

1.11a — Debugging your program (watching variables and the call stack)

BY ALEX ON NOVEMBER 21ST, 2007 | LAST MODIFIED BY ALEX ON APRIL 10TH, 2018

In the [previous lesson on stepping and breakpoints](#), you learned how to use the debugger to watch the path of execution through your program. However, stepping through a program is only half of what makes the debugger useful. The debugger also lets you examine the value of variables as you step through your code.

Our examples here will be using the Visual Studio Express debugger -- if you are using a different IDE/debugger, the commands may have slightly different names or be located in different locations.

Before proceeding: Make sure your program is set to use the [debug build configuration](#).

Watching variables

Watching a variable is the process of inspecting the value of a variable while the program is executing in debug mode. Most debuggers provide several ways to do this. Let's take a look at a sample program:

```
1  #include "stdafx.h"
2  #include <iostream>
3
4  int main()
5  {
6      int x =1;
7      std::cout << x << " ";
8
9      x = x + 1;
10     std::cout << x << " ";
11
12     x = x + 2;
13     std::cout << x << " ";
14
15     x = x + 4;
16     std::cout << x << " ";
17
18     return 0;
19 }
```

This is a pretty straightforward sample program -- it prints the numbers 1, 2, 4, and 8.

First, use the “Run to cursor” command to debug to the first `std::cout << x << " ";` line:

```

#include "stdafx.h"
#include <iostream>

int main()
{
    int x(1);
    std::cout << x << " ";

    x = x + 1;
    std::cout << x << " ";

    x = x + 2;
    std::cout << x << " ";

    x = x + 4;
    std::cout << x << " ";

    std::cout << std::endl;

    return 0;
}

```

At this point, the variable `x` has already been created and initialized with the value 1, so when we examine the value of `x`, we should expect to see the value 1.

The easiest way to examine the value of a simple variable like `x` is to hover your mouse over the variable `x`. Most modern debuggers support this method of inspecting simple variables, and it is the most straightforward way to do so.

```

#include "stdafx.h"
#include <iostream>

int main()
{
    int x(1);
    std::cout << x << " ";

    x = x + 1;
    std::cout << x << " ";

    x = x + 2;
    std::cout << x << " ";

    x = x + 4;
    std::cout << x << " ";

    std::cout << std::endl;

    return 0;
}

```

Note that you can hover over any variable `x`, not just the one on the current line:

```
#include "stdafx.h"
#include <iostream>

int main()
{
    int x(1);
    std::cout << x << " ";

    x = x + 1;
    std::cout << x << " ";

    x = x + 2;
    std::cout << x << " ";

    x = x + 4;
    std::cout << x << " ";

    std::cout << std::endl;

    return 0;
}
```

