

Debugging with Qt Creator

Introduction

Program crashing? Weird things happening? Error messages galore? These are problems that can be resolved by using a technique called debugging. Debugging is an important skill for all programmers to learn. In order to debug your program, you'll need to use your interactive development environment (IDE), Qt Creator, in order to step through the execution of your program to discover what is causing problems.

Print Debugging

Quick Summary

- Uses: print variables and help you tell what code your program is actually executing
- How: Print variables using cout along with other helpful text listed below
 - When printing a variable, also print its name to help distinguish print lines
 - Also print the location of print statement like "start of getNeighbors() method"
- Code: `cout << "myVec is " << myVec << " at end of calc()" << endl;`

Detailed Walkthrough

The first topic we'll cover is print debugging where print statements are inserted into a program in order to help the programmer determine the order of actions executing in the program as well as the value held by different variables at certain places. When a program's behavior does not match the expected results, it's important to get a high overview of what the program is doing. This is a great time to use print debugging. The code for printing a simple int variable is shown below.

```

C++
int x = 5;
cout << "The variable x equals " << x << endl;
//Prints "The variable x equals 5"
```

Let's break down the code above to show the basic steps of print debugging. The first line of the code declares and initializes an integer variable called x with the value 5. The second line of the code prints out a text message "The variable x equals" followed by the the value of x.

In this example, the message explains that the printed value belongs to the variable x. **It is important to include a message when printing out variable values to help you track which values belong to which variables as you start adding more print statements to your program.** Remember to put a space at the end of the message string so that the value of the variable is separated from the message to help readability. Also, make sure to print separate messages on different lines by including the endl keyword at the end of a cout statement in C++ as shown in the example above. Note that you should **always add an "endl" token** to the end of your debugging print statements since this will force the console buffer to flush the text to the console so that you will be able to see them. If you don't include the "endl" token, the console may choose to wait to print the text until the rest of the line has been given.

Along with the message, the code example prints the variable's value by appending it to the message. In C++, all primitive types (integer, double, boolean types etc.) and many objects have a toString() function which will convert the value of a variable into an easily readable string that can be printed out to show what the variable contains. In C++, in order to invoke the toString() function of a variable, you need to use the stream operator (<<) to add the variable to the cout stream which is responsible for printing text to the screen. For example, the Vector class has a toString() function so if you want to investigate the values stored in a Vector you can do the following:

```
C++
```

```
Vector<int> numbers;
numbers.add(2);
numbers.add(3);
numbers.add(4);
cout << "Numbers: " << numbers << endl;
//Prints "Numbers: {2, 3, 4}"
```

Another reason to use print debugging is to ascertain the execution order of certain statements within the program. This can be helpful to fix mistakes related to incorrect ordering of statements within the program even if those statements are in different methods. Also, print statements can be used to see if a method is being called at all if you suspect that it is not executing for some reason. In order to solve these issues, **in addition to printing a message describing the value and then the value, you should print out where in the program this message is originating from.** Here's an example:

```
C++
```

```
void method1() {
    int x = 0;
    cout << "At start of method1, the value of x is " << x << endl;
    //Prints "At start of method1, the value of x is 0"
    x = x + 1;
    cout << "At end of method1, the value of x is " << x << endl;
    //Prints "At end of method1, the value of x is 1"
}

void method2() {
    int x = 3;
    println("In method2, the value of x is " + x);
    //Prints "In method2, the value of x is 3"
}
```

In the example above, each print statement includes the name of the method which contains the print statement. This helps you determine where each print statement came from in the code especially when you are tracking the same variable or a variable with the same name. For example, below you can see the output of a possible program which calls these methods where the messages include and don't include location information:

Console

The value of x is 3.
The value of x is 0.
The value of x is 1.

Console

In method2, the value of x is 3.
At start of method1, the value of x is 0.
At end of method1, the value of x is 1.

The output on the left isn't very helpful when you look back at the code since you can't tell which print statement resulted in which output line. However, from the output on the right, you can tell that the program calls method 2 then calls method 1. As your programs become more complex, this location information in your print debugging can help you find errors in your program.

