

Conditionals and Loops

Getting Started:

As you have done each week, download the zip file InitialProject.zip file, import it into Code Composer, compile it, and run it in the debugger on the MSP432P401R. You are now ready to add the code for this week's learning.

Last week, you learned the basics of structures and pointers. This lab will introduce you to some basic concepts that you will recognize conceptually; 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 "msp.h"

/**
 * main.c
 */
void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    int id = 1;
    int myId = 0;

    if (id == 1)
        myId = id;

    return;
}
```

Notice the use of the `==` sign to check to see if two values are equal. We cannot use the `=` sign, it has already been used as the assignment operator.

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:

```
#include "msp.h"
```

```
/**
```

```
 * main.c
```

```
 */
```

```
void main(void)
```

```
{
```

```
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer
```

```
    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 {}. Here is some example code:

```
#include "msp.h"
```

```
/**
```

```
 * main.c
```

```
 */
```

```
void main(void)
```

```
{
```

```
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer
```

```
    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:

```
#include "msp.h"

/**
 * main.c
 */
void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    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:

```
#include "msp.h"

/**
 * main.c
 */
void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    int id = 1;
    int myId = 2;
```

```

int myId1 = 0;

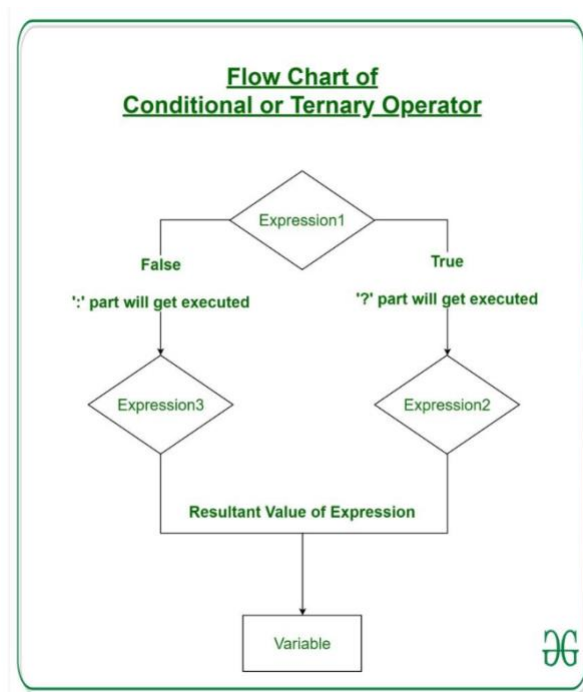
myId1 = (id > myId) ? id: myId;

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:

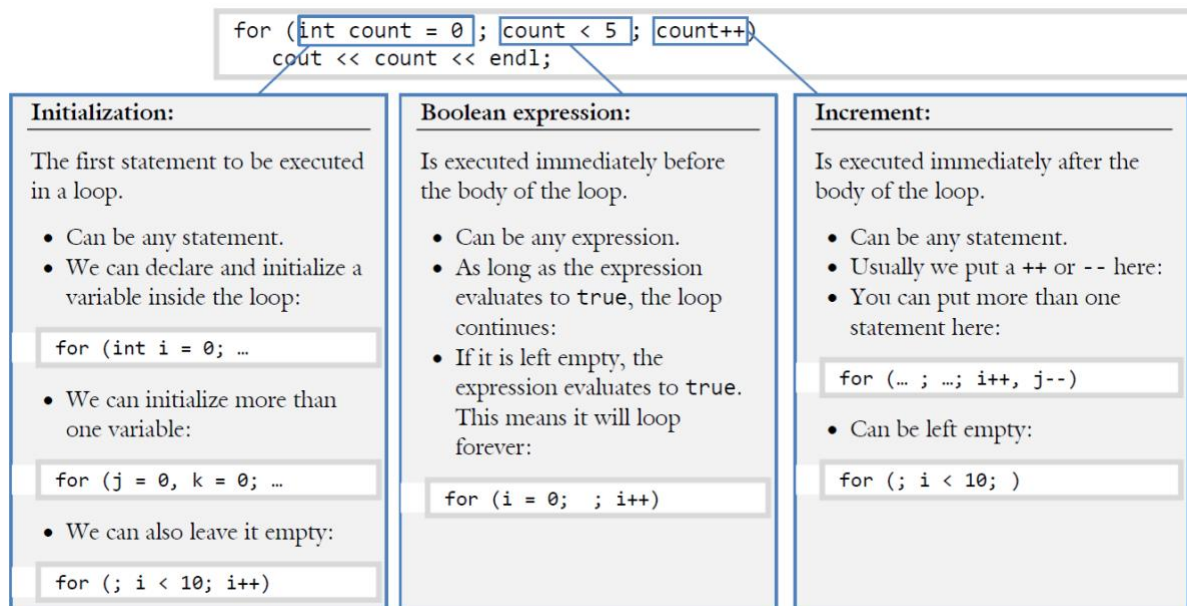


Loops in C

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	do-while	for
A WHILE loop is good for repeating through a given block of code multiple times.	Same as WHILE except we always execute the body of the loop at least once.	Designed for counting, usually meaning we know where we start, where we end and what changes.
<pre>{ while (x > 0) { x--; cout << x << endl; } }</pre>	<pre>{ do { x--; cout << x << endl; } while (x > 0); }</pre>	<pre>{ for (x = 10; x > 0; x--) { cout << x << endl; } }</pre>

The for loop syntax structure is fairly simple:



Here is a code example:

```
#include "msp.h"
```

```
/**
```

```
 * main.c
```

```
 */
```

```
void main(void)
```

```
{
```

```
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer
```

```
    int i;
```

```
    int value = 0;
```

```
    for (i = 0; i < 5; i++)
```

```

        value = i;

    return;

}

```

The While loop, and the do-While loop, are both related. They use a statement to decide whether or not 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 {}s:

```

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 near by. Be VERY careful if 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 to false.

The syntax is:

```

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

```

DO-WHILE loops are used far less frequently than the aforementioned 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 Code Composer IDE

One of the nice features of Code Composer and other IDE's 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.

```

#include "msp.h"

```

```

/**
 * main.c

```

```

*/
void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;    // stop watchdog timer

    int i;
    int value = 0;

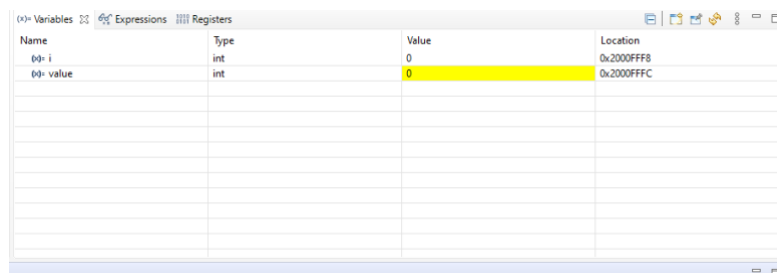
    for (i = 0; i < 5; i++)
        value = i;

    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. Lets stepwise debug the code.

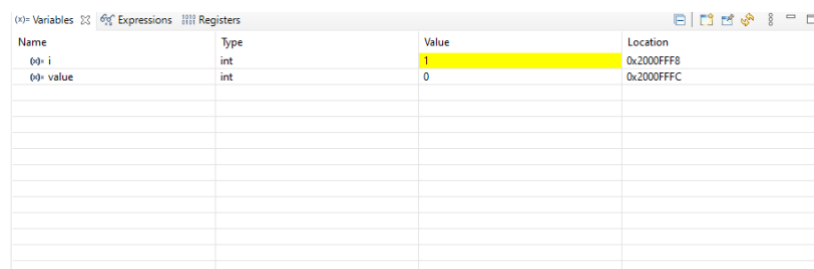
We're going to use the Step Into command (you can use F5 to accomplish this as well) to step through the program and watch the variables. Just before the loop starts, the variables have these values:



Name	Type	Value	Location
i	int	0	0x2000FFFB
value	int	0	0x2000FFFC

This makes sense, the loop hasn't started. Now press the F5 key, and step into the loop. Unfortunately, the variables 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:



Name	Type	Value	Location
i	int	1	0x2000FFFB
value	int	0	0x2000FFFC

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.

