3.8 — Using an integrated debugger: Watching variables

BY ALEX ON NOVEMBER 21ST, 2007 | LAST MODIFIED BY ALEX ON AUGUST 13TH, 2019

In the previous lessons (**3.6 -- Using an integrated debugger: Stepping** and **3.7 -- Using an integrated debugger: Running 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, all without having to modify your code.

As per previous lessons, our examples here will use Visual Studio -- if you are using a different IDE/debugger, the commands may have slightly different names or be located in different locations.

Warning

In case you are returning, make sure your project is compiled using a debug build configuration (see **0.9 -- Configuring your compiler: Build configurations** for more information). If you're compiling your project using a release configuration instead, the functionality of the debugger may not work correctly.

Watching variables

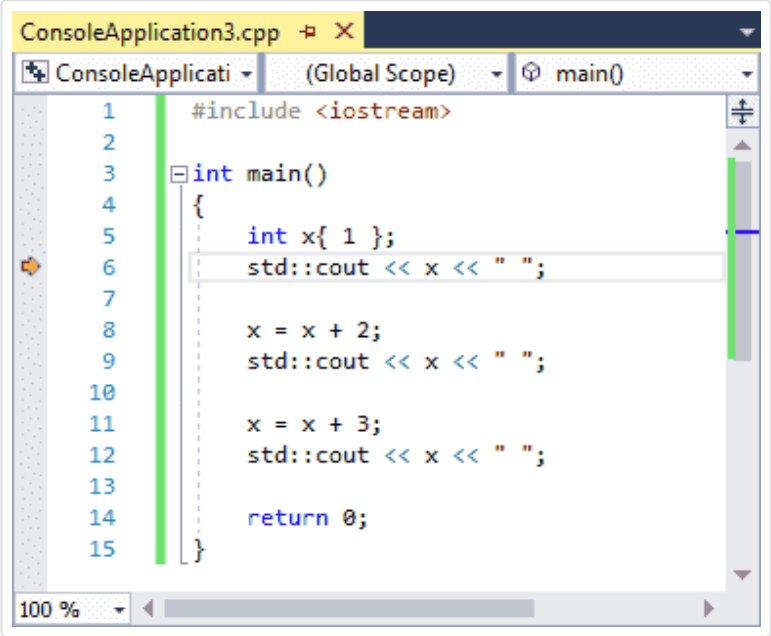
Watching a variable is the process of inspecting the value of a variable while the program is executing in debug mode. Most debuggers provide several ways to do this.

Let's take a look at a sample program:

```
#include <iostream>

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

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

    x = x + 3;
    std::cout << x << " ";

    return 0;
}
```

This is a pretty straightforward sample program -- it prints the numbers 1, 3, and 6.

First, *run to cursor* to line 6.

