Program Looping with the "for" loop.

One fundamental and necessary property of a computer is its ability to repetitively execute a set of instructions. The challenge here is to repeat code WITHOUT using "goto" statements. (A no-no in today's programming!) As we continue with programming we will quickly find the need to have portions of our programs repeated several times. We do this even in "real life". For example, if I wanted my children to take 5 bites of their vegetables, I don't say to them:

Take 1 bite of your vegetables
Take 1 bite of your vegetables
Take 1 bite of your vegetables
Take 1 bite of your vegetables
Take 1 bite of your vegetables

Instead, I would say: Take 5 bites of your vegetables.

(Although I have been found to repeat myself quite often with my children. I'm sure you know what I mean: Come here. Come here. Come HERE. COME HERE!!!!) Okay. I'm alright. Back to business...

We want this same ability with our computer programs. That is, to repeat a single, simple instruction, or perhaps a group of instructions, without having to literally repeat the code in our program. How do we do this?

C provides 3 tools for repeating code in programs. The are the "Repetition Structures", more commonly referred to as "loops". The 3 looping structures in C are:

1. The **for** loop
2. The **while** loop
3. The **do** loop

Although any of the above 3 looping structures can be used to repeat code in C, it is a good idea to get to know the slight differences amongst them because programming ***efficiency*** can be greatly enhanced if you select the loop that is best suited for solving you problem. This section discusses the first of the 3 looping structures: the for loop.

The for Loop

The for loop allows the programmer to specify that an action (1 or more statements of code) is to be repeated a specified <u>number of times</u>. So, if you know that the loop is to be repeated exactly 5 times, for example, then the for loop is a good one to choose. Using the previous analogy with the vegetables, I would pick a ***for loop*** to solve the problem, because I know that I want my children to eat exactly **5** bites of their vegetables.

Let's look at a program which uses a for loop to output the numbers from 1 to 5 along with their squares. The output of the program will look like:

```
Hello, and welcome to the squares program!

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

Goodbye!
```

It would be easy enough to simply create 7 **printf** statements in our program to output these 7 lines, however, what if I wanted to output 100 squares? How about 1000? You will see how useful using repetition structures are when you realize that most computer programs will loop hundreds, thousands, even millions of times to process data!

Let's get back to the above problem. The code, using a for loop, would look as follows.

Note that the numbers shown to the left of some of the code are not actually part of the program; they are there to assist me in describing the code to you.

```
#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. */
      /* ------------------------------------------------------------------------------------------- */
1.    for (x = 1; x <= 5;  x = x + 1)
2.    {
3.         result = x * x;
4.         printf ("%i squared is %i\n",  x, result);
5.    } /*  end for loop  */

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

      printf ("\nGoodbye!\n");

} /*end main   */
```

Let's take this one line at a time. The lines that are not numbered are lines that you have seen before. Variable declarations, as well as the initial and final greetings are nothing new. What is new here is the code at lines **1** - **5**.

```
Line 1:   for (x = 1;  x <= 5;   x = x + 1)
                    ^         ^              ^
                1st part    2nd part   3rd part
```

This is the beginning statement in a **for** loop. It is composed of the reserved word: for, then 3 segments (parts) of information, separated by semicolons, inside parenthesis. Notice NO SEMICOLON at the end of this initial **for** statement. More on that later. I will break down the 3 parts of the **for** loop statement and discuss each one separately.

1st part: ***The initial value*** (**x = 1**)

This segment of the for loop *sets the loop control variable to some initial value*. That is, in this example, I am using variable **x** as my **loop control variable**, and I am initializing it to **1**. The loop control variable (also known as a **loop counter)** controls the flow of the loop. (Note: Variable names **for** loop control variables do not have to be descriptive, and are commonly single character variable names.)

Since I want to process the numbers from 1 to 5, it makes sense to set this to this initial value 1. This portion of the **for** loop is executed only *once*, the first time through the loop. Depending on our problem to solve, we can set this initial value to any integer. For example, if I wanted to calculate and display the squares from 5 to 1 (decreasing order), it would make sense to set this initial value to 5. Remember, this initial value can be any value, as long as it is a whole number.

2nd part: ***The loop condition (or final value)*** (**x <= 5**)

This segment of the **for** loop *tests the loop control variable against some final value*. That is, in this example, we test that the loop control value, **x,** is "**less than**

or equal to 5". The **<=** operator is one of the 6 **Relational Operators** discussed in the next topic. As of now, you simply have to know that the test being performed in our example, using **<=** is as follows: is the loop control variable less-than or equal to the number 5? The loop will continue as long as this condition is true. However, as soon as this condition is no longer true, once variable x contains the value 5 in our example, the for loop will no longer execute. This portion (loop condition) of the for loop statement is executed each time through the loop.

