

Introduction to the C language and Variables

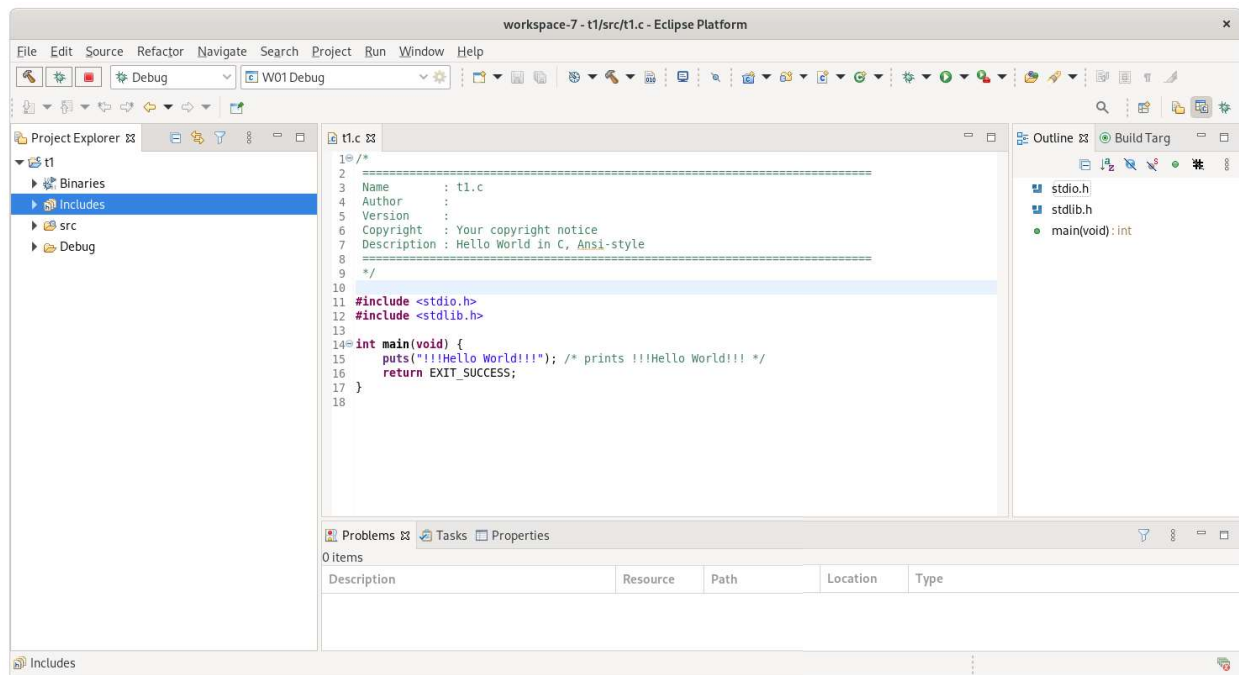
Introduction:

The objective of this lab is to allow you to understand how to use typed variables in the C coding language.

This text assumes that you are using the Eclipse IDE for writing and debugging your code. If you are using Windows, you should also have installed the MinGW and Msys packages. Using other environments in this course is allowed (such as Visual Studio Community on Windows or Xcode on MacOS); if these are used you should adapt these instructions for your own environment.

When you open Eclipse, you should be prompted for the workspace you want to use. Remember this workspace is the directory where your project files will be placed. You can use several different workspaces if you want to work on very different projects. Click Launch when you have specified the workspace directory (you can just use the default.)

After opening the workspace, you should create a new project. Under the File Menu, choose “New,” then “Project.” Under the C/C++ option, choose “C Project,” then click the “Next” button. Give the Project a name. Under Project type, open the “Executable” folder and choose “Hello World ANSI C Project.” Under Toolchains, choose “Linux GCC” if you’re on Linux, or “MinGW/Msys” if you are on Windows. Click the “Finish” button. Eclipse may then ask you if you want to open the C/C++ perspective. You should click the “Open Perspective” button. You should see something like this:

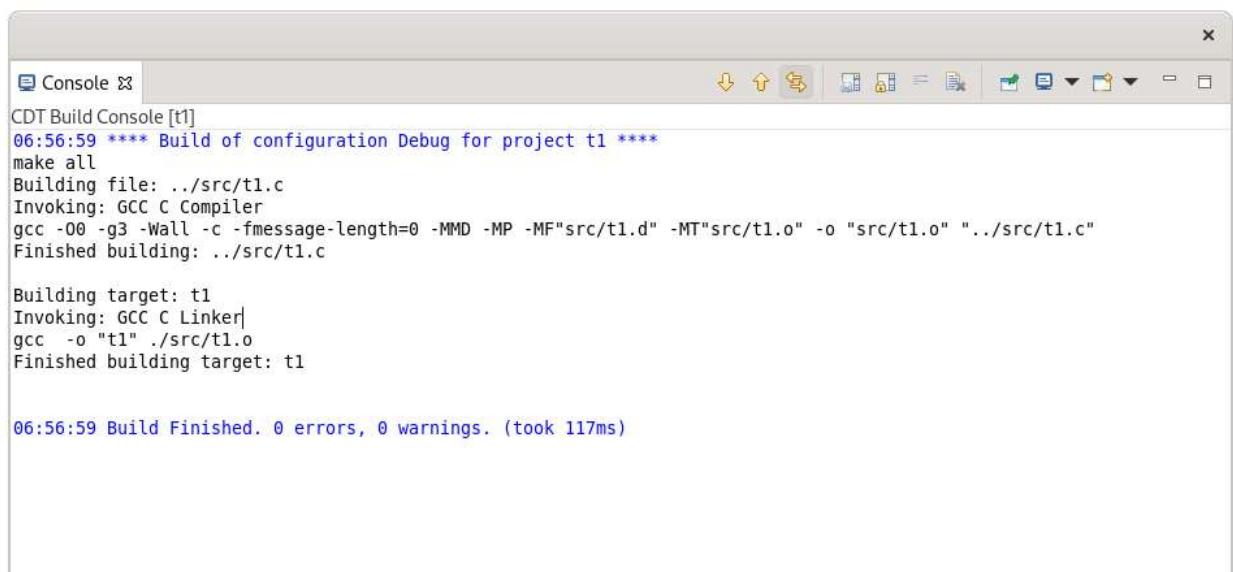


This will look different from a Python program. Let's look at some details. First, you will notice there is a main function. In C this is where execution of the code starts, not at the top of the file.

Also notice that there is *int* before the main function declaration. This tells the system that this function is set up to return an integer. Inside the parentheses after the main function name is the parameter list that this function takes. Right now, this is set to void which means that the function takes no parameters (void means nothing in C). C is a typed programming language, which means you'll always need to tell it the type of variables. This indicates to the compiler how much storage space should be allocated to store the variable's value.

Also notice the curly braces around the statements in the function. In Python you use indentation to denote the statements in a function, in C you will use {}.

Before you start making changes, let's make sure the code compiles. Right click on the project, then click on Build Project. In the Console window you should see the following:



```
CDT Build Console [t1]
06:56:59 **** Build of configuration Debug for project t1 ****
make all
Building file: ../src/t1.c
Invoking: GCC C Compiler
gcc -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"src/t1.d" -MT"src/t1.o" -o "src/t1.o" "../src/t1.c"
Finished building: ../src/t1.c

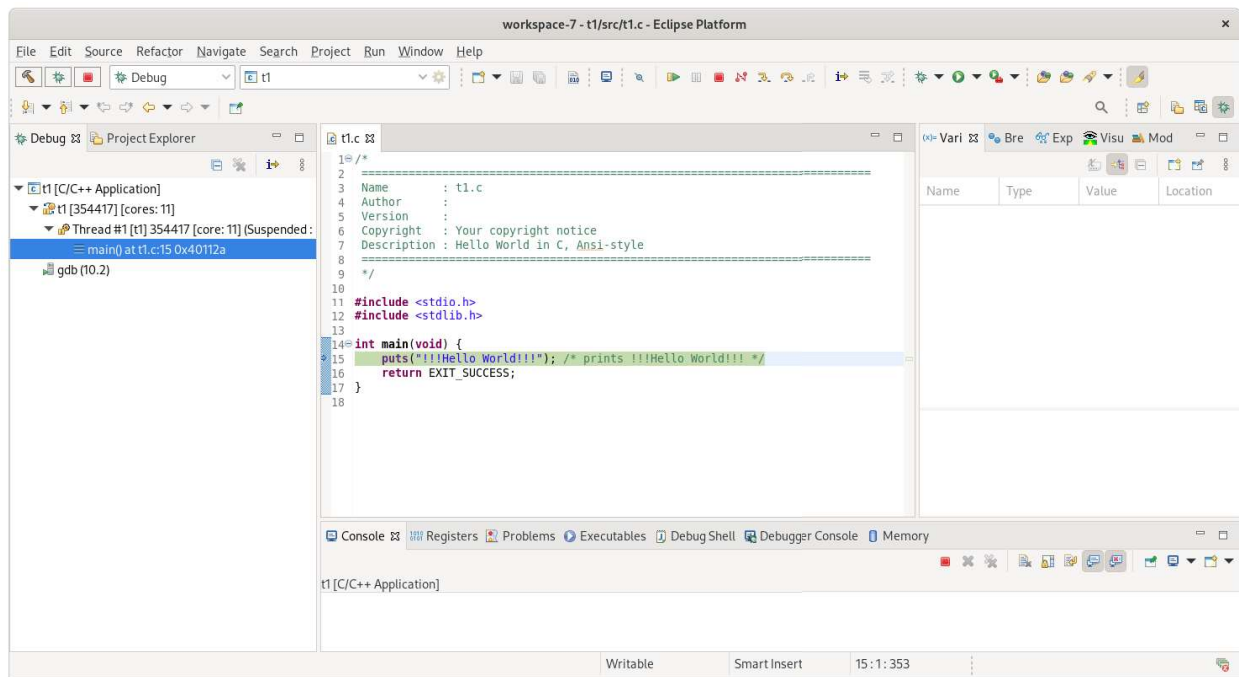
Building target: t1
Invoking: GCC C Linker
gcc -o "t1" ./src/t1.o
Finished building target: t1

06:56:59 Build Finished. 0 errors, 0 warnings. (took 117ms)
```

