More on Conditions

This is a good time to talk about conditions being satisfied. Before, we looked at conditions within IF statements; let's look at the following example:

```
int m;

m = 0;

if  (m == 0)
{
    fprintf(stdout,"m is equal to 0\n");
}
else if (m != 0)
{
   fprintf(stdout,"m is NOT equal to 0\n");
}
```

In this example, if integer **m** is equal to 0, the code prints "m is equal to 0", and if integer **m** is not equal to 0, the code prints "m is NOT equal to 0". What is really happening is this: When the code reaches a condition, it quickly analyzes it and replaces it with either a 1 or a 0. If the condition is TRUE, it replaces it with a 1, and if it is FASLE, it replaces it with a 0.

In our example, m is set equal to zero, so the code sees the condition **m == 0** and analyzes it. This condition asks if m is equal to 0. Because **m** is indeed equal to zero, the condition is TRUE, so it replaces the condition with a **1**. Now, when the code sees the next condition **m !=0**, it checks that condition. That condition asks whether m is not equal to zero. Well, because **m** is indeed equal to zero, the condition is FALSE, so it is replaced by a **0**.

So, here is what the code really sees, after the conditions have been checked:

```
int m;

m = 0;

if  (1)
{
    fprintf(stdout,"m is equal to 0\n");
}
else if (0)
{
   fprintf(stdout,"m is NOT equal to 0\n");
}
```

In other words, the code sees conditions as being either TRUE (represented by a 1) or FALSE (represented by a 0). If it is TRUE, then it is satisfied, and if it is FALSE, it is

unsatisfied. **In actuality, any non-zero integer represents TRUE and only integers that are of value 0 are FALSE.**

The following code should show you how this works.

code11.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(void)
{

int pp;

// the following line should NEVER print
if (0) fprintf(stdout,"This condition should never be satisfied\n");

// the following lines WILL ALWAYS print
if (1) fprintf(stdout,"Yep, this condition is satisfied\n");
if (5) fprintf(stdout,"Yep, this condition is satisfied\n");
if (-1) fprintf(stdout,"Yep, this condition is satisfied\n");
if (-100) fprintf(stdout,"Yep, this condition is satisfied\n");


// Lets run a little loop to test this.
// We will loop through pp from -4 to +4, and put its
// value into the conditional.  Note that it will print
// every time except for when pp is equal to 0.

for (pp= -4; pp <= 4; pp++)
{
   if (pp)
   {
      fprintf(stdout,"pp is value: %d\n", pp);
   }
}

// Now lets try the same, but lets change the conditional
// slightly by adding an !, which means NOT.
// I'll let you figure this one out.

fprintf(stdout,"Ok, now we try the opposite\n");
for (pp= -4; pp <= 4; pp++)
{
   if (!pp)
   {
      fprintf(stdout,"pp is value: %d\n", pp);
   }
}
```

```
// Notice that you can set integers to be equal to the outcome of
// a conditional. Look at the following examples.

fprintf(stdout,"lets try something new\n");

// In this example, pp will be set to 1 because the condition is true
pp = (4>3);
fprintf(stdout,"the value of pp is: %d\n", pp);
if (pp) fprintf(stdout,"Yep, its true that 4 is bigger than 3\n");


// In this example, pp will be set to 0 because the condition is false.
// Therefore, the condition should never be satisfied.
pp = (3>4);
fprintf(stdout,"the value of pp is: %d\n", pp);
if (pp) fprintf(stdout,"THIS SHOULD NEVER PRINT\n");

return 0;
}
```

Note in the above code, we could set an integer to a condition. Every condition is evaluated as an integer which is either 1 for TRUE or 0 for FALSE. Never try to set a float or a double to a condition! I have no idea what kind of trouble would await you.

If it is not clear how this works, then perhaps you should try experimenting with some of your own examples.

While Loops

Another type of loop is a **while** loop, which can be very useful if you need to loop over something, yet you don't know ahead of time how many times to loop. The basic structure of a **while** loop is as follows:

```
while (some_condition)
{
        Whatever code you want
}
```

As long as *some_condition* is satisfied, the loop will continue. From the previous section, it should be clear that as long as *some_condition* is a non-zero integer, the loop will also continue.

The following code will introduce you to **while** loops.

code12.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(void)
{

int pp;


// Here, we set integer pp to -10 before we enter the loop.
// The loop will continue to run as long as pp is less than 5.
// Within the loop, we print out the value of pp and increment
// its value by +2.
// Eventually, the value of pp will be greater than or equal to
// 5, so the while loop will end.

   pp = -10;
   while ( pp < 5)
   {
     fprintf(stdout,"Value of pp: %d\n",pp);
     pp=pp+2;
   }

return 0;
}
```

The output should be this:

```
Value of pp: -10
Value of pp: -8
Value of pp: -6
Value of pp: -4
Value of pp: -2
Value of pp: 0
Value of pp: 2
Value of pp: 4
```

Oftentimes, in my experience at least, it is convenient to use an integer to act as a flag to stop the loop. In the next example, we increase the radius of a sphere until the sphere's volume reaches some maximum value. This example will also introduce us to a variable set in the math library, *math.h*, called **M_PI**. which is simply the value of pi.

Carefully read this code, and make sure you follow its logic, particularly how the integer variable *keep_going* controls whether the loop continues. Once a condition is reached, we set *keep_going* to 0, which stops the next loop from starting.

code13.c

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

int main(void)
{

// We'll use an integer to flag the loop to stop at
// a certain point.  I like to name these integers
// meaningful names such as "keep_going".
// When keep_going becomes equal to zero, the loop will
// stop.

// Note that we use the variable M_PI in the loop.  This
// is a variable from the math library, math.h.  It represents
// the value of pi.

int keep_going;

double radius;
double volume_of_sphere;
double maximum_volume = 10.0;

    // set keep_going to 1 to keep the loop going
    keep_going = 1;

    // set the intial radius
    radius = 0.0;

    // In this loop, we will increase the radius
    // until the volume exceeds the maximum volume.
```

```
    while (keep_going)
    {
        volume_of_sphere = 4.0/3.0*M_PI*radius*radius*radius;

        // If the sphere volume is greater than
        // the maximum volume that we prescribe,
        // then we will set keep_going to 0.  This
        // will stop the loop from starting again!
        if (volume_of_sphere >= maximum_volume)
        {
          keep_going = 0;
        }

        // Print some information here
        fprintf(stdout,"radius: %f, volume: %f\n", radius, volume_of_sphere);

        // Now, lets make radius a little bigger for its next
        // journey within the loop.

        radius = radius + 0.1;
    }

   // Lets brag about getting out of the loop.
   fprintf(stdout,"I made it out of that loop\n");

return 0;
}
```

You should get the following output:

```
radius: 0.000000, volume: 0.000000
radius: 0.100000, volume: 0.004189
radius: 0.200000, volume: 0.033510
radius: 0.300000, volume: 0.113097
radius: 0.400000, volume: 0.268083
radius: 0.500000, volume: 0.523599
radius: 0.600000, volume: 0.904779
radius: 0.700000, volume: 1.436755
radius: 0.800000, volume: 2.144661
radius: 0.900000, volume: 3.053628
radius: 1.000000, volume: 4.188790
radius: 1.100000, volume: 5.575280
radius: 1.200000, volume: 7.238229
radius: 1.300000, volume: 9.202772
radius: 1.400000, volume: 11.494040
I made it out of that loop
```