3rd part: *The loop increment/decrement* (x = x + 1)

This final segment of the **for** loop *increments (or decrements) the loop control variable*. It is how the loop control variable gets from its initial value (**1**, in our example) to its final value (**5**, in our example). Here we add **1** to variable **x** each time through the loop. Without this portion, the loop control variable would never reach its final value, and the loop would execute forever (infinite loop). We could increment by 2, if we so choose (**x = x + 2**), or by whatever integer increment you choose. You can also decrement here, if for example, you need to process values in descending order. For example, if I wanted to display the squares for values between **5** and **1**, I would use the following for loop statement:

for (x = 5;   x >= 1; x = x - 1)

In the above statement, we initialize our loop control variable **x** to be the number **5**, we test that it is *greater or equal* to **1**, and then we **decrement** the loop control variable each time through the loop.

There are a variety of ways to represent the incrementing and decrementing of variables in C. Since incrementing and decrementing is so often performed, C has abbreviations for these sort of operations. These abbreviations will be discussed in a separate topic this week.

Code sequence plays a very important role when discussing loops. At this point it is difficult to understand when part 1, part 2, and part 3 are executed. Once we conclude our discussion of **lines 2 - 5** from the above program, I will discuss code sequence in more detail.

Lines 2 and 5:   { and }

As you probably guessed, these braces mark the **begin** and **end** of the block of code that make up the *body* of the **for** loop. Between these braces are the lines of code that will be repeated, through each iteration of the loop. Notice following the begin brace, I have indented the statements of code that make up the body of the loop. This is a good programming standard to follow, as it greatly improves readability. It is very consistent with the indentation that takes place between the begin and end braces of function main.

C allows you to leave the braces out if the body of the for loop consists of just a single line of code. For example, if you had a for loop that simply output: **Programming is fun!** 10 times, you could code it in either of the 2 ways below. The first method uses braces around the body of the loop. The second method does not use the braces. Either method is acceptable. Again, this is not a very practical program, but it helps to illustrate the concept.

First method (using braces):

```
void main (void)
{
    int   x;

    for (x = 1;  x <= 10;  x = x + 1)
    {
        printf ("Programming is fun!\n");
    }

} /*end main   */
```

Second method (no braces):

```
void main (void)
{
    int   x;

    for (x = 1;  x <= 10;  x = x + 1)
        printf ("Programming is fun!\n");

} /*end main   */
```

Both of these programs will do the exact same thing. One thing that I want to point out here is that even if the body is just 1 line of code, you should still **indent** that single line of code, as illustrated above.

```
Lines 3 and 4:  result = x * x;
                printf ("%i squared is %i\n", x, result);
```

These 2 lines make up the **body** of the **for** loop. The body of the **for** loop are the statements of code that will be repeated each iteration through the loop. These statements are also known as the **block** of code that makes up the **for** loop. In this particular example, **Line 3** uses the loop control variable **x**, and multiplies it by itself to get the squared value! Keep in mind that the value of variable **x** is **1** the first time through the loop. It is then incremented, and becomes **2** the second time through the loop. Then it is **3** the third time through the loop, and so on. This loop control variable **x** becomes a perfect variable for using within the loop for calculating the squares of the numbers **1** to **5**, since this loop control variable contains the desired number each time through the loop.

It is not uncommon for loops to use the loop control variable within the body of the loop as part of a calculation. The only thing you should avoid is modifying the value of the loop control variable within the body of a **for** loop. Recall, the value of the loop control variable is automatically updated each time through the loop in the 3rd segment of the **for** loop statement. You should not mess with it otherwise.

So, now we have the calculated value we want stored in variable **result**. **Line 4** is a simple **printf** statement that outputs the required information. Notice the loop control variable is used once again in this statement as part of the output!

At this time I will mention **code sequence**. The sequence of events is so important where loops are concerned. Another term used when referring to the sequence that the statements are executed in a loop is called: **Flow Control**.

Flow Control in a **for** loop:

1. The loop control variable (counter) is initialized (First segment of **for** loop statement).
2. The loop condition is evaluated (Second segment of **for** loop statement).
3. If the condition is **true**, the body of the loop is executed; the loop control variable is then incremented or decremented (3rd segment of **for** loop statement). Go back to **step 2**.
4. If the condition is **false**, the loop is terminated. Go to **step 5**.
5. Program execution continues with next executable statement following the loop.