If you're using Visual Studio, you can also use QuickWatch. Highlight the variable name `x` with your mouse, and then choose "QuickWatch" from the right-click menu.

```
#include "stdafx.h"
#include <iostream>

int main()
{
    int x(1);
    std::cout << x << " ";

    x = x + 1;
    std::cout << x << " ";

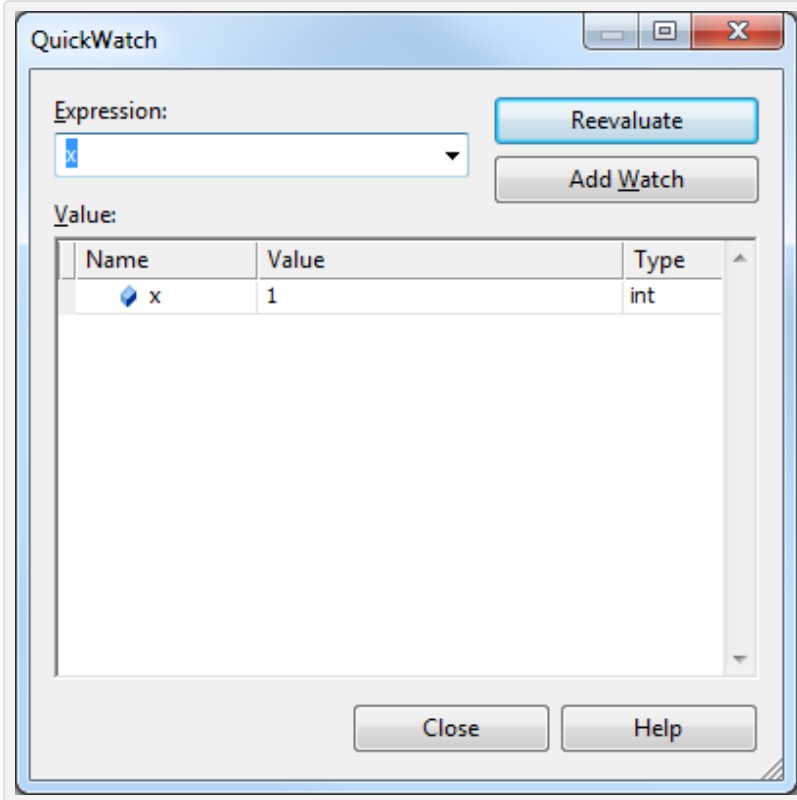
    x = x + 2;
    std::cout << x << " ";

    x = x + 4;
    std::cout << x << " ";

    std::cout << std::endl;

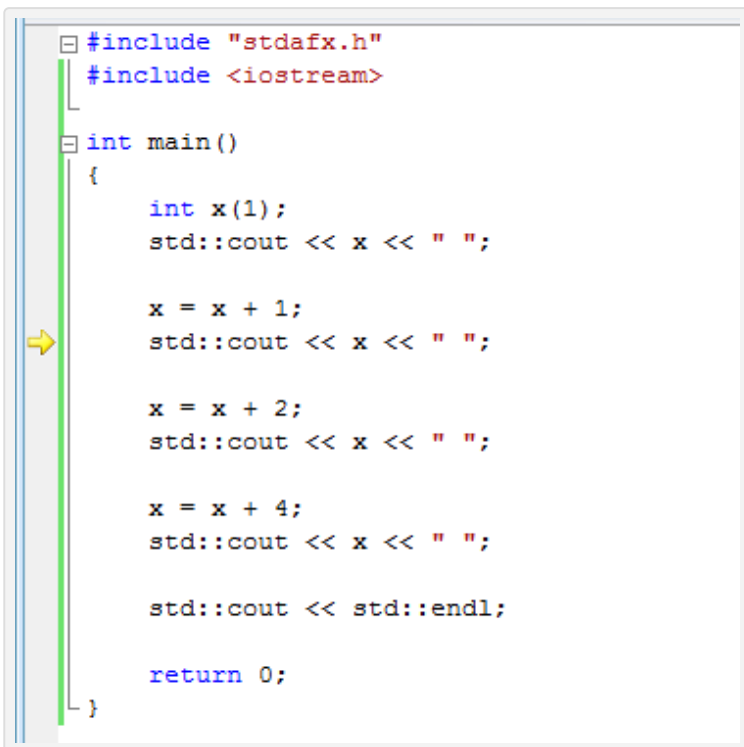
    return 0;
}
```

This will pull up a subwindow containing the current value of the variable:



Go ahead and close QuickWatch if you opened it.

Now let's watch this variable change as we step through the program. First, choose "Step over" twice, so the next line to be executed is the second `std::cout << x << " ";` line:



The variable `x` should now have value 2. Inspect it and make sure that it does!