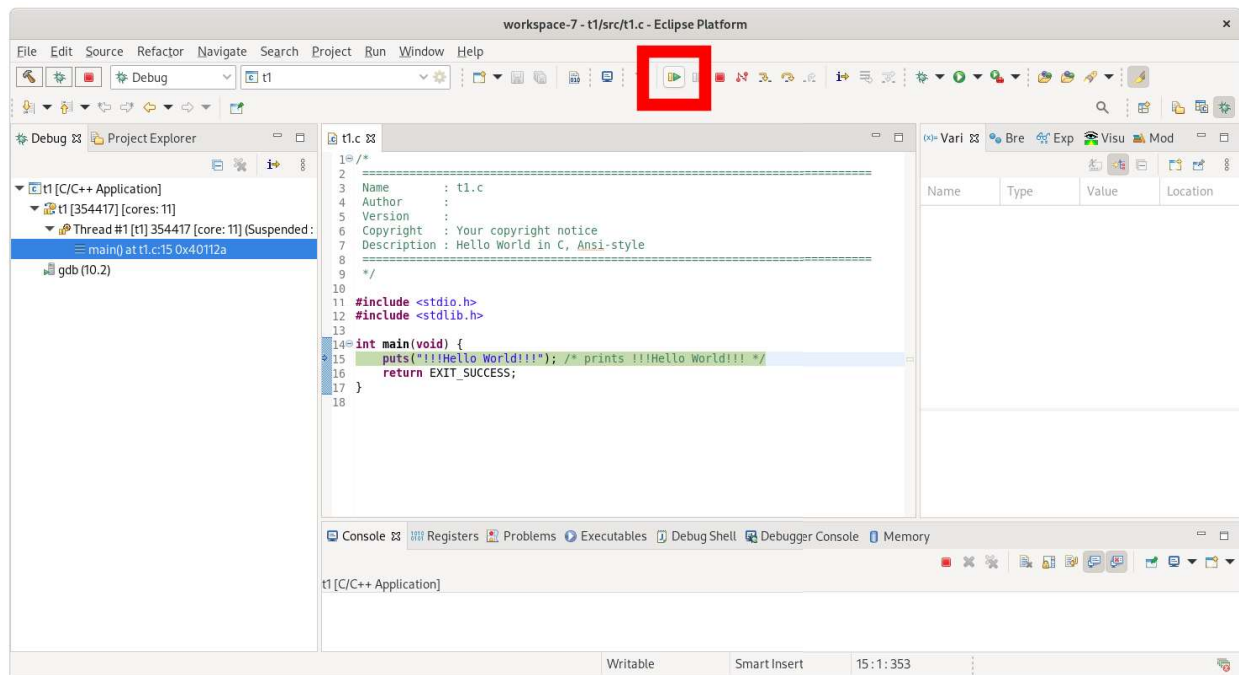
The Build process is something that all compiled languages, like C, must do prior to shipping or running. Some languages, like Python or JavaScript, are interpreted languages, means that there is a program running on the device that reads each command from the script file and interprets it for the computer. Thus, you need not only your code, but the interpreter program to be running, for your program to execute successfully.

Since C is a compiled programming language, a special program called a compiler takes your program and turns it into machine code. When you want to run your program you don't need any other program to run your code; your program is already in machine code in a runnable state.

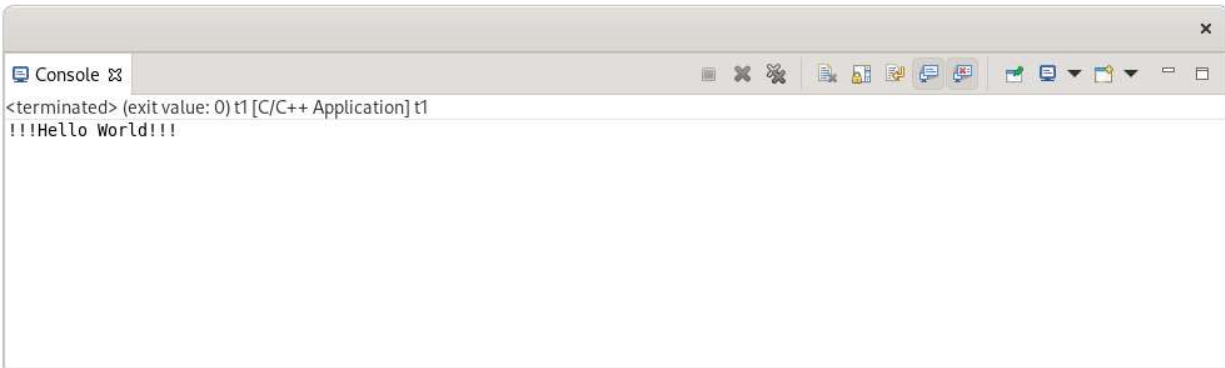
You can now run your program by right clicking on the project and hitting the Debug As ->Local C/C++ Application. The view will switch to the Debug View, and you should see this:



When you start debugging the program, it will always pause in the first statement inside your main function. To run the code, you should press the Resume button or hit F8:



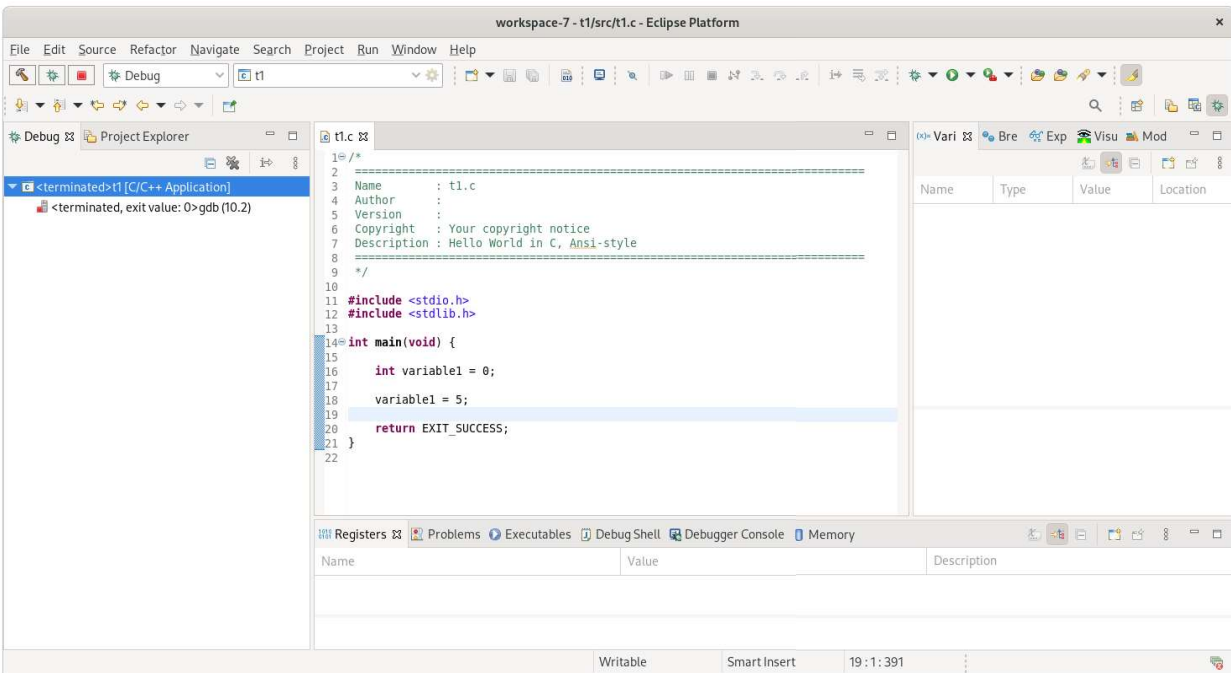
After you resume the program, you should see this in the console pane:



Now that you have a framework, you can add code.