Additional comments regarding the **for** loop:

- Initialization is only done once (step 1 above).
- The test is done before the body is executed (step 2 above).
- If the test is false initially (step 1 above), the body is not executed, thus *the body will not be executed at all*.

Additional Programming Examples:

I will provide a few additional programming examples here to be sure you understand loops fully. I am not too pleased with the programming example 5.2 on page 44 in the textbook, as I feel the triangular number algorithm on is a bit complex for explaining **for** loops, *although some people thoroughly enjoy it*. I think it is best to keep the algorithm simple while learning these concepts.

The program below is similar to the one we've worked on above (the square program), however it has been modified to prompt the user for the desired start and end numbers to be squared! This requires just a simple change, yet makes the program much more flexible.

The output of the program looks like:

```
Hello! This program will square a sequence of numbers for you.

Please enter a starting number: 3
Please enter an ending number: 10

3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100

Goodbye!
```

The code for the above program might look like:

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

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

        int     x,  result, initial_value, final_value; /* notice 2 new vars here */
        char  c;                                         /* for clearing input buffer */

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

        printf ("\nHello! This program will square a sequence of number for
you\n\n");

       /* Prompt user for starting and ending numbers. */
       /* -------------------------------------------------------------------- */
```

```
        printf ("Please enter a starting number: ");
        scanf ("%i", &initial_value);
        while ( (c = getchar() != '\n') && c != EOF);      /* clear input buffer  */

        printf ("Please enter an ending number: ");
        scanf ("%i", &final_value);
        while ( (c = getchar() != '\n') && c != EOF);      /* clear input buffer  */

        /* Calculate and output the squares for numbers entered by user.*/
        /* ------------------------------------------------------------------------------------------- */

        for (x = initial_value;  x <= final_value;  x = x + 1)
        {
            result = x * x;
            printf ("%i squared is %i\n",  x, result);
        } /*  end for loop  */

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

        printf ("\nGoodbye!\n");

} /*end main   */
```

Notice that the only real change that needed to be made to the **for** loop was in the **for** statement itself. Instead of setting the *initial value of **x*** to **1**, we set it to the *starting number* entered by the user. Also, instead of setting the *final value of **x*** (or loop condition) to **5**, we set it to the *ending number* entered by the user. This provides for a much more flexible program, as the user can enter any numbers (1 to 100, for example), and the program would process those numbers without any problem. The only thing you need to be concerned with here, is that if the user enters a number whose square would results in a value greater than 32,767 (the largest number that an integer can contain), it would not be able to be stored in variable **result**. So, you may want to make the data type for variable result to be of type **long** to be able to store the larger values. The other alternative is to test the numbers entered by the user to be sure they do not exceed a certain range, but we have not learned how to do that yet! :-)

One last comment to be made at this time. I know what your thinking: that "he always has **one last comment**". (Did I say I was psychic?!?!) Anyway, I wouldn't be able to sleep tonight if I neglected to tell you the following. And besides, I think you'll like it! :-)

The body of the for loop above contains 2 statements. I want you to be aware that you could *combine* the 2 statements into 1 statement! This has nothing to do with loops, but everything to do with **printf**. Recall that within a **printf** statement, when you have a format specifier, the **printf** function will look for a value to stuff into the spot in the **printf** string. The value that the **printf** outputs is usually located in a variable following the end of the **printf** string. Well, that variable *does not have to be a variable*. It could instead be a calculation! In other words, the **printf** statement would obtain the value needed, not from the contents of a variable, but instead from the results of a calculation.

So, in our example, where we have:

```
    result = x * x;
    printf ("%i squared is %i\n",  x, result);
```

We could have instead used

```
printf ("%i squared is %i\n", x,  x*x);
                                  ^
                                calculation performed inside the printf statement
```

Notice the calculation for the square (**x * x**) is performed directly *inside* the **printf** statement. This is legal, valid, and often done, ***even though I do not like this in a program***. In this example, if you did the calculation directly in the **printf** statement, then there would be no need for variable **result** at all! The body of the **for** loop would be whittled down to a single statement. Some people prefer this. Others, do not. C provides so many "shortcuts", and I believe that if too many are used, they can actually reduce the readability of the program. In this case, I would ***definitely*** keep the calculation separate from the **printf** statement.

The next section of notes discusses the 6 relational operators.