The watch window

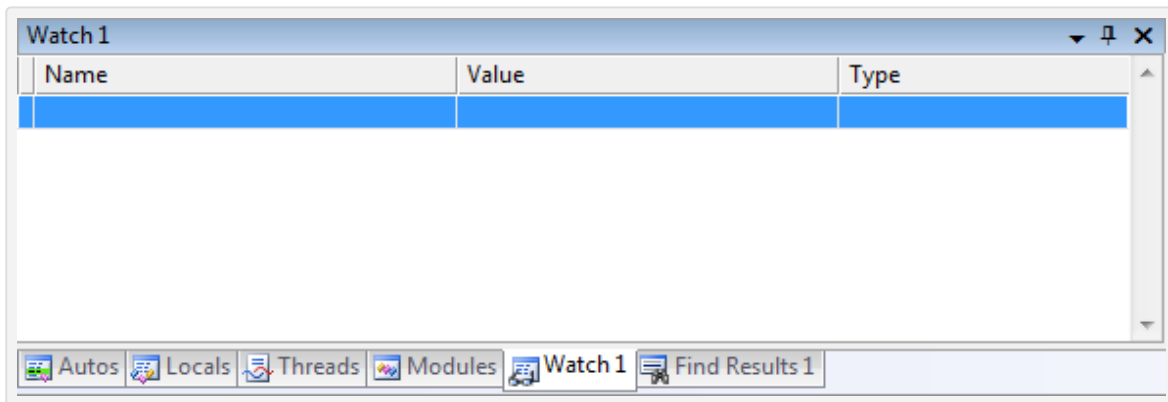
Using the mouse hover or QuickWatch methods to inspect variables is fine if you want to know the value of a variable at a particular point in time, but it's not particularly well suited to watching the value of a variable change as you run the code because you continually have to rehover/reselect the variable.

In order to address this issue, all modern IDEs provide another feature, called a watch window. The **watch window** is a window where you can add variables you would like to continually inspect, and these variables will be updated as you step through your program.

The watch window may already be on your screen when you enter debug mode, but if it is not, you can bring it up through your IDEs window commands (these are typically found in the view or debug menus).

In Visual Studio 2005 Express, you can bring up a watch menu by going to Debug Menu->Windows->Watch->Watch 1 (note: you have to be in debug mode, so step into your program first).

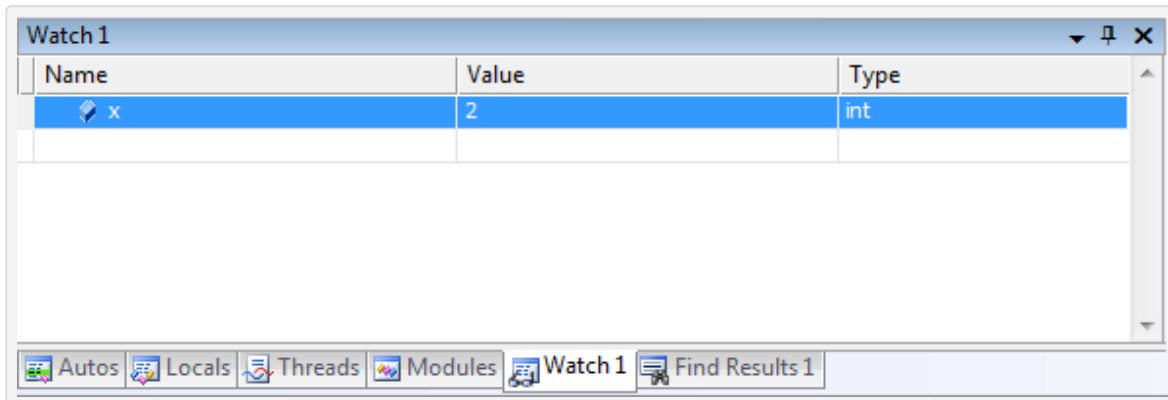
You should now see this:



There is nothing in this window because we have not set any watches yet. There are typically two different ways to set watches:

- 1) Type in the name of the variable you would like to watch in the "Name" column of the watch window.
- 2) Highlight the variable you would like to watch, right click, and choose "Add Watch".

Go ahead and add the variable "x" to your watch list. You should now see this:



Now chose the "Step over" command a few times and watch the value of your variable change!

Note that variables that go out of scope (e.g. variables declared in a function, when the function is not running) will stay in your watch window. If the variable returns to scope (e.g. the function is called again), the watch will pick it up and begin showing its value again.

Using watches is the best way to watch the value of a variable change over time as you step through your program.

The call stack window

Modern debuggers contain one more debugging information window that can be very useful in debugging your program, and that is the call stack window.

When your program calls a function, you already know that it bookmarks the current location, makes the function call, and then returns. How does it know where to return to? The answer is that it keeps track in the call stack.