If you find your program is printing out so many debugging print statements that it gets hard to read it, you can comment out old print statements or you can move onto other debugging techniques detailed in the rest of this handout.

Breakpoints & Debug View

Quick Summary

- Uses: pause program execution to allow you to step through program line by line
- How: Switch to debug view, add breakpoints to your program then use step over, step in, and step out button to walk through your program line by line.
 - See valuable debugging info in the Stack Trace window and Locals window.

Detailed Walkthrough

When you need further investigation into what your program is doing, breakpoints can be a great help. A breakpoint can be set at any line in your program. When your program is executing in debug mode, when it reaches a line with a breakpoint, the program will pause execution at that line. Once execution has paused, you will be able to look at the value of variables as well as step through your program line by line starting at this breakpoint in order to see exactly what is happening.

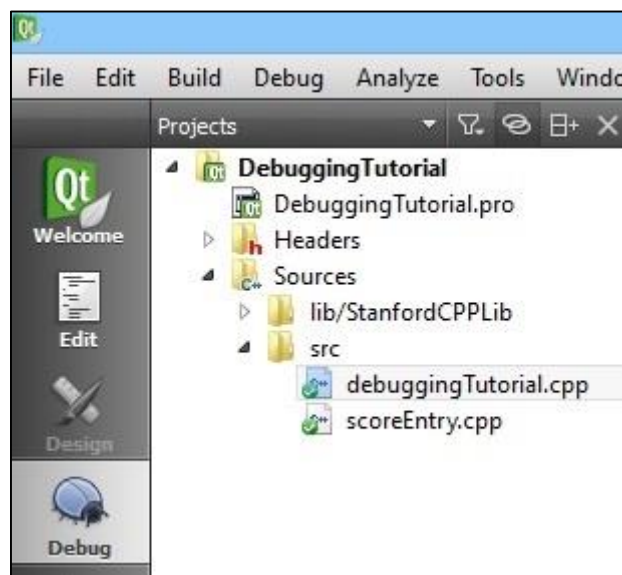


Figure 1

First, you must make sure you run your program in debug mode. **In Qt Creator, you'll want to switch your view to debug view by clicking the Debug view option in the left top menu as seen above in Figure 1.** Debug view allows you to see variable values, as well as which methods have been called which can be helpful to figure out what is occurring in your program. When you are done debugging and would like edit your program, you can switch back to edit mode by clicking the Edit view option in the left top menu as seen in Figure 1 as well.

Next, you can **begin debugging by clicking the Start Debugging button in the bottom left menu which is marked by the play button with the bug on it as seen in Figure 2.** You can also press F5 to start debugging as well. When you start debugging, the IDE will run your program like normal except that whenever a line with a breakpoint is encountered, the IDE will pause execution and then allow you to take control.

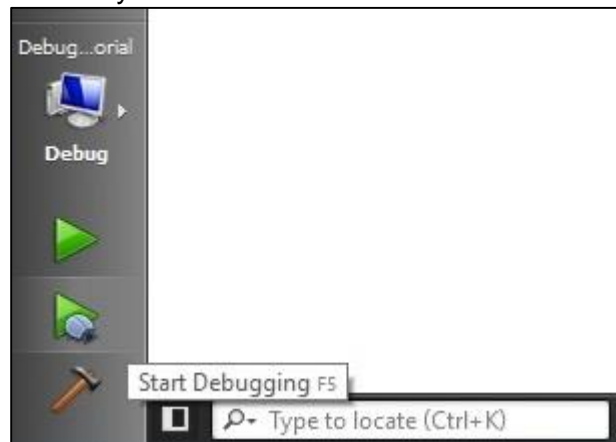


Figure 2

To set a breakpoint, find the line you wish to pause at, then click once to the left of that line number in the gray area. A red circle should appear where you click which shows that a breakpoint has been placed at this line. For example, in Figure 3, a breakpoint is set at line 17.

```
14     setConsoleFont ("Courier New-16");
15     setConsoleEcho (true);
16
17     Vector<ScoreEntry> scoreboard;
18     fillInScoresFromUser (scoreboard);
19
20     return 0;
21 }
```

