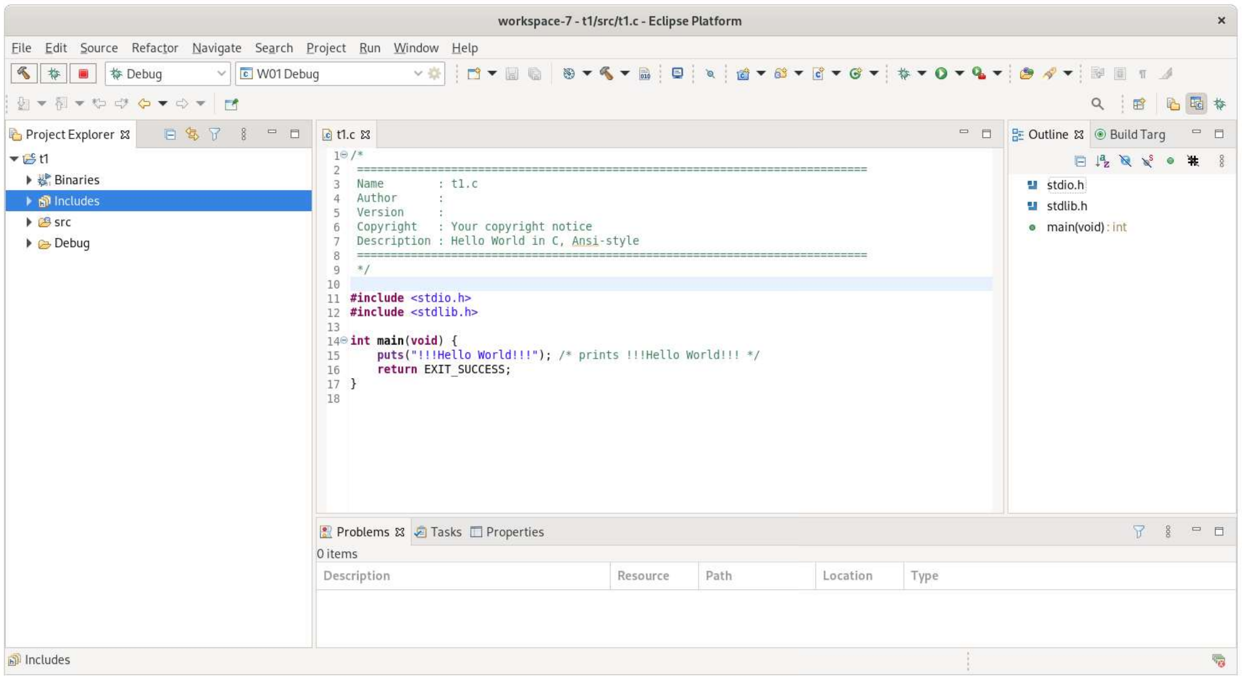
Functions and Variables

Simple Functions and Local Variables

Go ahead and start a new project and/or a new workspace for this week's tasks. If you create a new blank ANSI C project, your code should look like this:



```
1  /*
2   ============================================================================
3   Name        : t1.c
4   Author      :
5   Version     :
6   Copyright   : Your copyright notice
7   Description : Hello World in C, Ansi-style
8   ============================================================================
9   */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13
14 int main(void) {
15     puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
16     return EXIT_SUCCESS;
17 }
18
```

As noted in the last exercise, your code will start execution with the main function. All functions in C use the same pattern. When declaring a function, you need to first put the type of data the function returns, then the function's name, then the parameters that will be passed into the function. You can use *void* to indicate a function that returns no value or takes no parameters.

Notice line 11 in the code,

#include <stdio.h>

This is a compiler directive to include the system header file stdio.h and copy the entire contents of this file into your own file before compiling. This is a C version of python's *import* keyword and behavior.

You're going to add a function to main.c. Modify main.c to match this code example:

```
void test(void);

/**
 * main.c  */
void main(void)
{
    test();
    return;

}

void test(void) {
    int test1 = 0;
    test1 = 4;
    return;
}
```

Now let’s look at what the code does.

First, the statement:

void test(void);

is called a prototype of the test() function. While the execution of the program will start at main(), the compiler program, the program that translates the program to machine code, starts at the top of the file and works through the file. The prototype statement tells the compiler that at some point in the file you will be creating a function called test() somewhere in the file, so you can use the function name anywhere in the file.