The **call stack** is a list of all the active functions that have been called to get to the current point of execution. The call stack includes an entry for each function called, including what line was actively executing. Whenever a new function is called, that function is added to the top of the call stack. When the current function returns to the caller, it is removed from the top of the call stack, and control returns to the function just below it.

If you don't see the call stack window, you will need to tell the IDE to show it. In Visual Studio 2005 Express, you can do that by going to Debug Menu->Windows->Call Stack (note: you have to be in debug mode, so step into your program first).

Let's take a look at the call stack using a sample program:

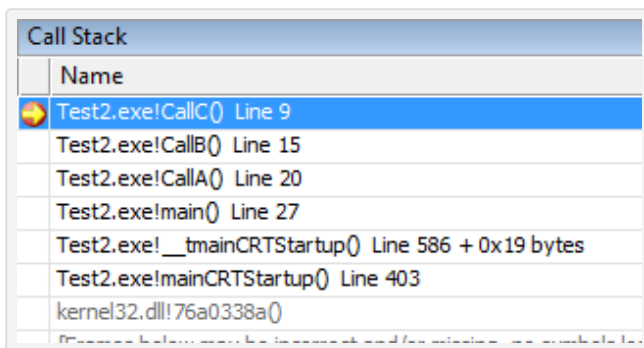
```

1  #include "stdafx.h"
2  #include <iostream>
3
4  void CallC()
5  {
6      std::cout << "C called" << std::endl;
7  }
8
9  void CallB()
10 {
11     std::cout << "B called" << std::endl;
12     CallC();
13 }
14
15 void CallA()
16 {
17     CallB();
18     CallC();
19 }
20
21 int main()
22 {
23     CallA();
24
25     return 0;
26 }

```

Put a breakpoint in the CallC() function and then start debugging mode. The program will execute until it gets to the line you breakpointed.


Although you now know the program is executing CallC(), there are actually two calls to CallC() in this program (one in CallB(), and one in CallA()). Which function was responsible for calling CallC() this time? The Call stack will show us:



(Note that your line numbers may differ slightly, as may the functions that appear in the call stack beneath main())

The program started by calling main(). main() called CallA(), which called CallB(), which called CallC(). You can double-click on the various lines in the Call Stack window to see more information about the calling functions. Some IDEs take you directly to the function call. Visual Studio 2005 Express takes you to the next line after the function call. Go ahead and try this functionality out. When you are ready to resume stepping through your code, double-click the top line of the call stack window and you will return to your point of execution.

Continue running your program. Your breakpoint should be hit a second time when CallC() is called a second time (this time, from CallA()). You should see this reflected in the call stack:

Call Stack	
	Name
	Test2.exe!CallC() Line 9
	Test2.exe!CallA() Line 21
	Test2.exe!main() Line 27
	Test2.exe!__tmainCRTStartup() Line 586 + 0x19 bytes
	Test2.exe!mainCRTStartup() Line 403
	kernel32.dll!76a0338a()
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dll!76a0338a()]	

Conclusion

Congratulations, you now know the basics of debugging your code! Using stepping, breakpoints, watches, and the call stack window, you now have the fundamentals to be able to debug almost any problem. Like many things, becoming good at using a debugger takes some practice and some trial and error. However, the larger your programs get, the more valuable you will find the debugger to be, so it is definitely worth your time investment!



[1.12 -- Chapter 1 comprehensive quiz](#)



[Index](#)



[1.11 -- Debugging your program \(stepping and breakpoints\)](#)

Share this:



[C++ TUTORIAL](#) | [PRINT THIS POST](#)

75 comments to 1.11a — Debugging your program (watching variables and the call stack)

[« Older Comments](#) [1](#) [2](#)

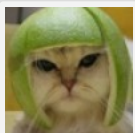


Virgi

[June 19, 2018 at 8:32 am](#) · [Reply](#)

Hi Alex,

I would like to know why when I put a breakpoint in codeblocks and run debug it doesn't stop there, it runs all the way through the program and ends with "Debugger finished with status 0"
Thank you so much for teaching us cpp!



Alex

[June 29, 2018 at 12:41 pm](#) · [Reply](#)

Are you sure the breakpoint is actually being hit? If the code never executes the line that is breakpointed, the breakpoint won't trigger.