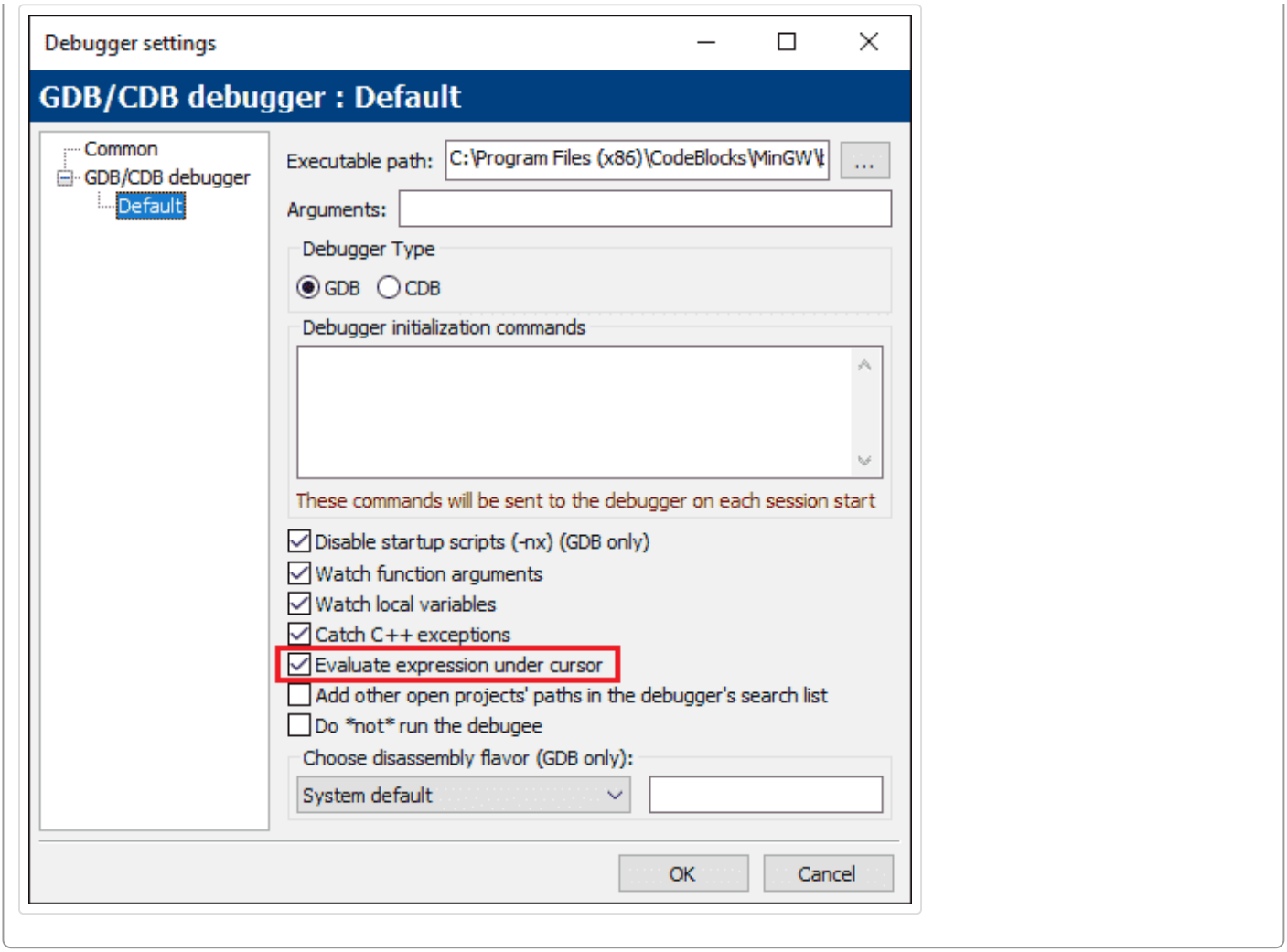


At this point, the variable x has already been created and initialized with the value 1, so when we examine the value of x, we should expect to see the value 1.

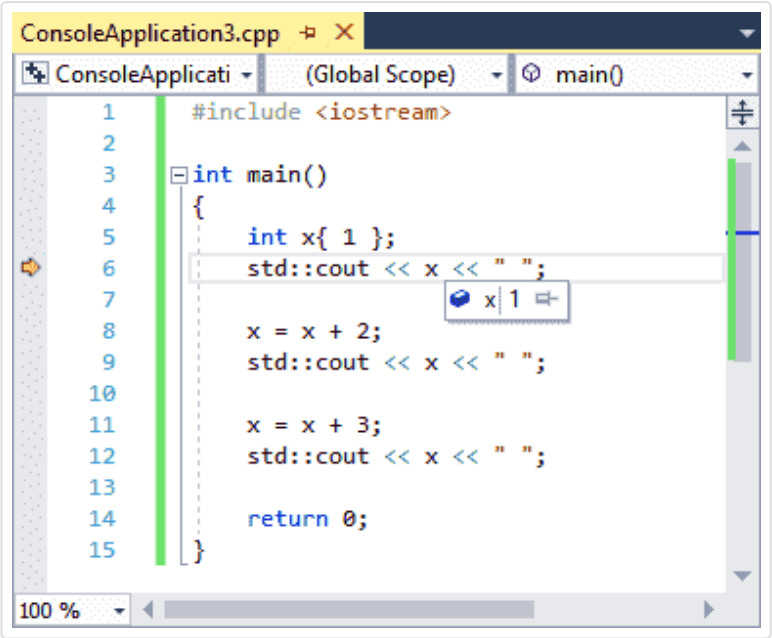
The easiest way to examine the value of a simple variable like x is to hover your mouse over the variable x. Some modern debuggers support this method of inspecting simple variables, and it is the most straightforward way to do so.