In the main function you’ll notice this statement:

test();

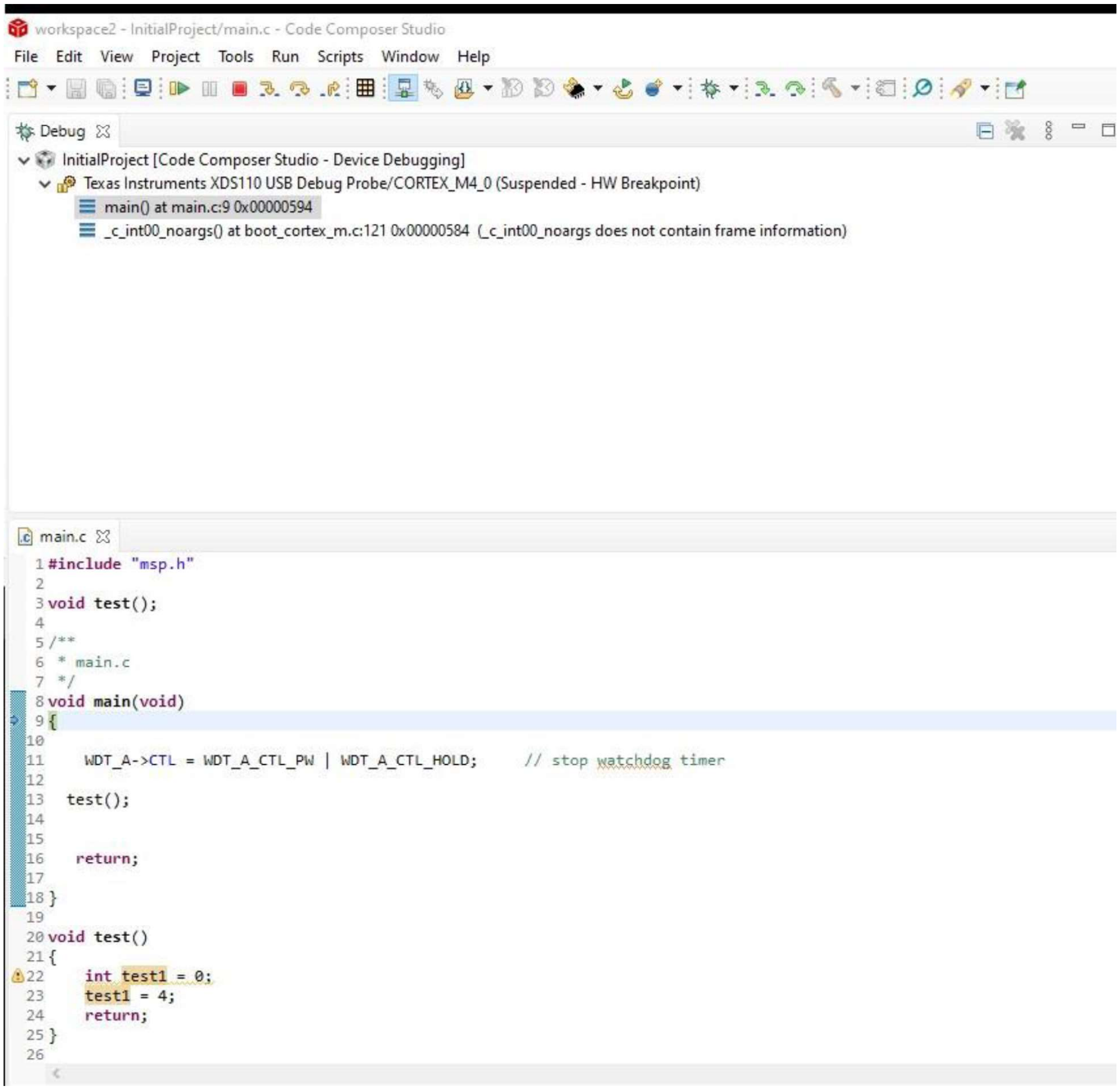
When this command is reached the execution of the code will go to the test()_function and execute the commands there.

Now, below the main function, the test() function is defined:

```
void test(void) {
    int test1 = 0;
    test1 = 4;
    return;
}
```

The void before the test() function name means that the function will return no data. The empty parenthesis () means that no data will be passed to the function. Build the program and then select Debug As… -> Code Composer Debug Session.

