Do-bee-do-bee-do...

Sinatra was one of my favorites.....

The Do Loop

The third, and final repetition structure is known as the "**do**" loop. It is similar to the **while** loop in that it physically resembles the **while**, and it is a good loop to choose if you do not know how many iterations of the loop there will be. The main difference however, between the **do** loop and the **while** loop is that the **do** loop performs its test on the loop condition AFTER the body of the loop has been executed at least once. ***So, the do loop will always execute at least once***, the **while** loop (and **for** loop) may not (if the loop condition is false to begin with).

The *format* of the **do** loop is as follows:

```
do
{
    statement(s);

} while (condition);
```

Notice I put the "while" portion of the **do** loop on the same line as the end brace, rather than on the next line, as the textbook illustrates. This is allowed, and preferred. The reason that this is preferred is because, if we use the syntax shown in the textbook, the while portion of the **do** loop actually looks like the *start* of a new **while** loop, with a semicolon at the end. This actually looks quite confusing at first glance, as shown below:

```
do
{
    statement(s);

}
while (condition);
```

The **flow control** for the do loop is as follows:

1. The body of the loop (including modification of the control value) is executed.
2. The loop condition is evaluated.
3. If true, go to step 1.
4. If false, loop is terminated. Go to step 5.
5. Program execution continues with next executable statement following the do loop.

I think it's time for some examples. First of all, before we look at some code, let's go back to my children and their vegetables. (Do you think I need counseling???) In the prior example, they were to take a bite of their vegetables until they were full. Well, what if they were full right away? Then they wouldn't even bother to take a single bite. (Hey, this example is sounding quite familiar!) However, if I used a do loop, then they would have to take a bite first, then test to see if they were full before they decided to stop. Now, that's a loop I can use! :-)

Back to code. Let's take a look at our example from the previous topic where we prompted the user for numbers until the user entered **-999**.

The program then went on to total up the numbers entered. The difference between this program and the program using the while loop, is that the body of the do loop will be entered at least once.

Below is the program from the previous topic, with a simple modification from the while loop to the do loop:

```
#include < stdio.h > void main

(void)
{
     /* Declare variables. */

     int    num, total = 0;
     char c;

    /* Prompt user for first number */

    printf ("Enter a positive number (-999 to end): ");
    scanf ("%i", &num);
    while ( (c = getchar() != '\n') && c != EOF); /* clear input buffer */

    /* Prompt user for numbers until user enters -999,
       keeping track of the total.  */

     do
     {
          total = total + num;
          printf ("Enter a positive number (-999 to end): ");
          scanf ("%i", &num);
          while ( (c = getchar() != '\n') && c != EOF); /* clear input buffer */

      } while ( num != -999 );

      /* Display total to user. */
```

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

} /*end main */
```

Now, the problem with the above program is that if the user enters **-999** initially (when first prompted), the **while** loop would have handled it well, as the loop would not have been entered. However, in this case, if the user enters **-999** in the first prompt, the program continues to prompt for at least 1 more value. This is not good. So, we either go back to the **while** loop, or we modify this **do** loop to make it do what we need it to do. I have come up with the following solution:

```
#include < stdio.h >

void main (void)
{
    /* Declare variables. */

    int  num,  total = 0;
    char c;

   /* Prompt user for numbers until user enters -999,
      keeping track of the total. */

    num = 0;
    do
    {
        total = total + num;
        printf ("Enter a positive number (-999 to end): ");
        scanf ("%i", &num);
        while ( (c = getchar() != '\n') && c != EOF); /* clear input buffer */
     } while ( num != -999 );

     /* Display total to user. */

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

} /* end main */
```

In this case, the prompt exists only in the loop, and the loop will exit if user enters **-999** the first time. This is actually a better way to handle this problem, as we do not need to repeat the prompt both before, and inside the loop.

This next example is the "square" example, however it is processed using a **do** loop.

SQUARING NUMBERS FROM 1 TO 5 USING A DO LOOP

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

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

      int     x,  result;

      /* Output initial greeting. */
      /*  ---------------------------- */

      printf ("\nHello, and welcome to the squares program!\n\n);

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

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

      } while ( x <= 5);

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

      printf ("\nGoodbye!\n");

} /* end main */
```

The output of the above program is the same as the output of the **for** loop and **while** loop versions of this program. That is:

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

Goodbye!
```

If you want additional practice with do loops, please review program 4.9 on page 61 in your textbook. It is the same program as 4.8. The only difference is that

program 4.8 uses a **while** loop to solve the problem, and program 4.9 uses a **do** loop.

Notice the similarities between the 3 loops discussed, but also be aware of the differences. Traditionally, if we know exactly the number of times a loop is to be performed, use the *'for loop',* if the loop must be performed at least once for an indeterminate number of times, use '*do*', and if the loop is contingent upon a starting condition to be performed at all, use '*while*'.

This ends the discussion of the 3 different repetition structures in C. Is this really the end, or just the end of the beginning, or is there a sequel? Who knows what lurks ahead of us, only the shadow knows....

Note: We will be skipping the section in the textbook that discusses the "**break**" and "**continue**" statements for now. I feel that we are better off discussing these 2 statements next week, where we cover the Selection Structures (if, else, else if, and switch). Although **break** and **continue** statements are used in loops, they cannot be effectively used unless combined with loops *and* selection structures. This will be clearer as we actually start discussing these topics.

That's all for this week's lecture notes. Whew!

No rest for the weary though. It's time to get started on your 2nd programming assignment.