Figure 3

Now when you start debugging and the execution reaches this breakpoint, it will pause execution and you should see a yellow arrow pointing at the line which is about to be executed. In Figure 4, execution has reached the breakpoint at line 17.

```

15     setConsoleEcho(true);
16
17     Vector<ScoreEntry> scoreboard;
18     fillInScoresFromUser(scoreboard);
19
20     return 0;

```

Figure 4

Now that execution has been stopped, you can **examine the value and contents of your variables by looking at the right side of the window (in Debug view) where you should see a list of your variables and their values like in Figure 5.**

Name	Value	Type
▲ scoreboard	@0x28fe28	Vector<ScoreEntry>
▶ [vptr]	0x28fe48	
capacity	3997696	int
count	0	int
▶ elements	@0x6fca18f2	ScoreEntry

Figure 5

In Figure 5, you can see the value and the type of the scoreboard variable. The value might be a little confusing. What does “@0x28fe28” mean? Since scoreboard is a Vector object, the value of the scoreboard variable is the location in memory where that object is stored. Thus 0x028fe28 is the memory location in your computer where the scoreboard object is stored. This information doesn’t help you very much so you can ignore it. In addition to the location of the object, the debugger will also show you the values of the variables declared inside this object. For instance, a Vector object has a count variable which tracks how many elements are in the Vector. We can see in Figure 4 that the scoreboard vector has 0 elements since the value of count is 0. This is an example of the helpful information that we can find in the variable description window on the right side of the debug view in Qt Creator. Note that **if a variable has an arrow (▶) to the left of its name that means that you can click on that arrow to show more information about that variable.** When you are debugging your programs, make sure that the values in the variable window make sense and correctly correspond to what your program should be doing. Any irregularities could be a sign of bugs in your program.

After execution has paused at a breakpoint, you can advance execution two different ways. First, you can use the step over, step into, and step out buttons to go line by line through your program. These buttons can be found below the code and above the application output window and in Figure 6 below. The step over button will execute the current line of code and then move to the next line of code. The step over button does not care if the current line is a function call or not. Usually when stepping through your code, you will use the step over button. If you want to be able to go into a function call and step through that function itself, you can click the step into button when execution is paused on a line which includes a function call and then the execution will jump to the beginning of that function. If you are inside of a function and you would like to execute the rest of that function and then return to where the function was called, you can click the step out button to do that. Second, if you would like to continue execution of the program until the next breakpoint or until the program stops, you can use the continue

button which is to the left of the step buttons as shown in Figure 6. Lastly, if you want to stop the execution of the program you can click the terminate button which is also shown in Figure 6.

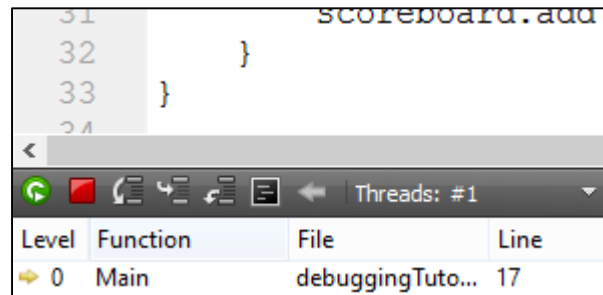


Figure 6

In Figure 6, the execution control buttons are as follows. The green button with a circular arrow on it is the continue button. The red square button is the terminate button. After the terminate button, from left to right, there is the step over button, the step into button and the step over button. These 5 buttons will help you control execution while debugging.

Another useful tool to use when in debug view in Qt Creator is the stack trace window which is right below the execution control button. You can see the top of the stack trace window which contains the yellow arrow in Figure 5. This window will have a yellow arrow pointing to the current function that the program execution is in. It will have the function name, what code file the function is defined in and the exact line of the function that the execution is on. Below the current function, it will list the parent function of the current function. This is the function from which the current function was called. For example, if functionA() calls functionB() on line 28 and there is a breakpoint in functionB() at line 56, then the stack trace window will show functionB() at line 56 at the top of the window followed by functionA() at line 28 below it. This is what we call the stack trace which is the list of functions that have been called but haven't returned yet. The stack trace gives us valuable information about how our program got to the current line. This can be helpful when there are multiple calls to a certain function in your program and you need to figure out which functions were called to get to the current line of the program.