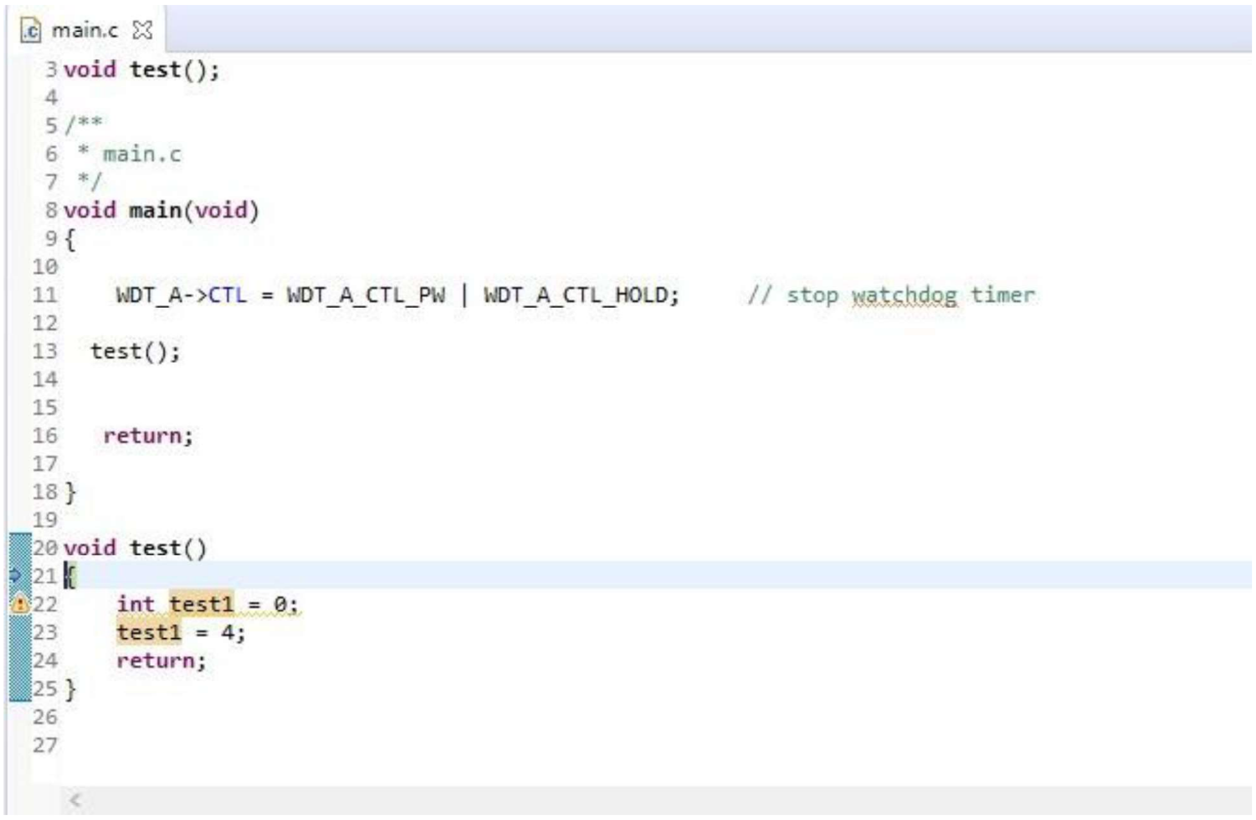
You should see this:


workspace2 - InitialProject/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
main.c

```
InitialProject [Code Composer Studio - Device Debugging]
  Texas Instruments XDS110 USB Debug Probe/CORTEX_M4_0 (Suspended - HW Breakpoint)
    main() at main.c:9 0x00000594
    _c_int00_noargs() at boot_cortex_m.c:121 0x00000584  (_c_int00_noargs does not contain frame information)
```