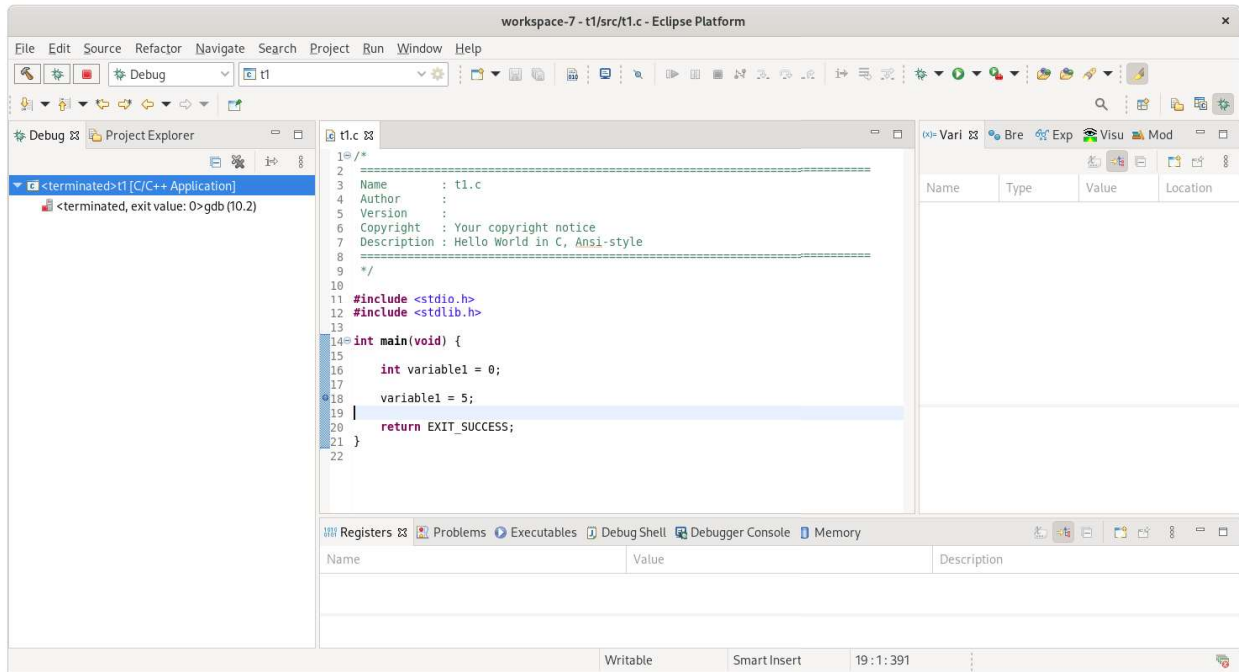
Simple Typed Variables

Let's understand typed variables in the C coding language. Change your main.c add a variable, like this:

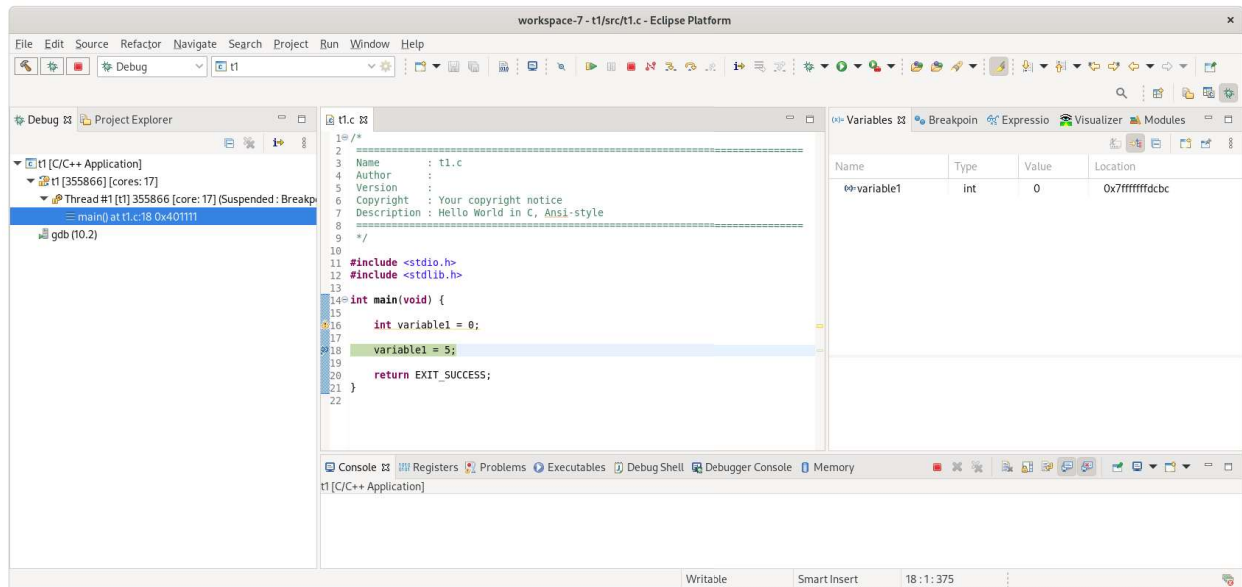


The declaration for any variable works like this; first you detail the type, then the name of the variable, and then, optionally, you can initialize the variable to a value. It's good practice to always initialize every variable you create.

Compile and load the code. Before running the code, however, let's put some break points in the code to see what is happening with our variable. To add a break point, after you've reached the debug view, go to the start of line 18, then right click and you should see the dialogue box pop up. Select "Toggle Breakpoint" here. The small dot will then appear in the gutter next to line 18 to indicate a breakpoint:

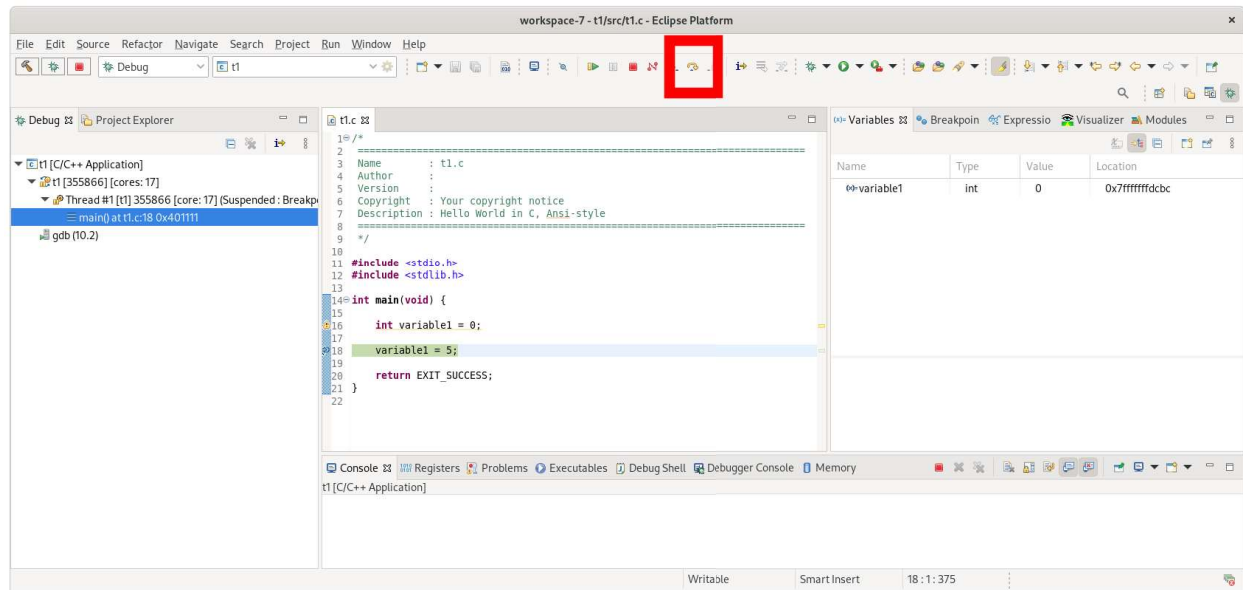


This will stop the execution of the code just before line 18. Now go ahead and hit resume. Your code should run but stop on line 18. Eclipse provides a way for you to look at the value associated with your variable. In the upper right pane, you can see the value of the variable:

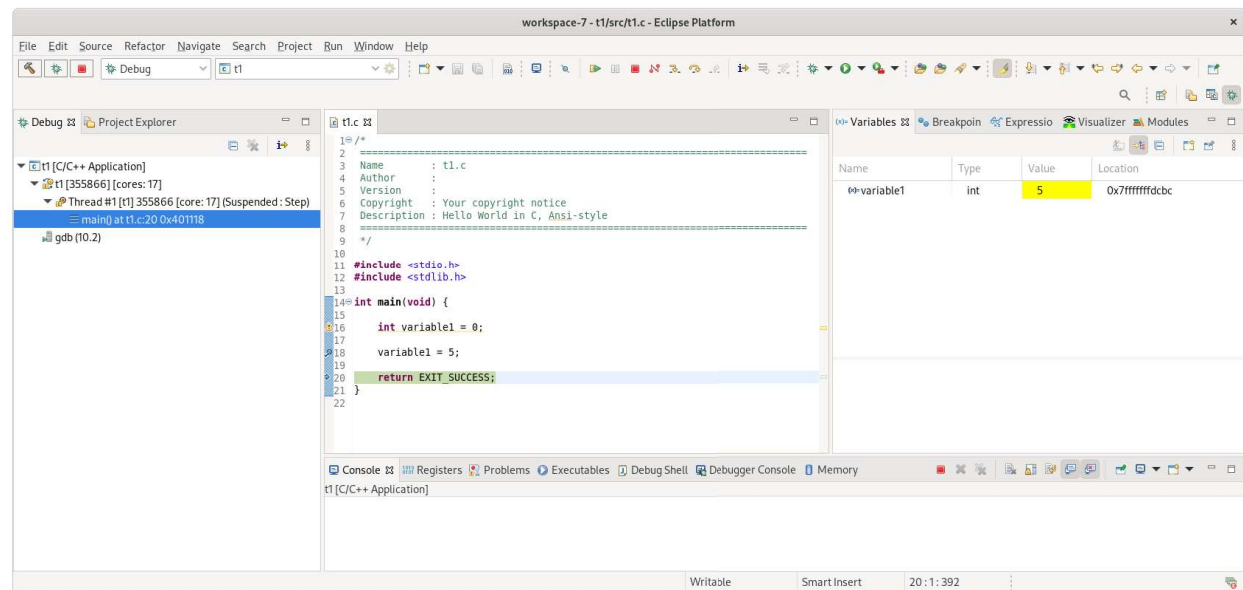


In this case you have a variable named variable1, it is of type int, its current value is 0, and its Location is 0x7fffffffdbcb. Your actual value may be different depending on where the compiler has decided to allocate memory for your variable.

Now use the Step Over button to step over line 18:



Now you should see this:



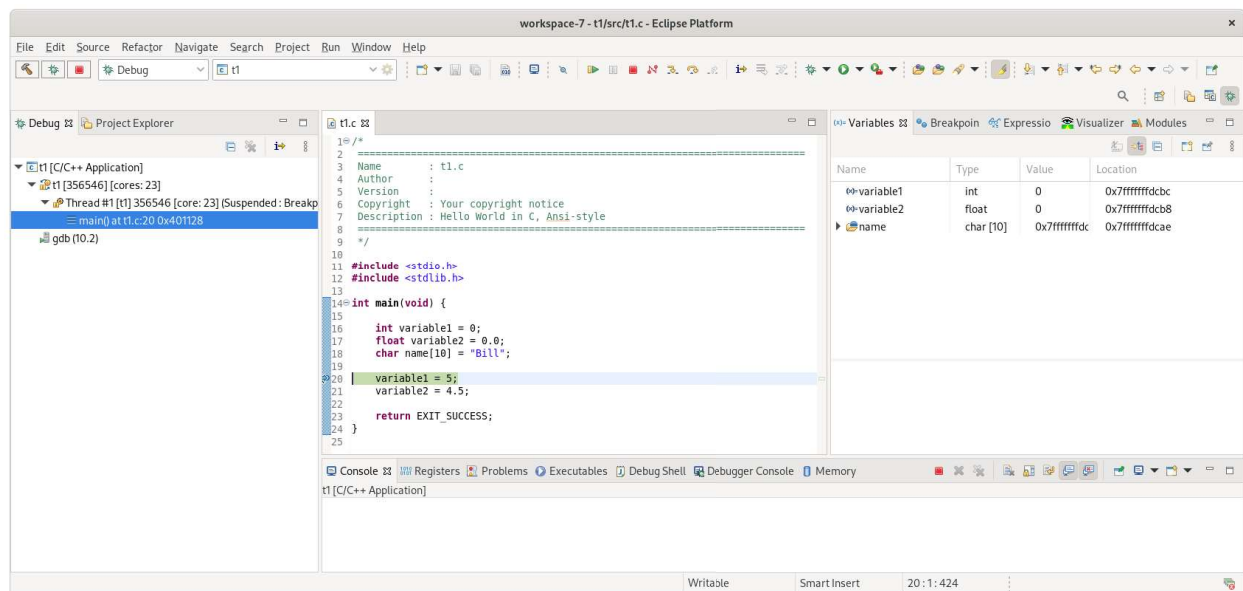
Notice the value has changed to 5.

