Loops

Often we want our program to repeat the same tasks multiple times.

We've seen our hardware kits have a place for code that is repeated for the life of the program. (ie. Blink on, off, on, off….)

Previously we learned about the *selection structure* where the program's flow we decided by statements like if..then.

In this section we will study **iteration structures** where looping is the main type.

Each loop passes through a group of statements called an **iteration**.

For example, a loop may iterate until a specified variable reaches the value 100.

While Loops

A while loop is a loop structure that will continue to go on forever until some condition INSIDE the loop causes it to stop.

For example, a while loop may be written to ask the user to input a series of numbers until the number 0 is entered. The loop would repeat until the number 0 is entered.

There are two types of while loop, the standard **while loop** and the **do while** loop.

The difference is where the control expression is tested.

While

The while loop repeats a statement or group of statements as long as the control expression is true.

In a while loop, the control expression is tested before the statements in

the loop begin.

In order for a while loop to come to an end, the statements in the loop must change a variable used in the control expression.

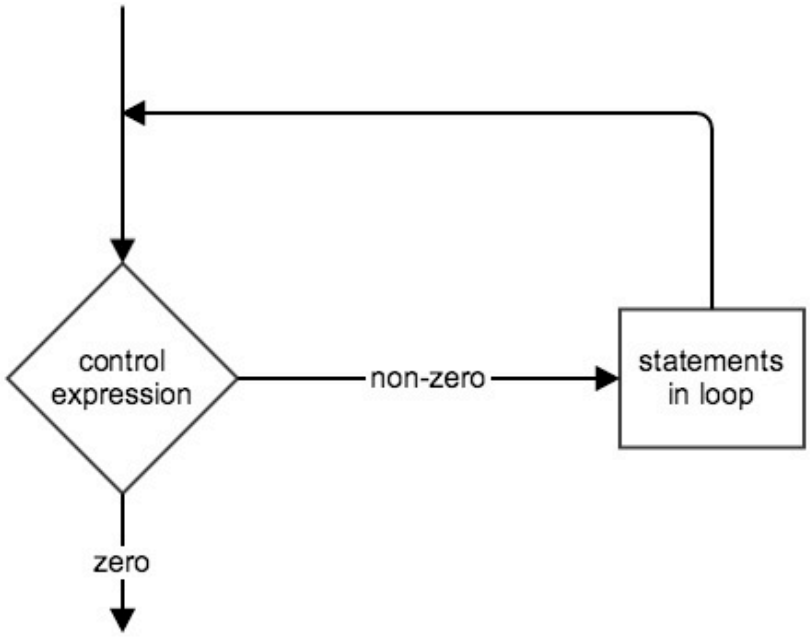
```
int main()
{
     float number = -1;
     float sum = 0;

     cout << "This program will sum numbers until the user
     inputs 0 to exit." << endl;

     while (number != 0)
     {
          cout << "Enter a number to add to the sum: ";
          cin >> number;
          sum += number;
          cout << endl << "The current SUM is: " << sum <<
          endl;
     }

return 0;
}
```

The flow chart for this process would look like:



The Do While Loop

The **do while loop** is very similar to the standard while loop except the statement or group of statements are repeated and the control expression is testing at the END of the loop.

Because the control expression is tested at the end of the loop, a do while loop is executed at least one time no matter what.

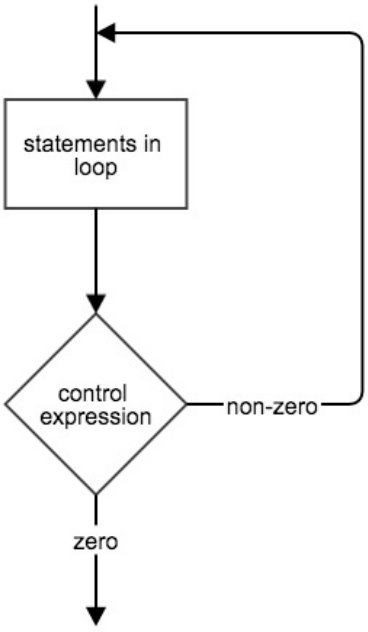
```
int main()
{

     float num;
     float squared;

     do
     {
          cout << "Enter a number (Enter 0 to quit): ";
          cin >> num;
          squared = pow(num,2);
          cout << num << " squared is " << squared << endl
          << endl;
     }
     while (num != 0);

return 0;
}
```

The flow chart for a do..while loop looks like:



Instead of having the user change a variable to stop a while or do..while loop we can also use a counter to have to loop execute a set number of times.

We do this by either **incrementing** or **decrementing** the variable that is part of the control expression.

```
int main()
{

     int j;
     j = 1;
     cout << "The loop is about to start!!" << endl;
     sleep(1);
     while (j <= 5)
     {
          cout << "The value of j is: " << j << endl;
          j ++;
          sleep(1);
     }
     cout << "The loop is done now!!" << endl;

return 0;
}
```

The for Loop

A more elegant way of having a loop repeat a fixed number of times is using the **for loop**.

It can be a little more difficult to read in the code than a while loop but it makes it is more compact when looping a fixed number of times.

Like an if statement, the for loop using parenthesis.

In parenthesis are three **parameters** which are needed to make the loop work.

```
for (initializing expression, control expression, step
expression)

{
     statements to execute
}
```

The first parameter is called the **initializing expression** and it initializes the counter variable to it's first value (often 0 or 1).

The second parameter is the condition to end the loop called the **control expression**.

As long as the control expression is true the loop will iterate.

The third parameter is the **step expression**.

It changes the counter variable, usually by adding or subtracting to it.

We often use the for loop to make the counter increase each time to run the loop a fixed amount of iterations.

```
int i;    //counter variable
for (i = 1; i <= 5; i++)
{
cout << i << endl;
}
```

We can also use the same technique but have the loop count down.

```
int i;    //counter variable
for (i = 10; i >= 0; i--)
{
cout << i << endl;
}
```

We can also get creative and have it count or change by whatever multiples we want

```
int i;    //counter variable
for (i = 0; i <= 100; i += 10)
{
cout << i << endl;
}
```

Nesting Loops

We've already nested if..then structures in previous sections and the same can be done for loops.

Loops within loops are actually a very common way of structuring the flow of a program.

It does present more of a challenge to trace a program that has nested loops however, but the results are worth it.

```
int i, j;
cout << "BEGIN" << endl;
for (i = 1; i <= 3; i++)
{
     cout << "Outer Loop: i = " << i << endl;
     for (j = 1; j <= 4; j++)
     {
          cout << "      Inner Loop: j = " << j <<
          endl;
     }
}
cout << "END" << endl;
```

Stopping in the Middle of a Loop

The keyword **break**, also used in switch statements, can be used to end a loop before the conditions of the control expression are met.

Once a break terminates a loop, the execution begins with the first statement following the loop.

This way of exiting a loop should be done cautiously. It is certainly not the best way to handle exiting a loop.

```
do
{
     cout << "Enter a number (Enter 0 to quit): ";
     cin >> num;
     if (num == 0.0)
     {break;}
     squared = pow(num,2);
```

```
        cout << num << " squared is " << squared << endl
        << endl;
    }
    while (1);
```

The **continue** statement is another way to stop a loop from completing each statement.

Instead of continuing with the first statement after the loop, the continue statement skips the remainder of the loop and starts the NEXT ITERATION of the loop.

```
int i;
for (i = 1; i <= 10; i++)
{
     if (i == 5)
     {continue;}
     cout << i << endl;
     sleep(1);
}
```