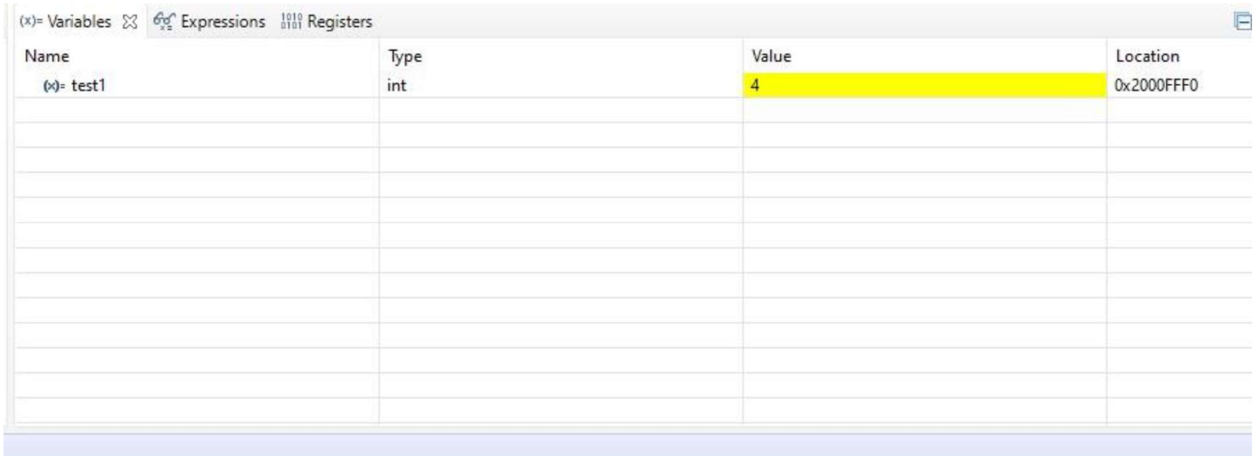
```c
#include "msp.h"

void test();

/**
 * main.c
 */
void main(void)
{

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

  test();


   return;

}

void test()
{
    int test1 = 0;
    test1 = 4;
    return;
}

```

Not select the Run->Step Over (F6) selection twice. Then select the Run->Step Into (F5) selection. You should now step into the *test()* function.


main.c

```c
void test();

/**
 * main.c
 */
void main(void)
{

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

  test();


   return;
}

void test()
{
    int test1 = 0;
    test1 = 4;
    return;
}
```

Select Run->Step Over 3 times. You should now be at the end of the function. You can see that the variable1 has changed value.



Name	Type	Value	Location
(x)= test1	int	4	0x2000FFF0

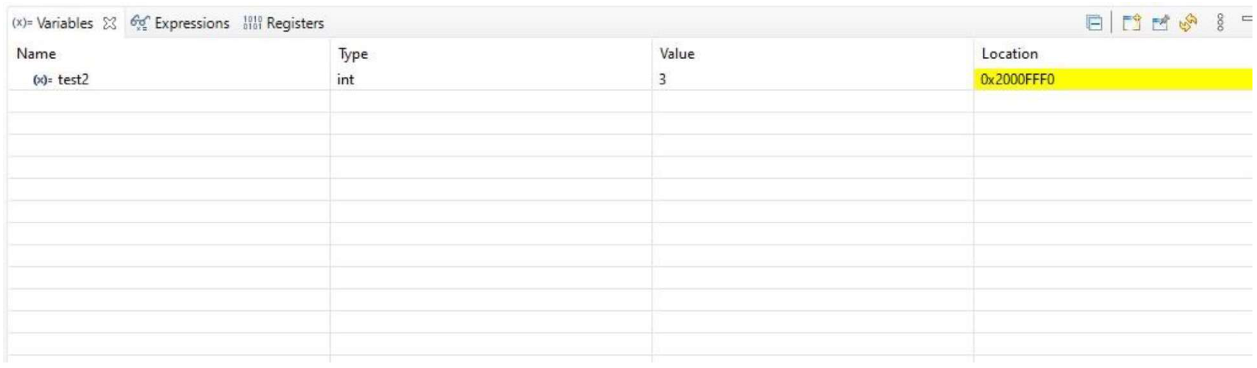
Select Run->Step Over again and you will leave the function and return to the main() function.

Just as in Python, you can write functions that have parameters and returns data Let's start with a simple example function that has a one parameter of type *int*. Change main to look like this:

```c
void test(int test2);
```