For Code::Blocks users

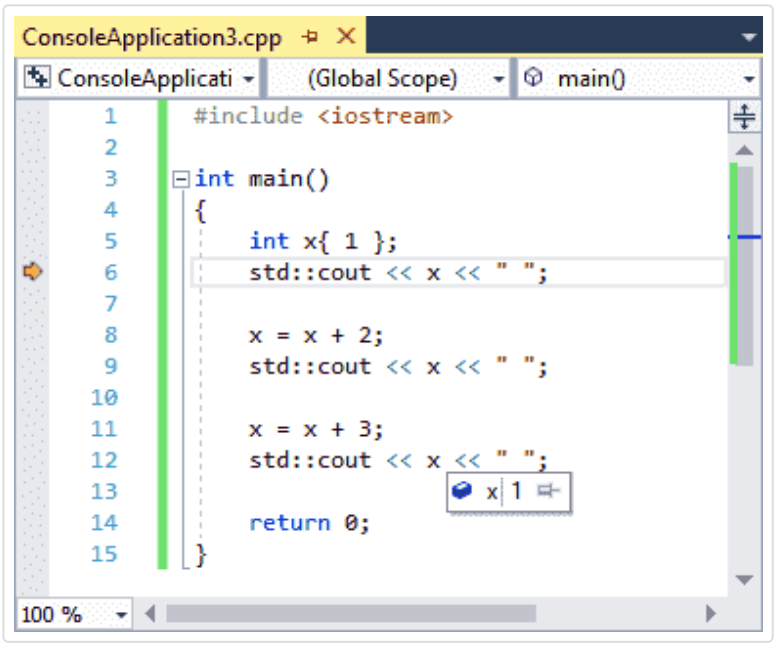
If you're using Code::Blocks, this option is (inexplicably) off by default. Let's turn it on. First, go to *Settings menu* > *Debugger...*. Then under the *GDB/CDB debugger node*, select the *Default* profile. Finally, check the box labeled *Evaluate expression under cursor*.



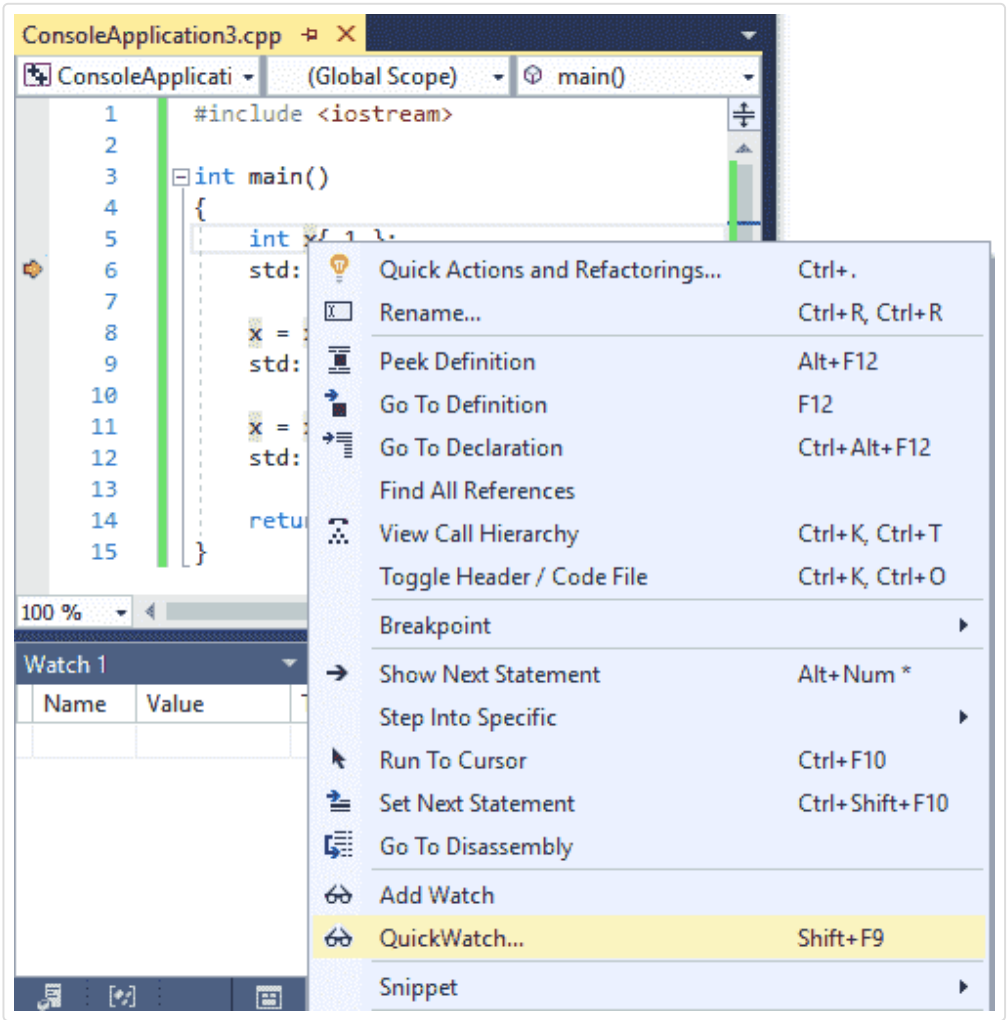
Hover your mouse cursor over variable x on line 6, and you should see something like this:



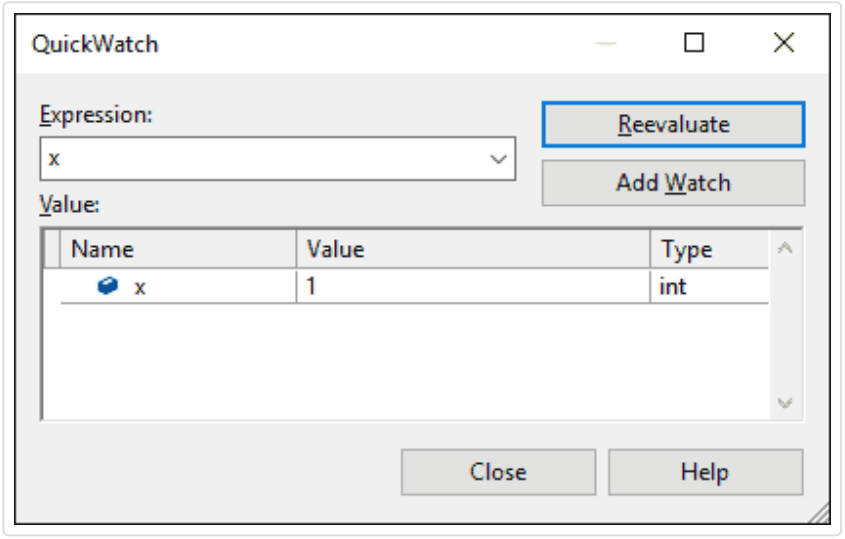
Note that you can hover over any variable x, not just the one on the current line. For example, if we hover over the x on line 12, we'll see the same value:



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.