Let's explore some other variable types. Change the code in main.c to 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     int variable1 = 0;
16     float variable2 = 0.0;
17     char name[10] = "Bill";
18
19     variable1 = 5;
20     variable2 = 4.5;
21
22     return EXIT_SUCCESS;
23 }
24
25
```

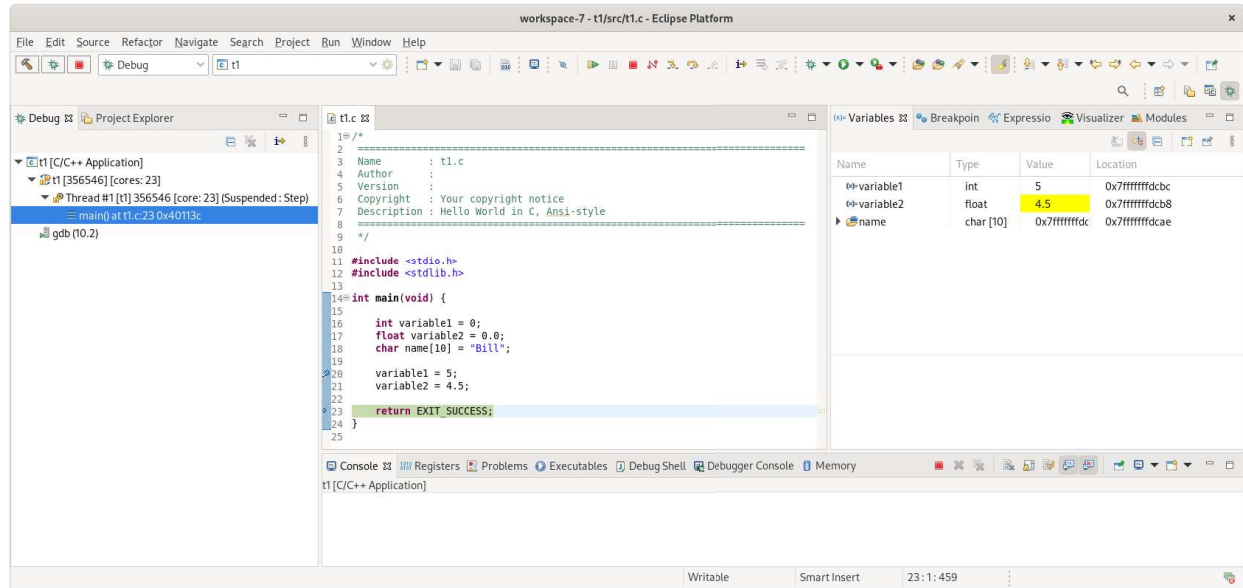
This introduces two new types of variables, a float and character array. Floats hold small values that have a decimal point. Character arrays hold string values. In this case you've allocated up to 10 characters for the *name* array. It is important to note that the NULL character, or the 0 value, must always be at the end of an array of characters, so if you want to store names of up to 10 characters in length, you'd need to allocate 11 characters for the array.

Now compile the code. If you haven't changed it your breakpoint should still be at the *variable = 5;* statement, so resume the code and it should stop at that line. Then you should see something like this:



Notice all our variables are shown in the Variables space. Each has a type, and an address (See the Location column in the table). If you look closely, you'll see that each character in the name array is stored as a numeric value as represented by the ASCII table. You'll also note that after the last "l" of the name is a NULL character to indicate that this name stops with the fourth character.

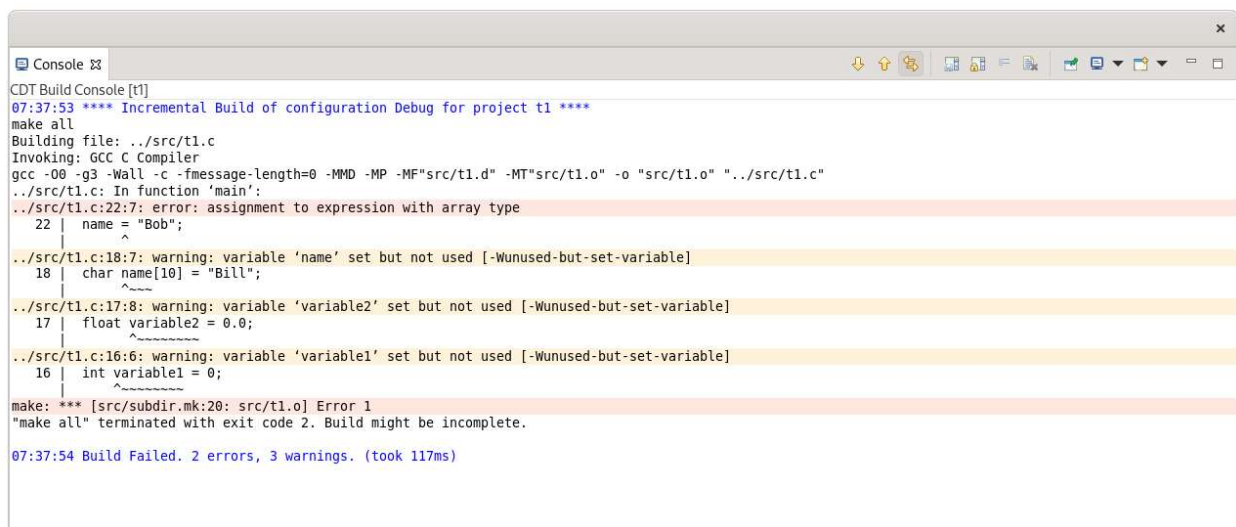
Now Step Over lines 20 and 21 and you should see this:



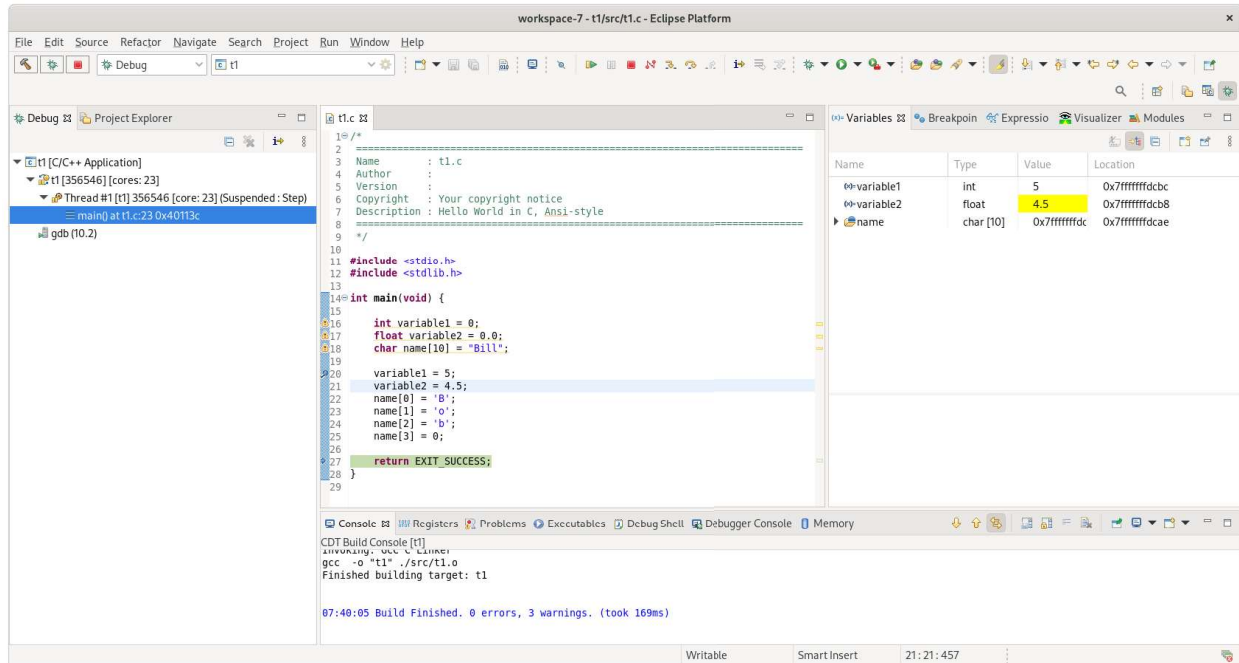
Now the obvious question is, “Why didn’t you assign a value to *name* like this?”

`name = “Bob”;`

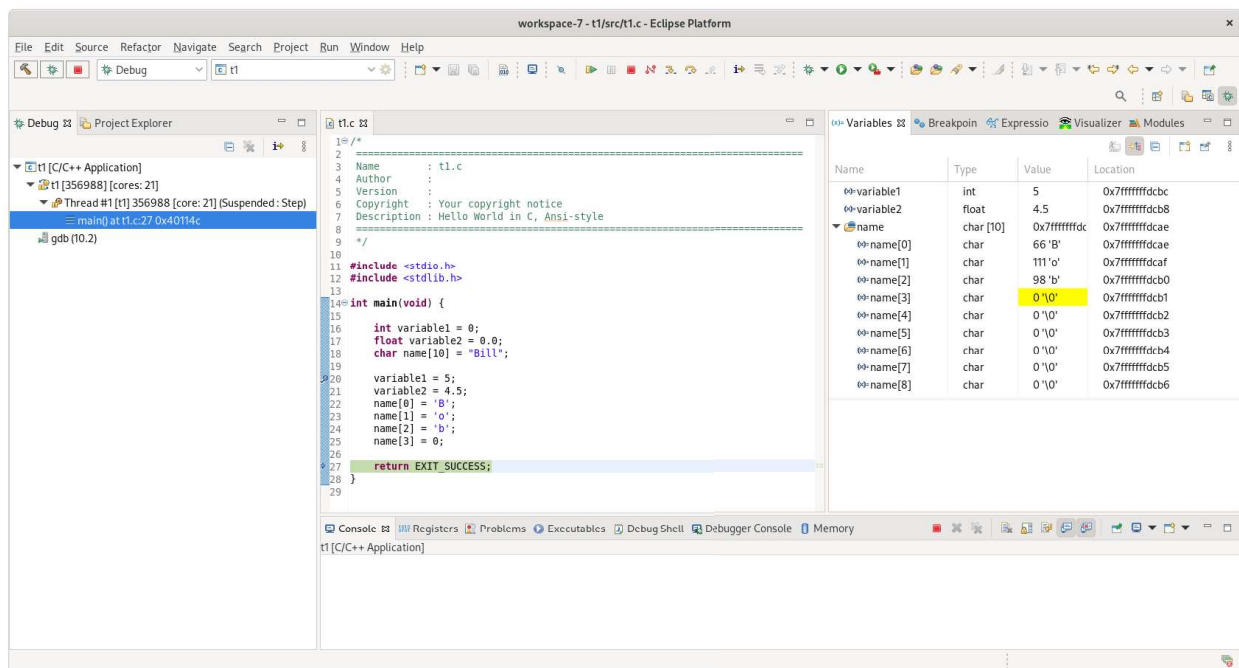
And the answer is this would have caused a compile error, like this:



Character arrays are a bit tricky; in C you cannot just use the assignment operator to change them. You can write individual characters, like this, but it is a cumbersome process:



Now when you run and step through the program you should see this in the variable space:



Notice line 25. It is important that you end your character array with the NULL character (value 0) or the computer won't know how many valid characters the array contains.

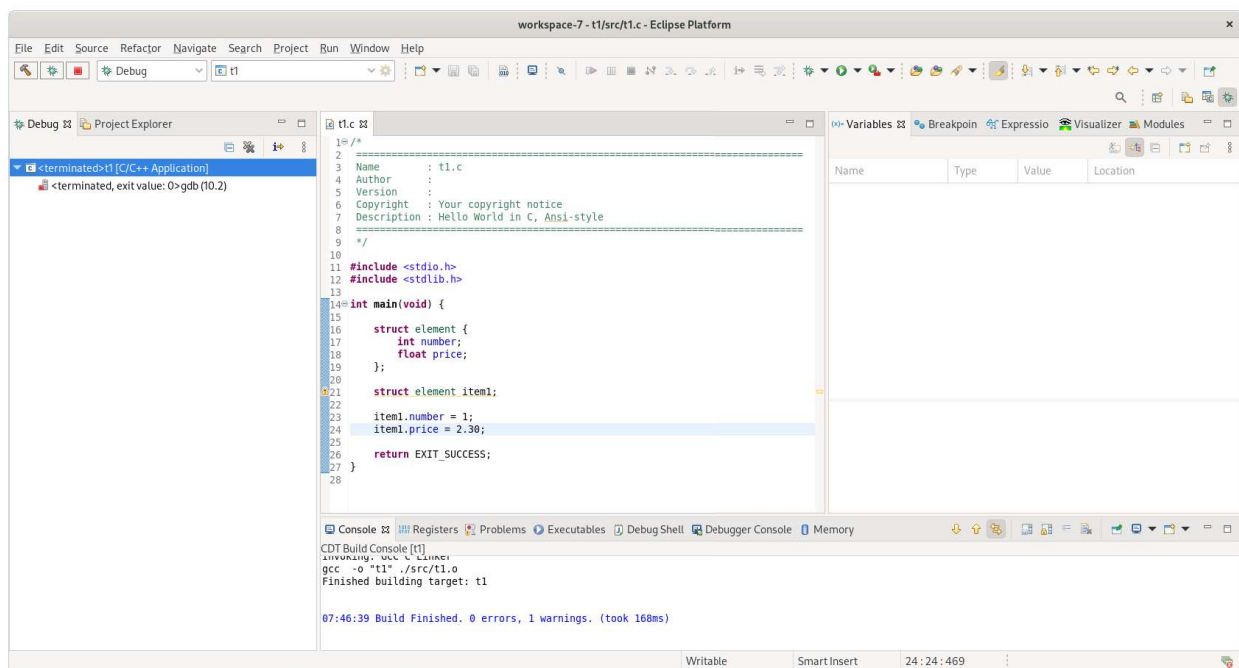
The compiler supports a significant number of other standard data types, including these:

int1	Defines a 1-bit number
int8	Defines an 8-bit number
int16	Defines a 16-bit number
int32	Defines a 32-bit number

char	Defines an 8-bit character
float	Defines a 32-bit floating-point number
short	By default, the same as int16
long	By default, the same as int32
void	Indicates no specific type

Combining Data Types using Data Structures

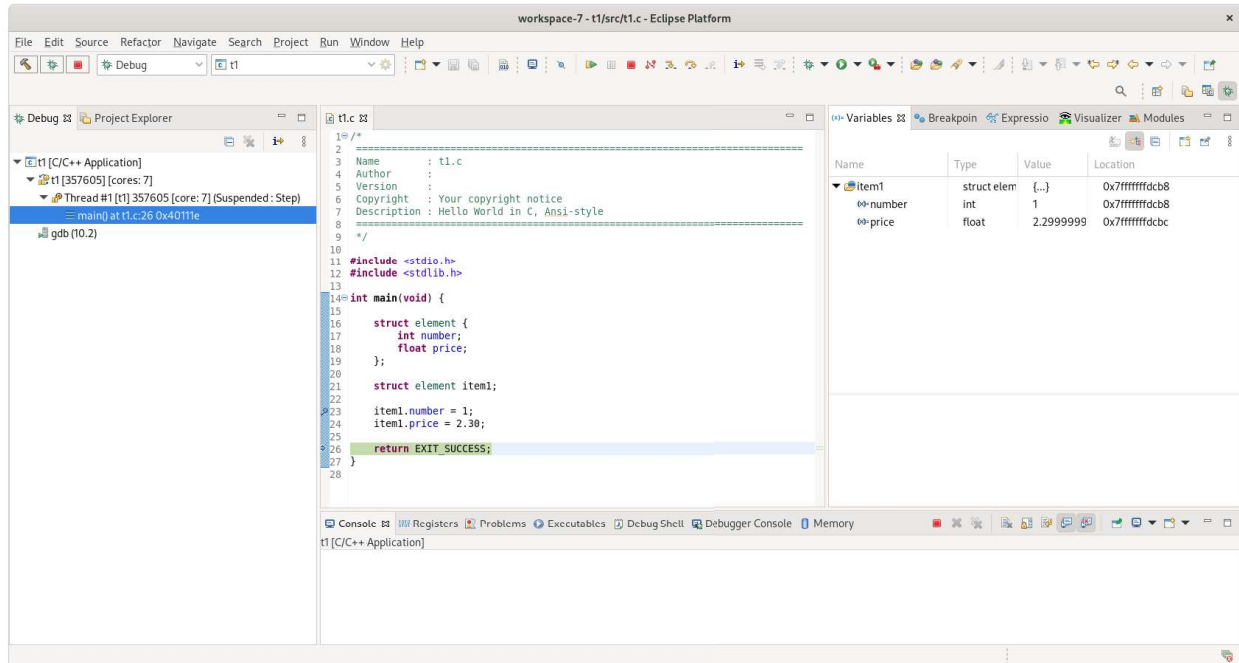
There are times when a data item might have more than one number associated with it. For example, you might be building an inventory of electronic parts. You might want to capture not only the number of parts available, but also the price that each part cost. To do this you can create a new data structure that contains both elements. To do this create the following code:



The *struct* keyword tells the compiler that you are going to create a custom, combined data type. In this case it has two elements, an integer variable named number, and a float variable named price. Be aware that the *struct* keyword used like this does not create a variable, it simply creates a definition of a new type you can use later.

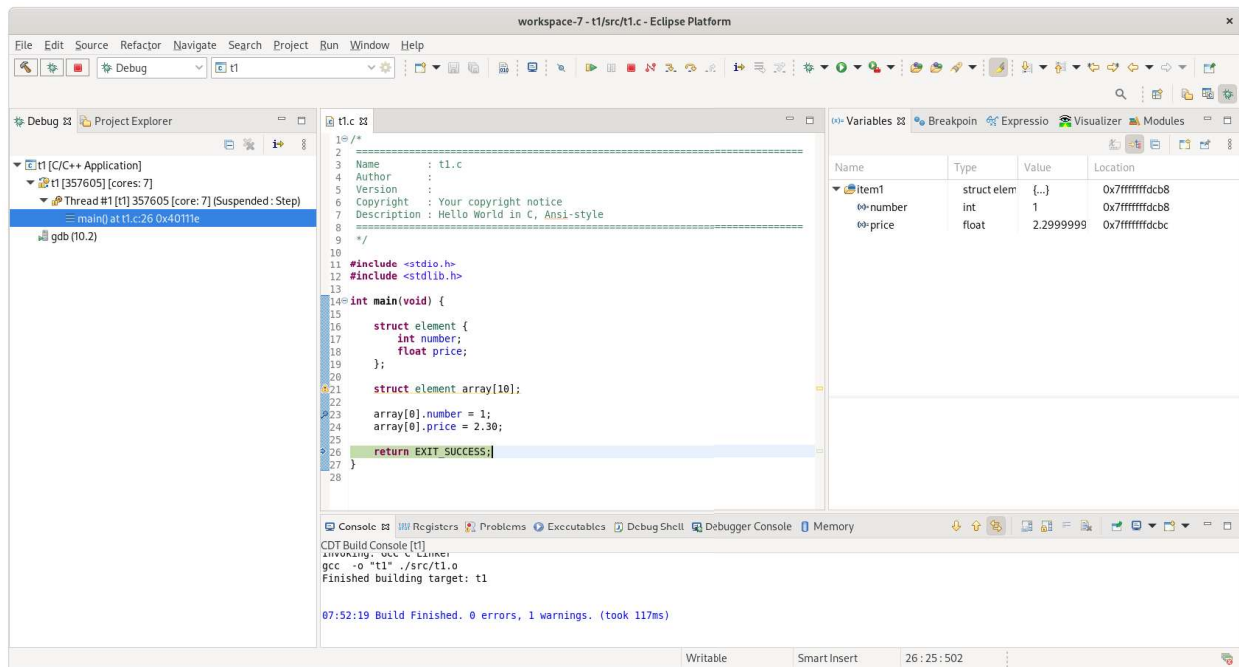
The *struct element item1* line of code creates an instance of the custom combined variable of type *element* and assigns it to the variable named *item1*. To use the variable requires the use of . (dot) notation. The line of code *item1.number = 1;* sets *item1*'s number. The next line of code, *item1.price = 2.30;* sets *item1*'s price.

Compile and run this code. Put a breakpoint on line 23. Hit Resume, and step over lines 23 and 24. In the variable selection click on the arrow key just to the left of the item1 variable name, and you should be able to expand the item to see its individual parts.



It is interesting to note that the price is not stored as exactly 2.30. The actual value stored is as close as possible to 2.30 based on the accuracy of the storage location.

An inventory of a single item doesn't make much sense, so let's create an array of *elements* called *array*:



Now compile and debug the program. If you step over until line 26 and look at the variables you should see something like this:

Name	Type	Value	Location
array	struct element [10]	0x7fffffffda0	0x7fffffffda0
array[0]	struct element	{...}	0x7fffffffda0
number	int	1	0x7fffffffda0
price	float	2.29999995	0x7fffffffda4
array[1]	struct element	{...}	0x7fffffffda8
array[2]	struct element	{...}	0x7fffffffdbc0
array[3]	struct element	{...}	0x7fffffffdbc8
array[4]	struct element	{...}	0x7fffffffdbc0
array[5]	struct element	{...}	0x7fffffffdbc8
array[6]	struct element	{...}	0x7fffffffdd00
array[7]	struct element	{...}	0x7fffffffdd08
array[8]	struct element	{...}	0x7fffffffdd10
array[9]	struct element	{...}	0x7fffffffdd18

Notice that the number and price values for the first element in the array have been set. Also notice that the other elements in the array have values that are not zero but hold whatever garbage the memory used happened to hold when it was allocated. It is important to note that the C programming language does not initialize variables. If you forget this, it can cause you real problems when running programs. That's why you should always initialize all variables you create. Otherwise, when your code runs, the results will seem random, when really your code is using memory that wasn't cleaned up and holds data from the last time the memory was used.

Also notice where the index operator `[]` is used when accessing the array. To access elements in an array, you always put the square brackets immediately after the name of the array. In this case it's the variable named `array`. Each element in the array is a structure, so the structure member operator `.` (dot) is used after the array index operator. It's important to understand this, since you can also have arrays that are part of a structure, like this:

```

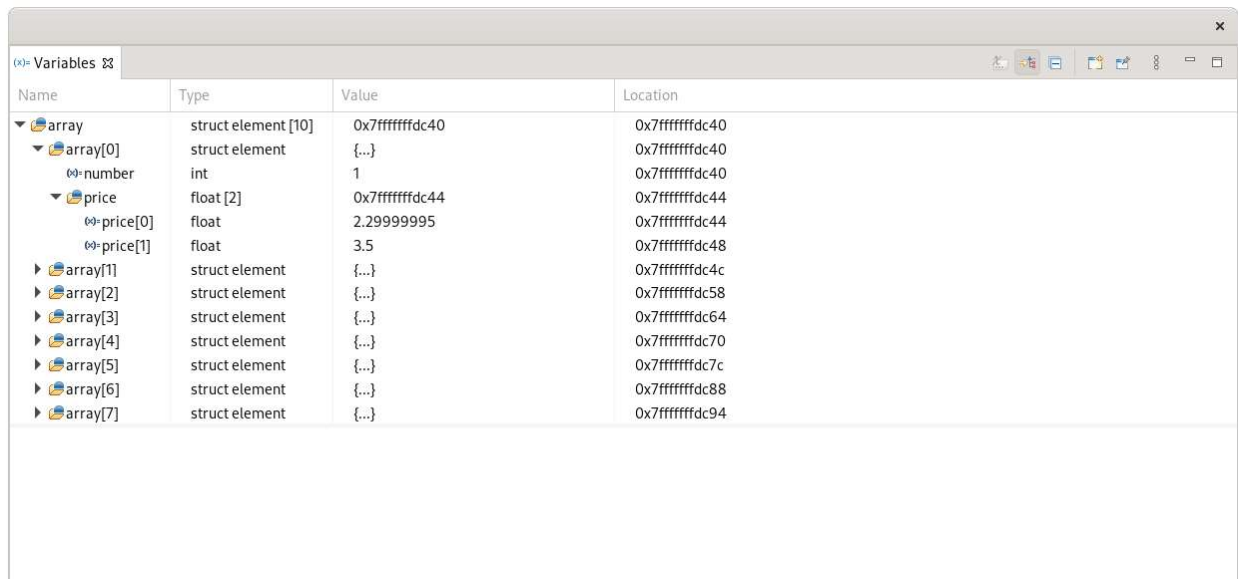
1 // **
2 =====
3 Name      : t1.c
4 Author    : 
5 Version   : 
6 Copyright : Your copyright notice
7 Description: Hello World in C, Ansi-style
8 **
9
10
11 #include <stdio.h>
12 #include <stdlib.h>
13
14 int main(void) {
15
16     struct element {
17         int number;
18         float price[2];
19     };
20
21     struct element array[10];
22
23     array[0].number = 1;
24     array[0].price[0] = 2.38;
25     array[0].price[1] = 3.58;
26
27     return EXIT_SUCCESS;
28 }
29

```

CDT Build Console [t1]
 gcc -o "t1" ./src/t1.o
 Finished building target: t1

07:59:44 Build Finished. 0 errors, 1 warnings. (took 118ms)

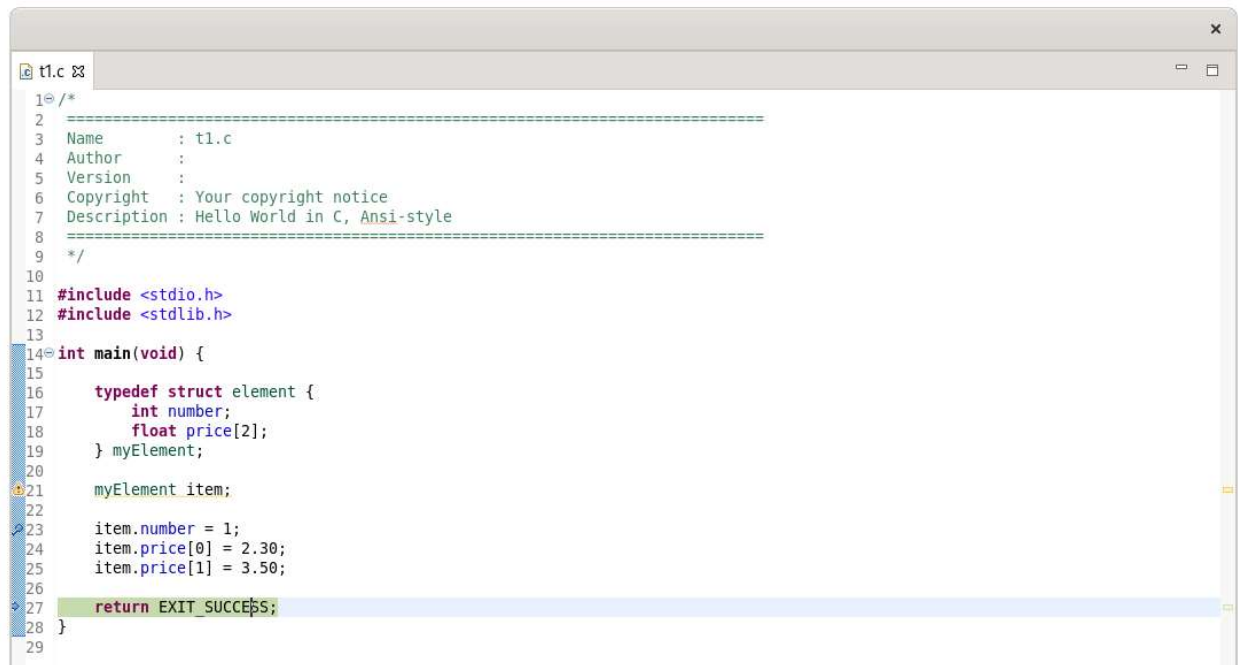
You might need to do this if you must capture the maximum and minimum price of an item. In this case, because the array is part of the structure, you put the index operator `[]` after the price. This tells the compiler that price is an array of floats instead of a single float. Build and debug this code, step over the assignment operators, and you should see this in the variable space:



Name	Type	Value	Location
array	struct element [10]	0x7fffffffcd40	0x7fffffffcd40
array[0]	struct element	{...}	0x7fffffffcd40
array[0].number	int	1	0x7fffffffcd40
array[0].price	float [2]	0x7fffffffcd44	0x7fffffffcd44
array[0].price[0]	float	2.29999995	0x7fffffffcd44
array[0].price[1]	float	3.5	0x7fffffffcd48
array[1]	struct element	{...}	0x7fffffffcd4c
array[2]	struct element	{...}	0x7fffffffcd58
array[3]	struct element	{...}	0x7fffffffcd64
array[4]	struct element	{...}	0x7fffffffcd70
array[5]	struct element	{...}	0x7fffffffcd7c
array[6]	struct element	{...}	0x7fffffffcd88
array[7]	struct element	{...}	0x7fffffffcd94

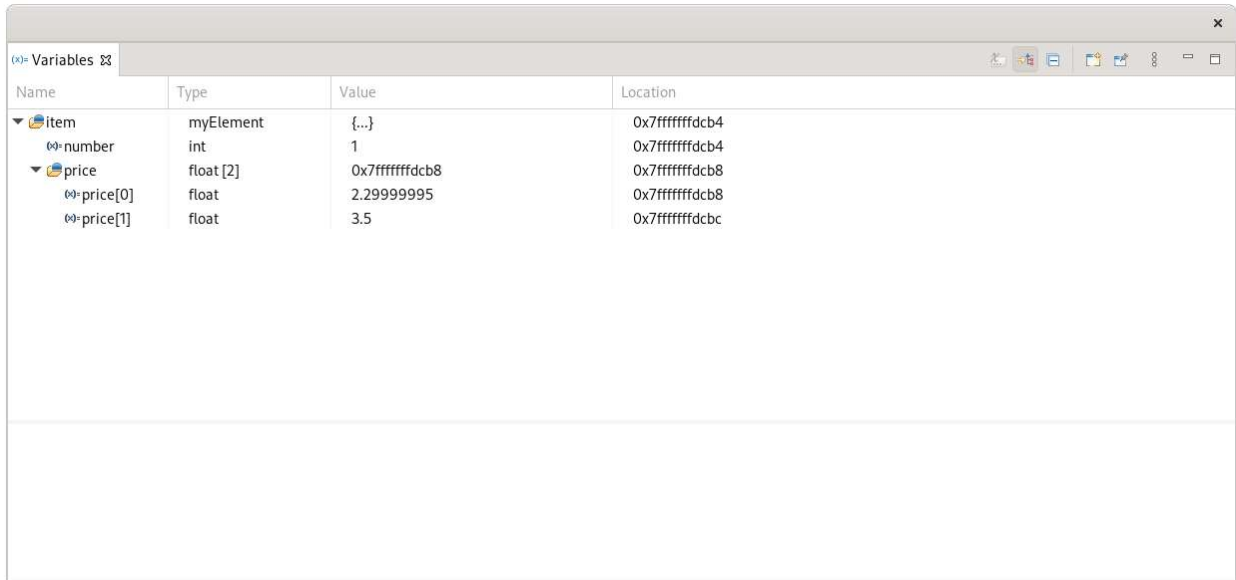
Creating Your Own Data Types using typedef

You can also create your own data types using *typedef* to make your programming easier to read. This is often used in conjunction with the *struct* keyword to create complex new types. If you change your code to 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     typedef struct element {
16         int number;
17         float price[2];
18     } myElement;
19
20     myElement item;
21
22     item.number = 1;
23     item.price[0] = 2.30;
24     item.price[1] = 3.50;
25
26     return EXIT_SUCCESS;
27 }
28
29
```