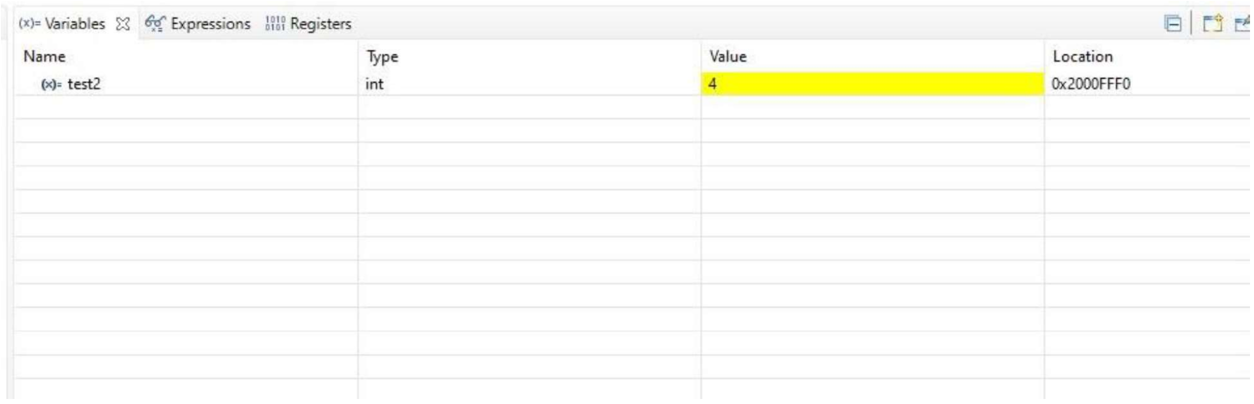
```c
/**
 * main.c  */
void main(void)
{
    int test1 = 0;
    test1 = 3;
    test(test1);

    return;
}

void test(int test2)
{
    test2 = 4;
    return;
}
```

Compile, debug, and run the program. Then use step-wise debugging and step into *test()* function. Notice that the value of *test2* is 3. This is a copy of the variable value *test1* used in the main() function.



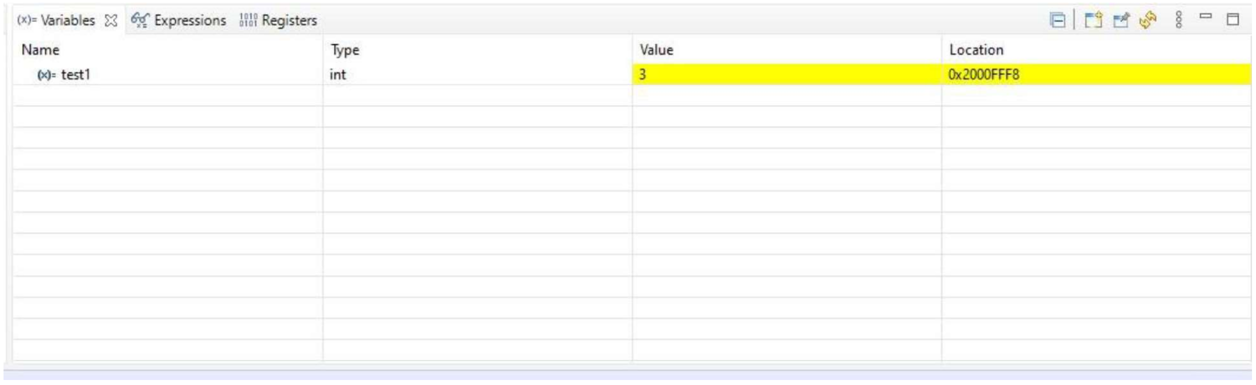
Name	Type	Value	Location
(x)= test2	int	3	0x2000FFF0

It is very important to understand that the storage location test1 and test2 are at two different locations.
Even if these two variables had the same name, they still would not share the same address. Now step through the function and you'll notice *variable1*'s value changes to 4:



Name	Type	Value	Location
(x)= test2	int	4	0x2000FFF0

Keep stepping until you return to main(). You'll notice that the value of *test1* is still 3.



Name	Type	Value	Location
(x)= test1	int	3	0x2000FFF8

It is important to note that even if these variables had the same name they wouldn't have the same address. Each time a function is called the variables in the argument list inside of the parenthesis are given their own memory addresses and the values are initialized using the values passed to them by the calling statement.

The last think we need to cover with respect to variables and functions is how to return data. C does this by declaring the type of the returned value as you read above., and then using the return keyword in the function just as you did in Python. Modify main.c to match the code below.

```
int test(int test2);

/**
 * main.c  */
void main(void)
{
    int test1 = 0;
    test1 = 3;

    test1 = test(test1);
    return;
}
```

```
int test(int test2)
{
    test2 = 4;
    return test2;
}
```

Notice that you have changed the test() function declaration by declaring the return type to be *int*.

int test(int test1);

This tells the compiler that you will be returning an integer from your function. Now in the calling statement you can use the assignment operator, =, to assign the value that returned from the function to the variable named test1. The memory assigned to contain test1's value will be updated.

