Conditionals and Loops

Conditionals in C

The most basic conditional in C is the if statement. An example of one is shown in the code below.

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

/**
 * main.c  */
void main(void)
{
    int id = 1;
    int myId = 0;

    if (id == 1)
        myId = id;
    return;
}
```

Notice the use of the == operator to check to see if two values are equal. This is different from the assignment operator =. Beware of using one when you want to use the other!

The allowed Boolean operators in the if statement for C are:

>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
==	equal to
!=	not equal to

As in some programming languages, an optional else keyword is also available, as is shown here:

```
/**
 * main.c
 */
void main(void)
{
    int id = 1;
    int myId = 0;

    if (id == 1)
        myId = id;
    else
        myId = 2;
```

```
   return;
}
```

Often there are times when you want more than one line of code to be part of an if statement or an else. You then use the C scope operator that you've seen before {}. If you don't use {} then only the first statement that follows the if or the else will be executed! Here is some example code:

```
/**
 * main.c  */
void main(void)
{
    int id = 1;
    int myId = 0;
    int myId1 = 0;

    if (id == 1)
    {
        myId = id;
        myId1 = id;
    }
    else
    {
        myId = 2;
        myId1 = 1;
    }

    return;
}
```

You can also have an if statement that contains more than one true or false clause as you can in Python. Here is an example:

```
/**
 * main.c
 */
void main(void)
{
    int id = 1;
    int myId = 0;
    int myId1 = 0;

    if (id == 1 && myId == 0 && myId1 == 0)
    {
        myId = id;
        myId1 = id;
    }
    else
    {
        myId = 2;
```

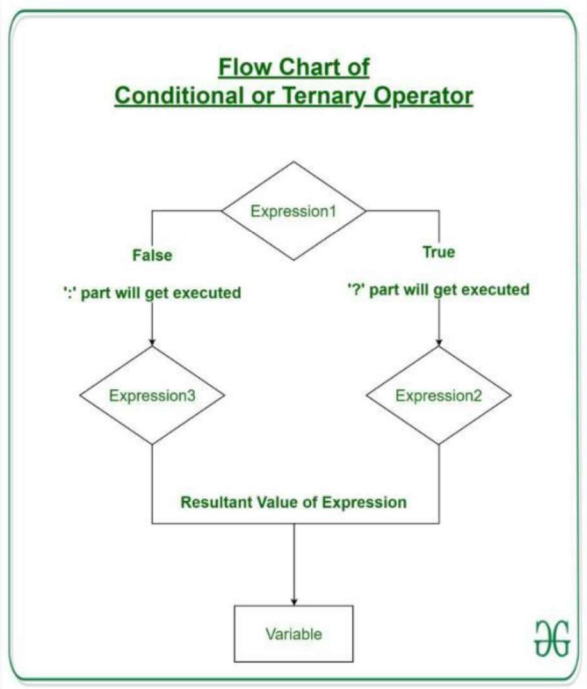
```
        myId1 = 1;
    }

    return;
}
```

The && is equivalent to the AND operation of the two expressions. The || is the equivalent to the OR operation. Conditional statements, like *if*, can be nested in C just like they could in Python.

Finally, there is also a unique way to write a special if statement in C so that it takes up only one line of code. It is called the Ternary Operator. Here is some code that illustrates using the ternary operator:

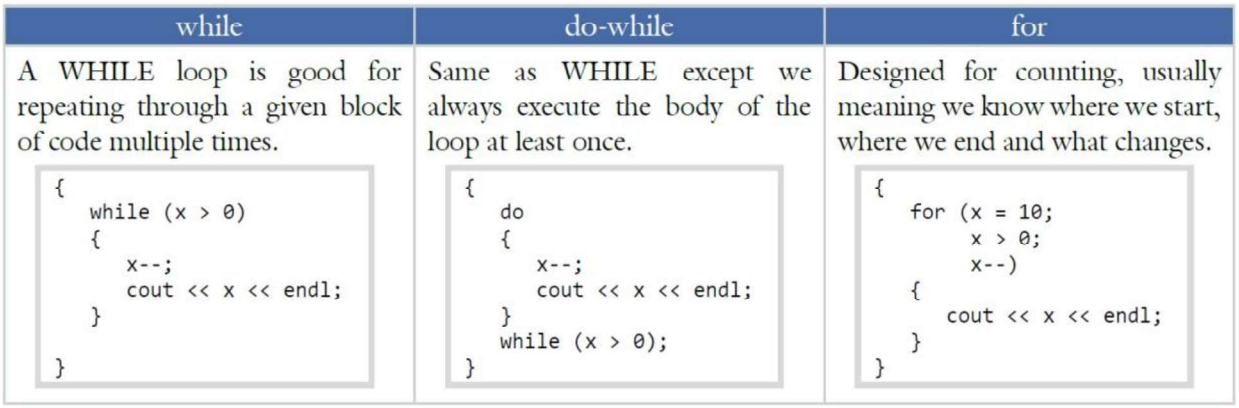
```
/**
 * main.c  */
void main(void)
{
    int id = 1;
    int myId = 2;
    int myId1 = 0;

    myId1 = (id > myId) ? id : myId;
    return;
}
```

The ternary operator has three parts. The first is the logical check followed by the *?* part of the operator. The second is the value to assign the variable *myId1* if the check returns true. This is followed by the separator operator, *:*, the third part is the value assigned to the variable, *myId1,* if the check is false Here is a diagram that shows the execution flow of the ternary statement:



Loops in C

As in most computer languages, C comes with three types of loops: the for loop, the while loop, and the do-while loop. Here is an overview of the three loops:



while

A WHILE loop is good for repeating through a given block of code multiple times.