All of the tools mentioned above will help improve your debugging skills. Now that you understand how to use breakpoints, the debug view, the stack trace window, the execution control buttons, the variable windows and more, you are now more empowered to find and solve those mysterious bugs that will attempt to plague your programs.

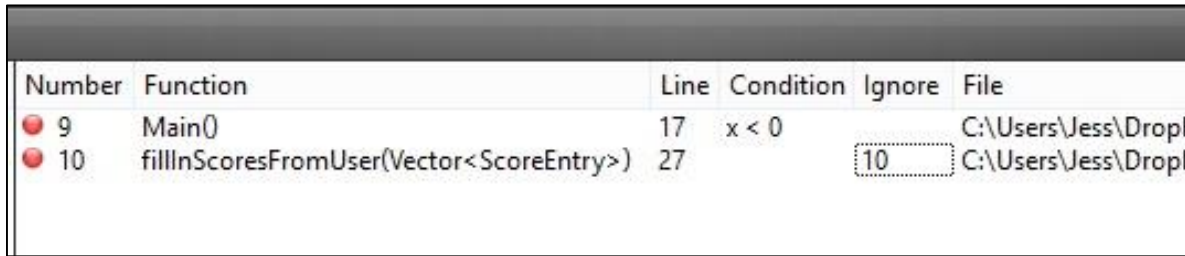
Advanced Breakpoint Settings

Quick Summary

- Uses: stopping after certain number of loop iterations or when some condition is true
- How: edit breakpoint settings to change the condition and ignore fields of a breakpoint
 - Condition: if condition is true when breakpoint is reached, it will pause execution.
 - Ignore: it will ignore breakpoint this many times before pausing execution.

Detailed Walkthrough

There are times when you'll notice that a bug only appears in your program under certain conditions or after a certain amount of time. Breakpoints normally trigger any time that line is reached during program execution. However, it is possible to edit the settings for a breakpoint so that it only pauses execution in certain situations. To reach these settings, first create a breakpoint in the normal way as described in the above section. After a breakpoint has been created, it will show up in the breakpoint window which is in the lower right of debug view as seen in Figure 7.



Number	Function	Line	Condition	Ignore	File
9	Main()	17	x < 0		C:\Users\Jess\Dropk
10	fillInScoresFromUser(Vector<ScoreEntry>)	27		10	C:\Users\Jess\Dropk

Figure 7

In the breakpoint window, each breakpoint in the program is described by its line number, the location of the breakpoint in the program (function name and line number), and something called the condition and ignore fields. What are these?

The *Condition* field is a boolean statement that controls whether the breakpoint activates or not when reached by program execution. In Figure 7, breakpoint #9 only activates if its condition, “x < 0”, is true. Therefore, when the program reaches line 17, if x < 0, then the execution will pause at this breakpoint. Otherwise, the breakpoint will be ignored and execution will continue. This can be helpful if you notice that problems only happen in certain situation so you only want to pause execution if one of those situations occur. For example, if a bug only seems to occur when the Vector named scores is empty, then you can create a breakpoint with the condition “scores.isEmpty()”.

The *Ignore* field is a count of how many times the breakpoint should be ignored before it starts to activate. In Figure 7, breakpoint #10 has an ignore count of 10 which means the program execution will not pause at this breakpoint the first 10 times execution reaches this line then will pause execution when this line is reached every time afterwards. This can be helpful if you have a breakpoint inside of a loop but you don't want it to pause execution until a certain number of loop iterations have passed.