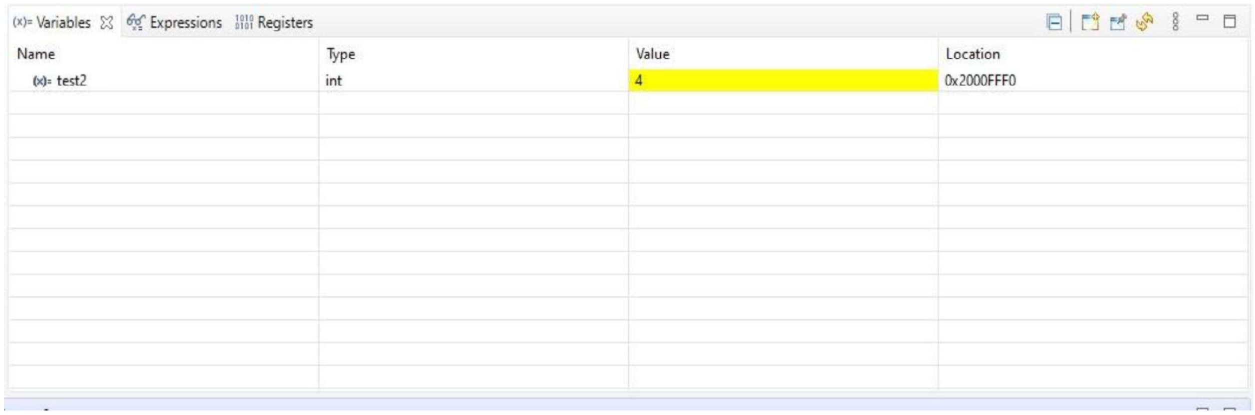
test1 = test(test1);

The final step is to change the function itself and add to the return statement the value you wish to return, in this case it is whatever is stored in the location named *test2*.

return test2;

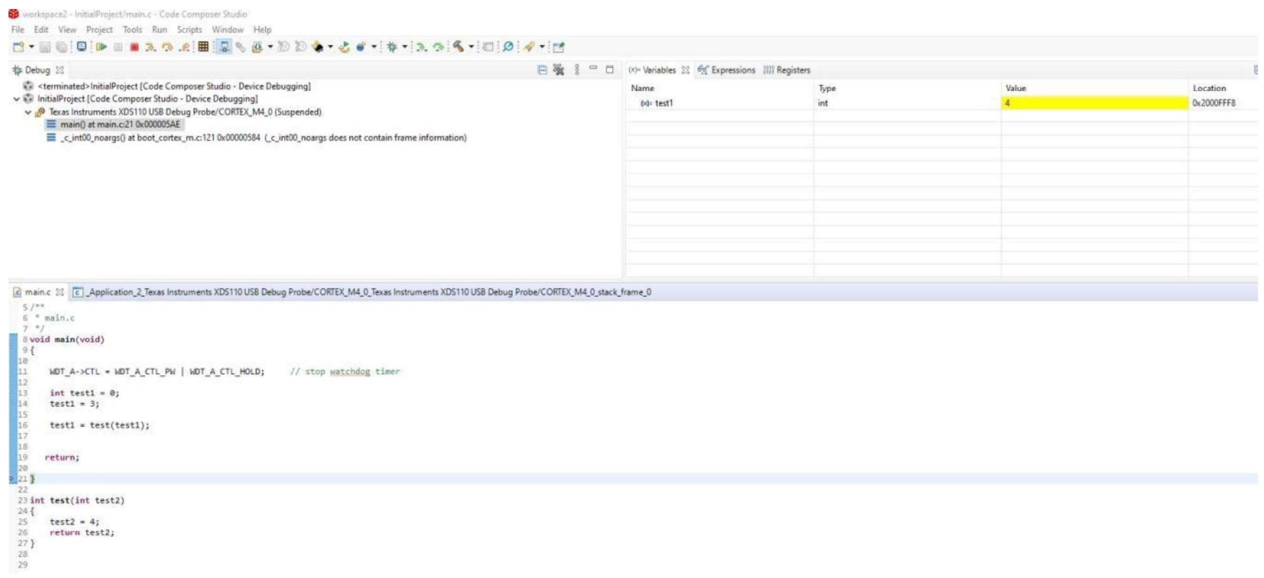
Save main.c then compile, run, and debug this code.