You will now notice that you've created a new type called `myElement`. It is associated with the data structure element, and you can now use it as you would any data type to declare new variables. Now build and run this code. Resume the code and step over the assignments and you should see this in the variable space:



Name	Type	Value	Location
item	myElement	{...}	0x7fffffffedcb4
number	int	1	0x7fffffffedcb4
price	float [2]	0x7fffffffedcb8	0x7fffffffedcb8
price[0]	float	2.29999995	0x7fffffffedcb8
price[1]	float	3.5	0x7fffffffedcbc

Arrays and Structures

Here's another example.

```
struct myItem {
    int id;
    float price;
};

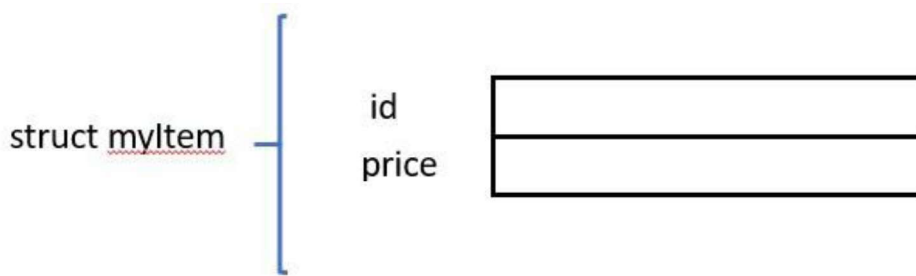
/** main.c */
void main(void)
{
    struct myItem item1;
    struct myItem item2;

    item1.id = 1;
    item1.price = 2.30;

    item2.id = 2;
    item2.price = 3.50;
}
```

The introduction of structures allows you to organize your data, when it makes sense, even if the data is not the same type. In this case you've introduced a new data type, the *struct myItem* which has an integer `id` and a float `price` associated with each variable of type `struct myItem`. When you declare of

variable type struct myItem it will contain both an integer and a float. Here is a diagram that might help:



To access each individual element of the structure you'll use the dot notation. For example:

```
item1.id = 1;
```

assigns the id value of the variable name item1 to 1.

Now compile the program and then debug it. Step over the two variable creations, let's look at the variable view:

Name	Type	Value	Location
▼ item1	struct myItem	{id= 1,price= 1.89035163e-42 (DEN)}	0x2000FFE8
(x)= id	int	1	0x2000FFE8
(x)= price	float	1.89035163e-42 (DEN)	0x2000FFEC
▼ item2	struct myItem	{id= 0,price= 0.0}	0x2000FFF0
(x)= id	int	0	0x2000FFF0
(x)= price	float	0.0	0x2000FFF4

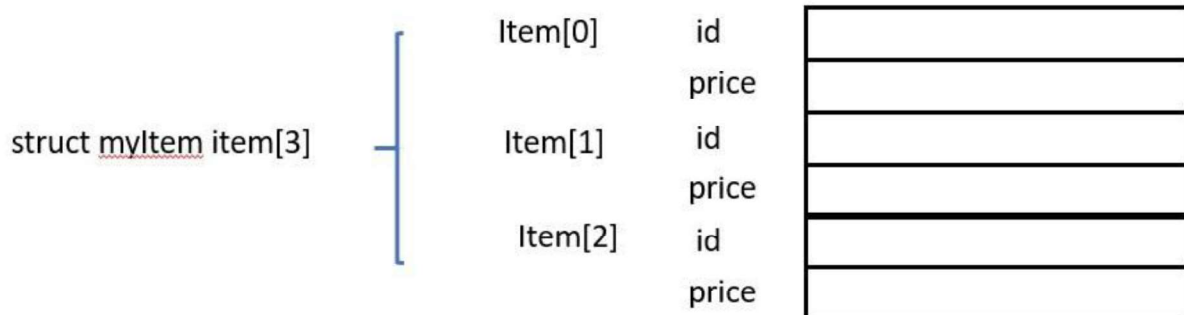
Notice how the variables item1 and item2 have two elements. At this point they haven't been set to anything, but they do exist. Now step to the point where item1 has been set:

Name	Type	Value	Location
▼ item1	struct myItem	{id= 1,price= 2.29999995}	0x2000FFE8
(x)= id	int	1	0x2000FFE8
(x)= price	float	2.29999995	0x2000FFEC
▼ item2	struct myItem	{id= 0,price= 0.0}	0x2000FFF0
(x)= id	int	0	0x2000FFF0
(x)= price	float	0.0	0x2000FFF4

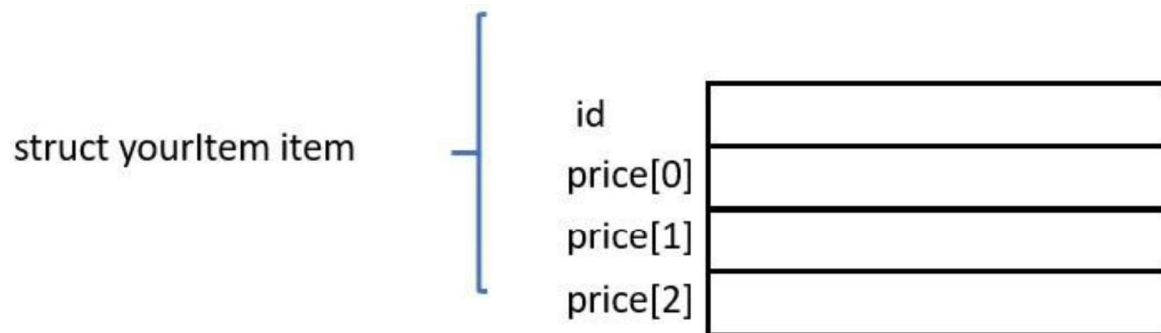
You can now see the values have been set for item1. Now step into the function test_function. As you step through the function you can see them change. When you return from the function you can also see your original item2 change.

One of the most useful ways to use structures is in arrays. There are, however, two ways to do this, and they can be confusing, so let's look at each.

First you can create an array of the struct `myItem`. This would look like this in memory:



You can also create a struct that contains an array as one of the elements:



Let's show some code with an example of both.

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

struct myItem {
    int id;
    float price;
};

struct yourItem {
    int id;
    float price[3];
};

/**
 * main.c */
void main(void)
{
    struct myItem item1[3];
```

```

struct yourItem item2;

item1[0].id = 1;
item1[0].price = 3.56;

item2.id = 2;
item2.price[0] = 5.30;
}

```

Notice for item1, which is the array of structures, to access a single item you use the syntax:

```
item1[0].id = 1;
```

For item2, which contains an array of prices, to access a single item in the array you use the syntax:

```
item2.price[0] = 5.30;
```

Now build and debug this code. When you reach the end of the program and look at the Variable viewer you should see this:

(x)= Variables Expressions Registers			
Name	Type	Value	Location
▼ item1	struct myItem[3]	{id= 1,price= 3.55999994},{id= 1520,price=...	0x2000FFD8
▼ [0]	struct myItem	{id= 1,price= 3.55999994}	0x2000FFD8
id	int	1	0x2000FFD8
price	float	3.55999994	0x2000FFDC
▼ [1]	struct myItem	{id= 1520,price= 0.0}	0x2000FFE0
id	int	1520	0x2000FFE0
price	float	0.0	0x2000FFE4
▼ [2]	struct myItem	{id= 1,price= 1.89035163e-42 (DEN)}	0x2000FFE8
id	int	1	0x2000FFE8
price	float	1.89035163e-42 (DEN)	0x2000FFEC
▼ item2	struct yourItem	{id= 2,price= [5.30000019,0.0,1.97162694e-42 (DEN)]}	0x2000FFF0
id	int	2	0x2000FFF0
price	float[3]	[5.30000019,0.0,1.97162694e-42 (DEN)]	0x2000FFF4
[0]	float	5.30000019	0x2000FFF4
[1]	float	0.0	0x2000FFF8
[2]	float	1.97162694e-42 (DEN)	0x2000FFFC

You can see that item1 is an array of structs, while item2 is a single item that contains an array of prices.