the single statement can be replaced by a compound statement composed of several statements bounded by braces. The expression `"data == 2"` is simply asking if the value of **data** is equal to 2. This will be explained in detail in the next chapter. (Simply suffice for now that if `"data = 2"` were used in this context, it would mean a completely different thing. You must use the double equal sign for comparing values.)

NOW FOR THE IF-ELSE

The second if is similar to the first with the addition of a new keyword, the **else** in line 15. This simply says that if the expression in the parentheses evaluates as true, the first expression is executed, otherwise the expression following the **else** is executed. Thus, one of the two expressions will always be executed, whereas in the first example the single expression was either executed or skipped. Both will find many uses in your C programming efforts. Compile and run this program to see if it does what you expect.

THE BREAK AND CONTINUE

Load the file named breakcon.c for an example of two new statements. Notice that in the first for loop, there is an **if** statement that calls a **break** if **xx** equals 8. The **break** will jump out of the loop you are in and begin executing statements following the loop, effectively terminating the loop. This is a valuable statement when you need to jump out of a loop depending on the value of some results calculated in the loop. In this case, when **xx** reaches the value of 8, the loop is terminated and the last value printed will be the previous value, namely 7. The **break** always jumps out of the loop just past the terminating brace.

The next **for** loop starting in line 12, contains a **continue** statement which does not cause termination of the loop but jumps out of the present iteration. When the value of **xx** reaches 8 in this case, the program will jump to the end of the loop and continue executing the loop, effectively eliminating the **printf()** statement during the pass through the loop when **xx** is eight. The **continue** statement always jumps to the end of the loop just prior to the terminating brace. At that time the loop is terminated or continues based on the result of the loop test.

Be sure to compile and execute this program.

THE SWITCH STATEMENT

Load and display the file switch.c for an example of the biggest construct yet in the C language, the switch. The switch is not difficult, so don't let it intimidate you. It begins with the keyword **switch** followed by a variable in parentheses which is the switching variable, in this case **truck**. As many cases as needed are then enclosed within a pair of braces. The reserved word **case** is used to begin each case, followed by the value of the variable for that case, then a colon, and the statements to be executed.

In this example, if the variable named **truck** contains the value 3 during this pass of the switch statement, the **printf()** in line 9 will cause `"The value is three"` to be displayed, and the **break** statement will cause us to jump out of the switch. The **break** statement here works in much the same manner as the loop, it jumps out just past the closing brace.

Once an entry point is found, statements will be executed until a **break** is found or until the program drops through the bottom of the switch braces. If the variable has the value 5, the statements will begin executing at line 13 where `"case 5 :"` is found, but the first statements found are where the case 8 statements are. These are executed and the **break** statement in line 17 will direct the execution out of the bottom of the switch just past the closing brace. The various case values can be in any order and if a value is not found, the default portion of the switch will be executed.

It should be clear that any of the above constructs can be nested within each other or placed in succession, depending on the needs of the particular programming project at hand. Note that the **switch** is not used as frequently as the loop and the if statements. In fact, the switch is used infrequently but should be completely understood by the serious C programmer. Be sure to compile and run switch.c and examine the results.

THE EVIL GOTO STATEMENT

Load and display the file gotoex.c for an example of a file with some goto statements in it. To use a **goto** statement, you simply use the reserved word **goto** followed by the symbolic name to which you wish to jump. The name is then placed anywhere in the program followed by a colon. You can jump nearly anywhere within a function, but you are not permitted to jump into a loop, although you are allowed to jump out of a loop.

This particular program is really a mess but it is a good example of why software writers are trying to eliminate the use of the **goto** statement as much as possible. The only place in this program where it is reasonable to use the **goto** is the one in line 18 where the program jumps out of the three nested loops in one jump. In this case it would be rather messy to set up a variable and jump successively out of each of the three nested loops but one **goto** statement gets you out of all three in a very concise manner.

Some persons say the **goto** statement should never be used under any circumstances, but this is narrow minded thinking. If there is a place where a **goto** will clearly do a neater control flow than some other construct, feel free to use it. It should not be abused however, as it is in the rest of the program on your monitor.

Entire books are written on "gotoless" programming, better known as Structured Programming. These will be left to your study. One point of reference is the Visual Calculator described in Chapter 14 of this tutorial. This program is contained in four separately compiled files and is a rather large complex program. If you spend some time studying the source code, you will find that there is not a single goto statement anywhere in it.

Compile and run gotoex.c and study its output. It would be a good exercise to rewrite it and see how much more readable it is when the statements are listed in order.

FINALLY, A MEANINGFUL PROGRAM

Load the file named tempconv.c for an example of a useful, even though somewhat limited program. This is a program that generates a list of centigrade and fahrenheit temperatures and prints a message out at the freezing point of water and another at the boiling point of water.

Of particular importance is the formatting. The header is simply several lines of comments describing what the program does in a manner that catches the readers attention and is still pleasing to the eye. You will eventually develop your own formatting style, but this is a good way to start. Also if you observe the **for** loop, you will notice that all of the contents of the compound statement are indented 3 spaces to the right of the **for** keyword, and the closing brace is lined up under the "f" in for. This makes debugging a bit easier because the construction becomes very obvious. You will also notice that the **printf()** statements that are in the **if** statements within the big for loop are indented three additional spaces because they are part of another construct.

This is the first program in which we used more than one variable. The three variables are simply defined on three different lines and are used in the same manner as a single variable was used in previous programs. By defining them on different lines, we have an opportunity to define each with a comment. It would be possible to define them on one line, but to do so would remove the ability to include a comment on each line. This is illustrated in the next program. Be sure to compile and execute this program.

ANOTHER POOR PROGRAMMING EXAMPLE

Recalling uglyform.c from the last chapter, you saw a very poorly formatted program. If you load and display dumbconv.c you will have an example of poor formatting which is much closer to what you will find in practice. This is the same program as tempconv.c with the comments removed and the variable names changed to remove the descriptive aspect of the names. Although this program does exactly the same as the last one, it is much more difficult to read and understand. You should begin to develop good programming practices now by studying this program to learn what not to do.

OUR FIRST STYLE PROGRAM

Example program: **style1.c**

This program does nothing practical except to illustrate various styles of programming and how to combine some of the constructs introduced in this chapter. There is nothing in this program that we have not studied so far in this tutorial. The program is heavily commented and should be studied in detail by the diligent C student to begin learning proper C programming style. Like all other example programs, this one can be compiled and executed, and should be.

Programming Exercise:

1. Write a program that writes your name on the monitor ten times. Write this program three times, once with each looping method.
2. Write a program that counts from one to ten, prints the values on a separate line for each, and includes a message of your choice when the count is 3 and a different message when the count is 7.

Index Previous Next . . . . *C Tutorial*

The Webwizard