Now step through the code. When you get to the return statement of the function you should see the test21 value in the debug window:



Name	Type	Value	Location
(x)= test2	int	4	0x2000FFF0

Now step until you reach the end of the main function. Notice how the value of the memory location named test1 has changed to 4 based on the return from the function.


workspace2 - InitialProject/main.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
Variables
Expressions
Registers

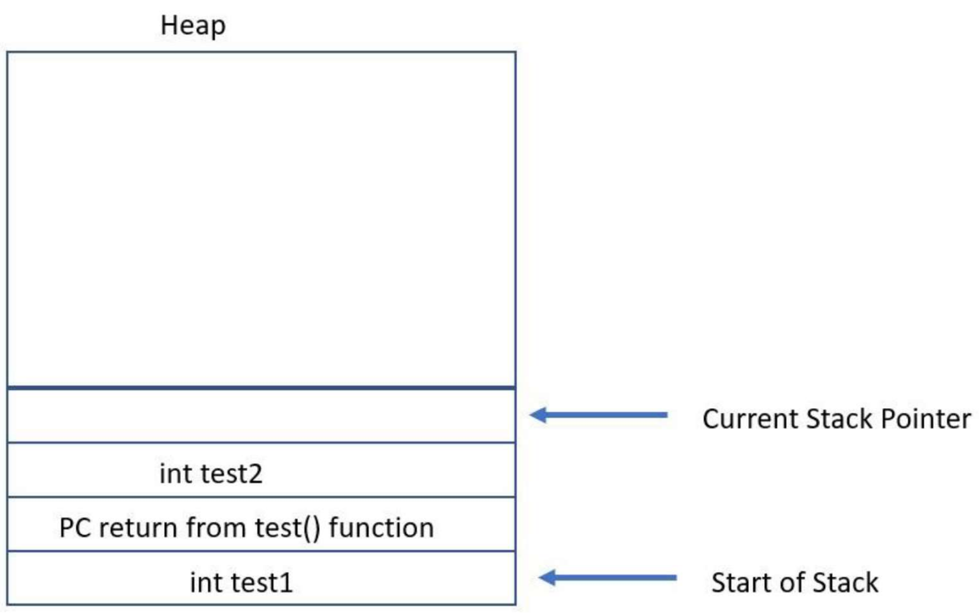
```
<terminated>InitialProject [Code Composer Studio - Device Debugging]
InitialProject [Code Composer Studio - Device Debugging]
  Texas Instruments XDS110 USB Debug Probe/CORTEX_M4_0 (Suspended)
    main() at main.c:21 0x000005AE
    _c_int00_noargs() at boot_cortex_m.c:121 0x00000584  (_c_int00_noargs does not contain frame information)
```

Name	Type	Value	Location
test1	int	4	0x2000FFF8

main.c _Application_2_Texas Instruments XDS110 USB Debug Probe/CORTEX_M4_0_Texas Instruments XDS110 USB Debug Probe/CORTEX_M4_0_stack_frame_0

```
 5 /**
 6  * main.c
 7  */
 8 void main(void)
 9 {
10
11     WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer
12
13     int test1 = 0;
14     test1 = 3;
15
16     test1 = test(test1);
17
18
19    return;
20
21 }
22
23 int test(int test2)
24 {
25     test2 = 4;
26     return test2;
27 }
28
29
```

Now a couple of details for using local variables. Normally a computer system has two types of memory. One is the program memory space; this memory holds the actual instructions that your CPU will execute. The other is data memory space, this memory holds values that you will use in your program. The data memory space starts as one contiguous space of memory. The system then uses this memory space in several ways. It holds dynamically allocated variables, the space available to these dynamically allocated variables is called the Heap. The compiler uses a data context called a stack to organize a part of this space. The stack holds several pieces of information. One of the items it holds is the history of the functions called up until the current command. This is called the call stack. It also holds all the local variables. For this example, at the end of the test_function (line 19) it would look like this:



You can see the local variables. The PC return is the Program Counter address that the program needs to return to the proper point in program execution when the test_function is complete.