In order **to set these advanced settings for a breakpoint, right click on the desired breakpoint in the breakpoint window which will bring up a menu of options as seen in Figure 8. Click the ‘Edit Breakpoint...’ option (3rd from top) to open the “Edit Breakpoint Properties” window as seen in Figure 9.**

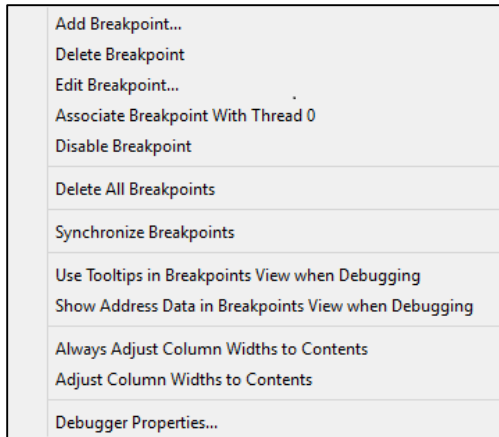


Figure 8

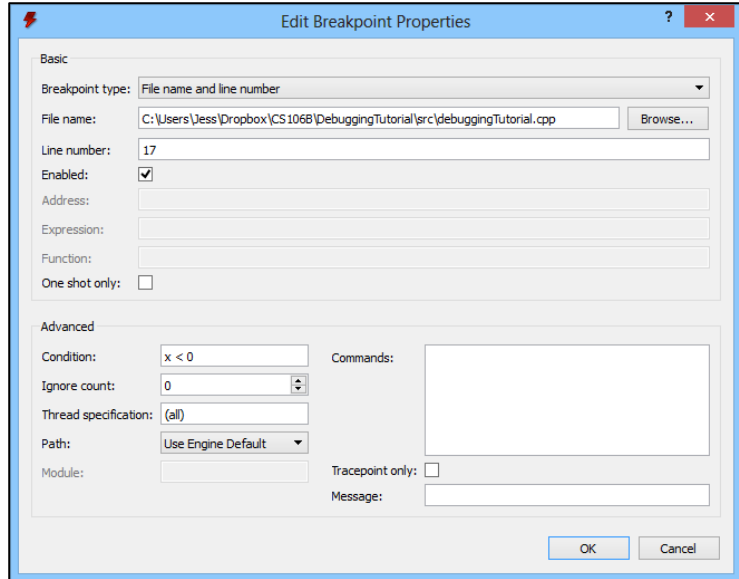


Figure 9

In the “Edit Breakpoint Properties” window, you can edit the condition field and the ignore field for this breakpoint. When you are done editing the settings, click ok to change the settings and close the window.

How do I view multiple elements in an array? (Especially helpful for looking at Vectors)

Unfortunately Qt Creator does not always show you the most helpful information in the variable value window when debugging especially when it comes to arrays. For example, Figure 9 shows you the default view when looking at a Vector of integers containing 2 elements.

Name	Value	Type
▶ display	@0x28f5c8	LifeDisplay
▶ grid	@0x28f618	Grid<int>
▲ scoreboard	@0x28f608	Vector<int>
▶ [vptr]	0x435d38	
capacity	2	int
count	2	int
elements	5	int

Figure 10

In Figure 10, you can see that this isn’t very helpful at all. It tells you that the elements field is just an integer (the first value in the vector actually) when in fact it’s an array of the values inside the vector. These are the values we want to see. This isn’t very helpful since the count field shows that this Vector has 2 elements. How can you see the other elements? If you right click on the “elements” line, then hover over the “Change Local Display Format...” option in the initial menu which will bring up a pop-up menu. Then in this pop-up menu, select “Array of 10 items” under the Change Display for Object Named “elements”: section. This series of operations is shown in Figure 11.

