Lesson 3 PROGRAM CONTROL (LOOPS)

The C programming language has several structures for looping and conditional branching. There are three major loop structures in C. The **for** loop, the **while** loop and the **do while** loop. **The for loop**

Example for.c

```
#include<stdio.h>
void main(void)
{
 int index;
 for(index=0;index < 10;index++)
   {
     printf("%3d",index*index);
     printf("\n");
    }
}
```

Load the file named for.c on your monitor for an example of a program with a **for** loop.

The **for** loop consists of the keyword **for** followed by an expression in parentheses. This expression is composed of three fields separated by semi-colons. The first field contains the expression "index = 0". This 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 < 10", 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.

The expression contained in the third field is executed each time the loop is exercised but it is not executed until all the 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 almost 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.

Nesting of for loops

Nesting of for loops is allowed in C. To demonstrate this, consider the following program that prints out the multiplication table

```
#include<stdio.h>
void main(void)
{
  int rows,cols;
  for(rows=1;rows<=12;rows++)    /* outside loop */
     {
      for(cols=1;cols<=12;cols++) /* inside loop */
      printf("%5d",rows*cols);
      printf("\n");
      }
 }
```

In this program the inner loop steps through 12 rows (1-12) and the outer loop steps through 12 columns (1-12). For each row the inner loop is cycled through once, then a new line is printed in

preparation for the next row. We then print out the product of row times col. You can try and line up the different columns by changing the field width specifier.

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.

Example while.c

```
#include<stdio.h>   /*Demonstrates a while loop*/
void main(void)
     {
      int count=0;
       while(count<6)
           {
            printf("\nThe value of count is %d\n",count);
            count++;
           }
     }
```

Load the program while.c which is 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. This variable is set to zero. The syntax of a **while** loop shown. 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. The statements will not be executed when **count** is not less than 6, hence the loop will be terminated. The program control will resume at the statement following the statements in braces.

Several things must be pointed out regarding the **while** loop. Firstly, 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. Make sure you understand how the code works.

You should make some modifications to this program and make sure it delivers the desired effects.

Note: Sometimes in situations where the number of iterations in a loop are known in advance a **while** loop is less appropriate. In this case the for loop is a more natural choice. In general **while** loops are more appropriate than **for** loops when the condition that terminates the loop occurs unexpectedly.

Nesting of while loops

The following program gives an example of nesting of while loops

```
#include<stdio.h>
 void main(void)
{
  char letter1 = ' ';
```

```
 char letter2 = ' ';

 while( letter2 != 'c')
       {
       while( letter1 != 'f' )
         {
          puts("Try and Guess the first letter\n");
/*scanf("%c",&letter1);*/
          letter1 = getc(stdin);
         }
       puts("First letter guess correctly, not guess the second\n");
/* scanf("%c",&letter2); */
       letter2 = getc(stdin);
      }
 printf("Well done, both guess correctly. GAME IS OVER\n");
}
```

In this program the inner **while** loop is used to check to see if your guess for letter1 is correct or not. The outer loop is used to check is letter2 is correct or not. Run the code above and comment on any possible improvements.

The do-While Loop

Example dowhile.c

```
#include<stdio.h>
void main(void)
     {
      float x=2;
      do
       {
        x = x*2;
        printf("x = %f\n",x);
        } while( x < 100);
      }
```

A variation of the **while** loop is illustrated in the program dowhile.c, which you should load and run.

This syntax is nearly identical to the previous one except that now, 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 the parentheses is true. When the expression in parentheses becomes false, execution is terminated, and control passes to the statements following this statement.

Several things should 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 condition for termination is not met within the loop, the loop would never terminate, and hence the program would never terminate.

It should come as no surprise to you that the do while loop can also be nested.

What loop to use ?

The **while loop** is convenient to use for a loop when you don't have any idea how many times the loop will be executed.

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. Depending on how they are used, it is possible with each of these two loops to never execute the code within the loop at all. This is because the test is done at the beginning of the loop and the test may fail during the first iteration. The **do-while** loop however, due to the fact that the code within the loop is executed prior to the test, will always execute the code at least once.

The break statement

The **break** statement bails you out of a loop as soon as its executed. Its mainly used when an unexpected condition occurs.

The continue statement

The **continue statement** is inserted in the body of the loop and when executed takes you back to the beginning of the loop, bypassing any statements not yet executed.

Both break and continue are useful when you want to terminate the loop or go back to the top for the next iteration.

The goto statement

Some people say that the **goto** statement should never be used under any circumstances. Their conjecture is that use of the statement can lead to unreliable, unreadable and hard to debug code. I am of the opinion that if you feel that a **goto** statement will clearly do a neater control flow than some other construct, then you should use it. In the next lesson, we shall give an example of where a **goto** statement can be used.

PROGRAMMING EXERCISES

1. Write a short program that prints out the cube of the first 5 Natural numbers using a for loop.
2. Print out the ASCII Table from 1-255 using a for loop. Hint:Use the fact that format specifiers can interpret the same variable in different ways.
3. Write a program that reads in a character and prints out the ASCII code for that character. If return is pressed than the program exits.
4. Explain when it may be better to use the “while” loop instead of the “for” loop.
5. Write a program that demonstrates the subtle difference between the “while” loop and the “do while” loop.
6. What will the following code output to the screen

```
(a).  i=1; while(1){i++; printf("%d",i);}
(b).  for(i=1;i<=120;1*=3) printf("%d ",i);
```

7. Modify any program so that it prompts the user if they wish to repeat the program.
8. Explain the use of **continue** and **break**.
9. Write a program in which you enter the number of bits in a binary number, enter these bits and then calculate the decimal equivalent. A sample run is given below.

```
ENTER NO. OF BITS-> 4
bit 1  1 bit 2  0 bit 3  1   bit 4
1 DECIMAL EQUIVALENT OF (1011) is
13
```

10. Modify your program from the last lesson to calculate the standard deviation of a set of number that the user types in – allow them use 2-10 numbers.