Here we go loop-dee-loo. Here we go loop-dee-light...

I think I might be going a bit loopy. Oh well, here we go.....

The While Loop

The **while** loop is a variation of the **for** loop. Like the **for** loop, the **while** loop is a looping structure which allows the programmer to specify that an action is to be repeated while some condition remains true (or, in other words, until the condition becomes false).

Please be aware that any problem that can be solved using a **for** loop, can also be solved using a **while** loop, and vice-versa. The reason we have the different loops is because, based on the problem, one loop may be better suited to solving the problem than the other. There are some very subtle differences between the **for** and the **while** loops, which we will discuss here. I also want to point out that first-time programmers tend to pick one loop and stick with it for all of their programming problems. Only when new programmers have some experience do they feel confident enough to choose different loops for solving different problems. I think I used the **for** loop for at least 3 months before I ever attempted to use any other loop!

The **while** loop is often chosen when the programmer ***does not know how many times the loop will be executed***. That is, instead of a *counter* controlling the loop, some other condition will control the loop. For example, if we want to write a program which prompts the user for 10 positive numbers and we keep a running total, and then output the sum, we could easily do this with a **for** loop. However, if we want to prompt the user for as many numbers as the user wants, and only stop when the user enters something like: -999, then we are not sure how many times the loop will execute. It will stop only when the user enters -999. This is an ideal application for using a **while** loop, and I will code it below.

However, before I get to that, I'd like to go back to the vegetable example from last week. (Have you noticed my food fetish??!?) Recall, if I told my children to eat 5 bites of vegetables, I could use a **for** loop to represent that. However, if I told my children to eat their vegetables until they were full, then I would have no idea how many bites they would be taking. They would stop only when they were full. This is an ideal situation for a **while** loop, rather than a for loop.

Now, back to the problem mentioned above. The following program will prompt the user to enter some numbers to add. It will keep prompting, and adding until the user enters **-999**. The program will then display the total of the numbers entered by the user. I will use a **while** loop here, since I do not know how many times to run the loop. Please pay particular attention to the format of the while loop.

A sample output of the program below might look like:

Enter a positive number (-999 to end): 10
Enter a positive number (-999 to end): 25
Enter a positive number (-999 to end): 923
Enter a positive number (-999 to end): 45
Enter a positive number (-999 to end): 87
Enter a positive number (-999 to end): -999

The total of your numbers entered is 1090

```
#include < stdio.h >
void main (void)
{
      /* Declare variables. */

      int  num, total = 0;

     /* Prompt user for first number */

     printf ("Enter a positive number (-999 to end): ");
     scanf ("%i", &num);

     /* If user did not enter -999 as first number, continue
         prompting for number until user enters -999,
         keeping track of the total, otherwise, skip loop.  */

      while (num != -999)
      {
            total = total + num;
            printf ("Enter a positive number (-999 to end): ");
            scanf ("%i", &num);

       } /*     end while loop */

       /* Display total to user. */

       printf ("\nThe total of your numbers entered is %i\n", total);

} /*   end main    */
```

Notice the ***format*** of the **while** loop:

```
while (condition)
{
```

```
    statement(s);
}
```

As mentioned previously, the **while** loop will continue repeating until the condition becomes false. Also, like the **for** loop, the body of the **while** loop can contain 1 or more statements. If the body is just a single statement, then the begin and end braces are not needed, however the code should still be indented for readability.

Note: There must be some code, inside the body of the **while** loop, which modifies the loop control variable, so that it will eventually terminate the loop. In our example, variable **num** is the loop control variable, and it is modified each time the user enters a value. If the loop control variable is not modified within the body of the **while** loop, you will have an infinite loop. This is not an issue with the **for** loop, as the loop control variable is incremented or decremented in the for statement itself (the 3rd segment).

The "**sequence**" or "**flow control**" of the while loop is as follows:

1. The loop condition is evaluated.
2. If true, the body of the loop is executed, including the modification of the loop control variable. Go to step 1.
3. If false, the loop is terminated. Go to step 4.
4. Program execution continues with next executable statement following the while loop block.

Note: If the loop condition is false initially, then the body of the **while** loop will never be executed. This was also the case with **for** loops.

I will illustrate below how the same problem can be solved using a **while** loop or a **for** loop. I will not use the above example, as we have not yet covered the "**break**" statement which would be needed if I converted the above **while** loop to a **for** loop. Instead, I will convert the program we reviewed last week, which squared the numbers from 1 to 5.

SQUARING NUMBERS FROM 1 TO 5 USING A WHILE LOOP

```
#include < stdio.h >
void main(void)
{

    /* Variable Declarations. */
    /*  ----------------------------- */

    int    x,  result;
```

```
    /* Calculate and output the squares for numbers between 1 and 5. */
    /* ------------------------------------------------------------------------- */

    x = 1;
    while ( x <= 5 )
    {
        result = x * x;
        printf ("%i squared is %i\n",  x, result);
        x++;                /*  increment variable x     */

    } /*      end while loop */

    /* Output the final greeting. */
    /* ------------------------------- */

    printf ("\nGoodbye!\n");>

} /*   end main   */
```

The output of the above program is the same as the program we reviewed last week using a for loop. That is:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25

Goodbye!
```

Notice that the 3 components of the **for** loop (initial value, final value, increment/decrement) are also present in this **while** loop. Setting the initial value is performed in the statement right before the while loop (**x = 1**), the final value (or test) is performed right in the **while** statement itself (**x <= 5**), and the increment is performed within the body of the **while** loop (**x++**).

Program 4.6 on page 56 is a straightforward program which illustrates the **while** loop. Please review it on, and post questions in the discussion are if you are unclear about any portion of that program.

Programs 4.7 and 4.8 on pages 58 - 60 also contain **while** loops, but their algorithms are somewhat complex. I am not requiring that you review these programs, as I feel you have seen enough examples which illustrate the basic fundamentals of a **while** loop. However, if you are looking for additional examples, both of those programs are interesting and certainly worth looking at.

Next, we discuss the final repetition structure: The "**do**" loop.