Global Variables

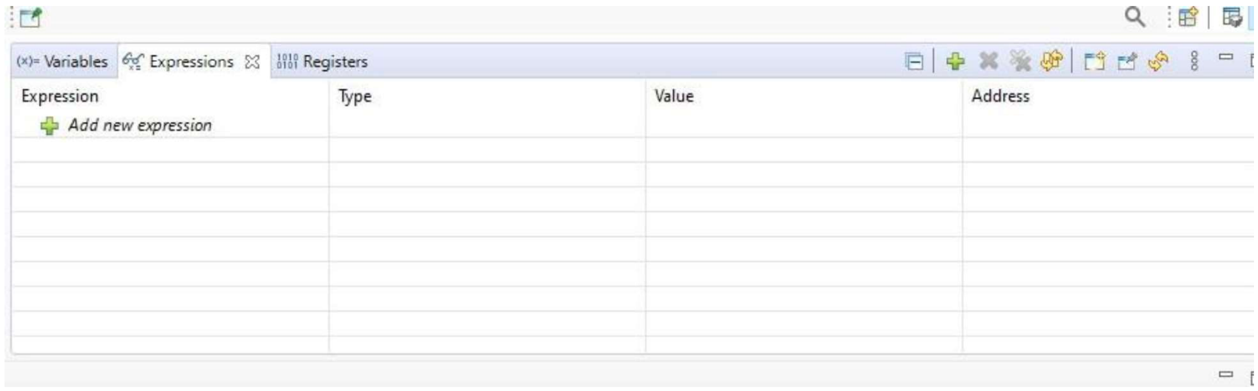
While the use of global variables is considered bad, programmers writing C to run on controllers and other small pieces of hardware do use them. Most often it is used to hold addresses of key hardware pieces that might be needed throughout the program. A global variable is created whenever you create a variable outside of a function, like this:

```c
int test(int test2);
int test3 = 0;

/**
 * main.c  */
void main(void)
{
    int test1 = 0;
    test1 = 3;
    test3 = 2;

    test1 = test(test1);
    return;
}

int test(int test2)
{
    test2 = 4;
    test3 = 3;
    return test2;
}
```

All global variables you create exist the whole time your program is running and can be accessed from anywhere in your code. You might be tempted to use global variables all the time instead of local ones, that way you wouldn't have to pass any parameters or return any values. If you make this choice, your application will have problems. Since your variables will be alive all the time, instead of only being placed on the stack when they are needed, your app will waste a lot of memory. It will also make your code buggy, crash, and hard to debug. Use global variables ONLY when you must.

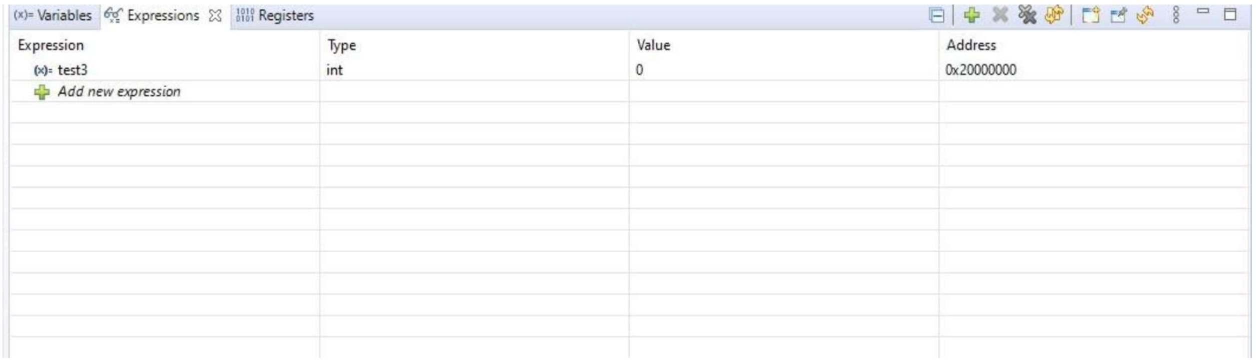
You are going to want to use stepwise debugging to see what happens to *test3* and when it changes. Unfortunately, it won't show up in your variables watch list like local variables, so you'll need to add it by hand as an expression. I'll step you through how to do this.

In the watch window, select the Expressions tab.



Expression	Type	Value	Address
Add new expression			

Now click the + sign before the Add new expression. Enter the *test3* name. You should now be able to see the value of the *test3* global variable:



Expression	Type	Value	Address
(x)= test3	int	0	0x20000000
Add new expression			

After completing this, compile, run, and debug the global variable version of the program and step through it. Watch what happens to the value of the global variable *test3*.