```
{
   while (x > 0)
   {
      x--;
      cout << x << endl;
   }

}
```

do-while

Same as WHILE except we always execute the body of the loop at least once.

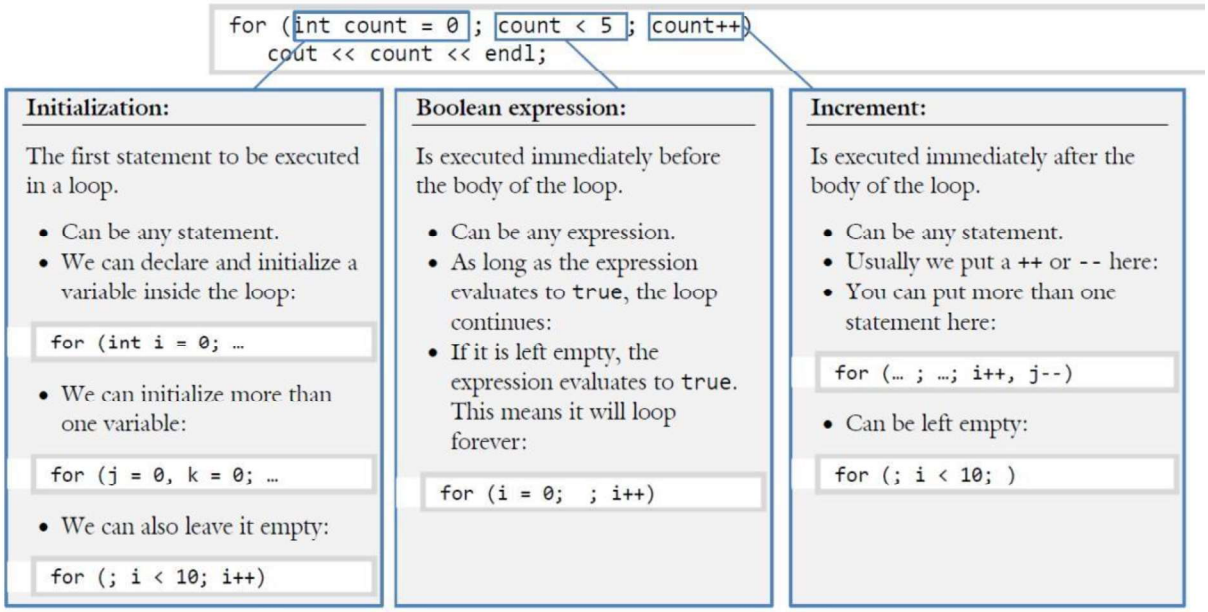
```
{
   do
   {
      x--;
      cout << x << endl;
   }
   while (x > 0);
}
```

for

Designed for counting, usually meaning we know where we start, where we end and what changes.

```
{
   for (x = 10;
        x > 0;
        x--)
   {
      cout << x << endl;
   }
}
```

The for loop syntax structure is fairly simple:



Here is a code example:

```
/**
 * main.c  */
void main(void)
{
    int i;
```

```
    int value = 0;

    for (i = 0; i < 5; i++)
        value = i;
    return;
}
```

The *while* loop and the *do-while* loop are related. They use a statement to decide whether to execute the loop.

The simplest and safest of these two is the *while* statement. The *while* loop will continue executing the body of the loop until the controlling boolean expression evaluates to false just as in Python. The syntax is:

```
while (<boolean expression>)
  <body code line>;
```

As with the *if* statement, we can always have more than one statement in the body of the loop by adding curly braces {}:

```
while (<boolean expression>)
{
    <body code line 1>;
    <body code line 2>;
     ...
}
```

The *do-while* loop is the same as the *while* loop except the controlling boolean expression is checked after the body of the loop is executed. This is why using this type of loop is dangerous. It's like running around with a blindfold on knowing there is a cliff nearby. Be VERY careful is you choose to use this loop. It's known to cause code crashes and code security issues.

As with the *while* statement, the loop will continue until the controlling boolean expression evaluates false. The syntax is:

```
do
  <body line of code>;
while (<boolean expression>);
```

Do-while loops are used far less frequently than the *while* loops. Those scenarios when the *do-while* loop would be the tool of choice center around the need to ensure the body of the loop gets executed at least once, though there are safer ways to do this using the *while* loop.

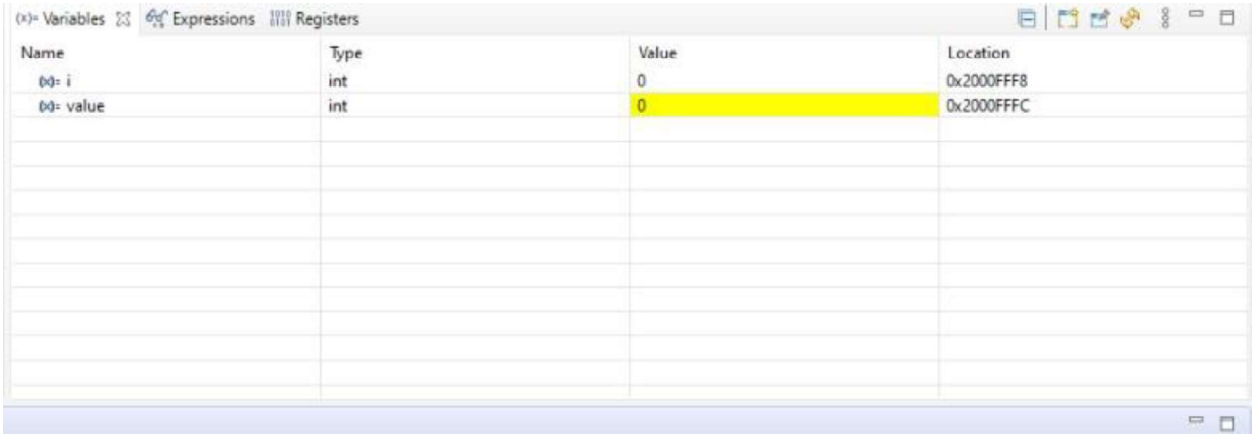
Debugging Loops using the Eclipse IDE

One of the nice features of Integrated Development Environments is the ability to use stepwise debugging, as you've seen before. Stepwise debugging is particularly helpful in trying to find defects caused by bad code in loops. Here is an example. Type it into this week's project.

```
/**
 * main.c  */
void main(void)
{
    int i;
    int value = 0;

    for (i = 0; i < 5; i++)
        value = i;
    return;
}
```

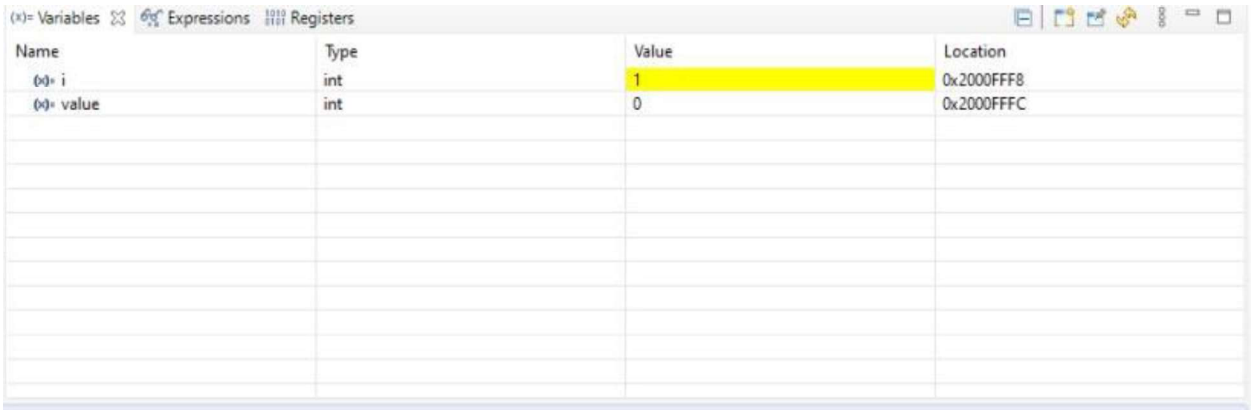
This is a fairly simple loop, but let's assume that for some reason I thought the loop would start with *i* being 1 and going until *i* is 5. Let's stepwise debug the code.

We're going to use the Step Into command (you can use F5 to accomplish this as well) to step through the program and watch the variables. Just before the loop starts, the variables have these values:



This makes sense; the loop hasn't started. Now press the F5 key, and step into the loop. Unfortunately, the variable values still stay the same due to a code defect. Ah… the loop doesn't start with the value 1, it starts with the value 0. To make the loop start at one you would need to change the initializer to be i = 1;

Having made a code change, recompile and re-run the code. Enter F5 again until you are in the loop again, and then once more. You should see this:



The value of i has now change to 1. As you step through the loop, the value stored in *i* will change to 1, then to 2 and so on.

When you get to i = 5, notice the loop will no longer continue, but will exit. Ah… the other defect. The *i* variable will never be 5 as you intended. To make this happen you'll need to change the Boolean expression that controls the loop to include the value 5 by setting the comparison to be *i <= 5* or *i < 6*. The second is preferred since its intent is clearer than the first.