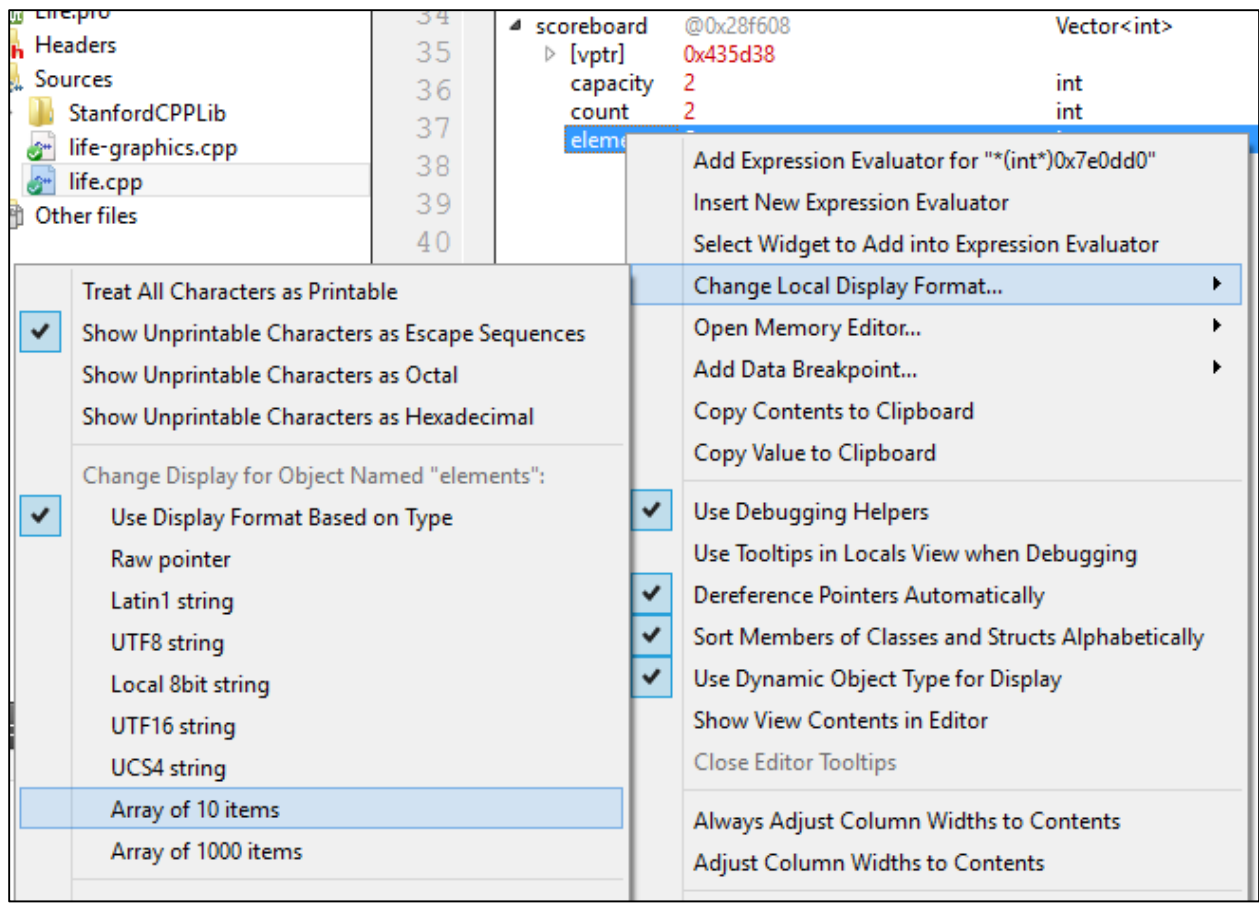


Figure 11

Once you've changed this setting for the elements array field of this Vector object, Qt Creator will display 10 elements for the array as seen in Figure 12. However, there may not be 10 elements in the array. Therefore, make sure to look at the count field of the Vector to see how many elements are in the Vector and do not look at more than that many elements since any elements shown past that number are junk elements and not valid.

Name	Value	Type
display	@0x28f5c8	LifeDisplay
grid	@0x28f618	Grid<int>
scoreboard	@0x28f608	Vector<int>
[vptr]	0x435d38	
capacity	2	int
count	2	int
elements	<10 items>	int *
[0]	5	int
[1]	2	int
[2]	-1414812757	int
[3]	-1414812757	int
[4]	-17891602	int
[5]	-17891602	int

Figure 12