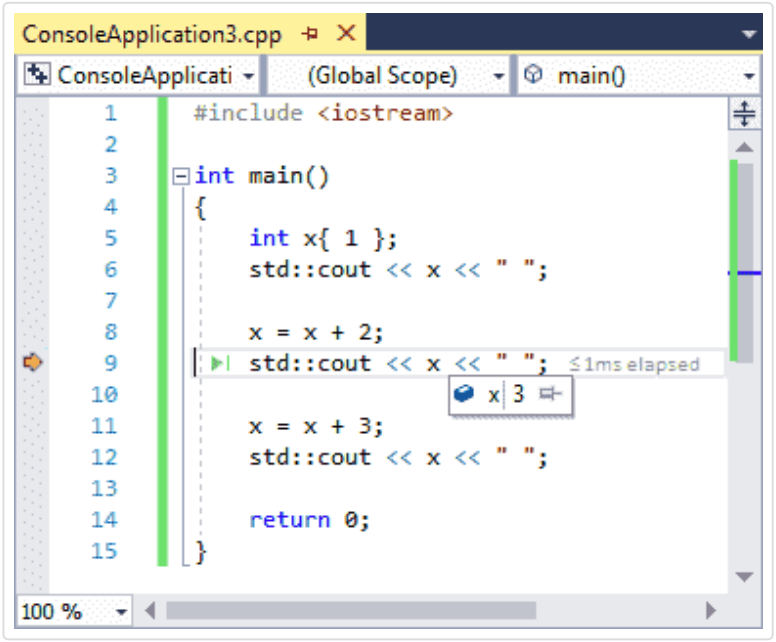


This will pull up a subwindow containing the current value of the variable:



Go ahead and close QuickWatch if you opened it.

Now let's watch this variable change as we step through the program. Either choose *step over* twice, or *run to cursor* to line 9. The variable x should now have value *3*. 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 integrated debuggers 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 IDE's window commands (these are typically found in a View or Debug menu).

For Visual Studio users

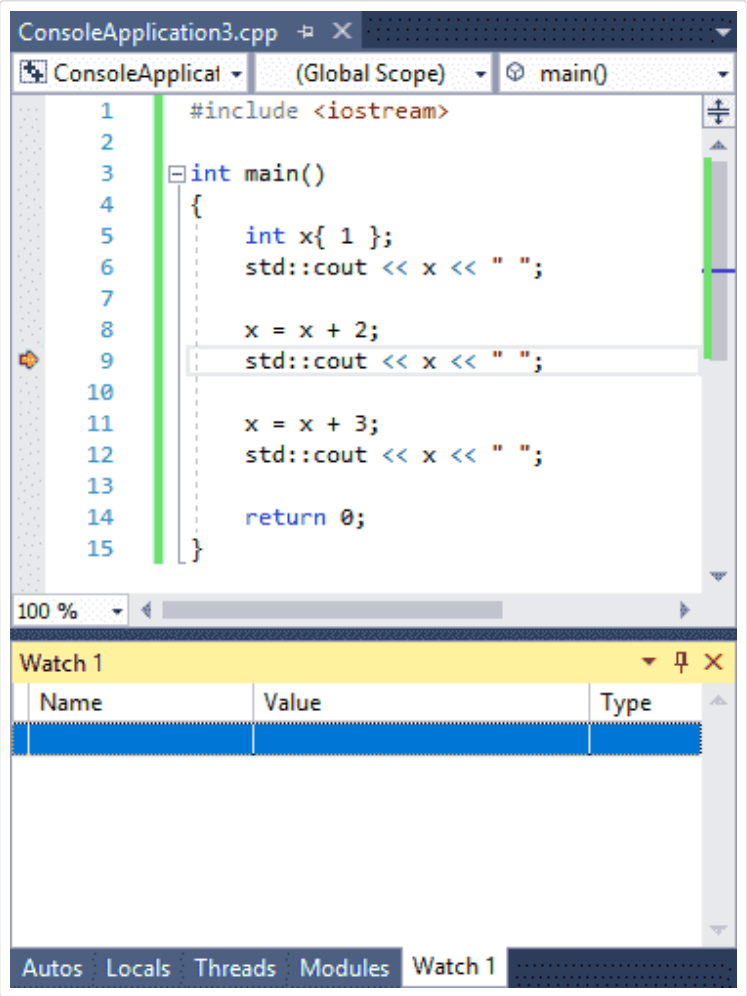
In Visual Studio, the watch menu can be found at *Debug menu* > *Windows* > *Watch* > *Watch 1*. Do note that you have to be in debug mode for this option to be enabled, so *step into* your program first.

Where this window appears (docked left, right, or bottom) may vary. You can change where it is docked by dragging the *Watch 1* tab to a different side of the application window.

For Code::Blocks users

In Code::Blocks, the watch menu can be found at *Debug menu* > *Debugging windows* > *Watches*. This window will likely appear as a separate window. You can dock it into your main window by dragging it over.

You should now see something like this:



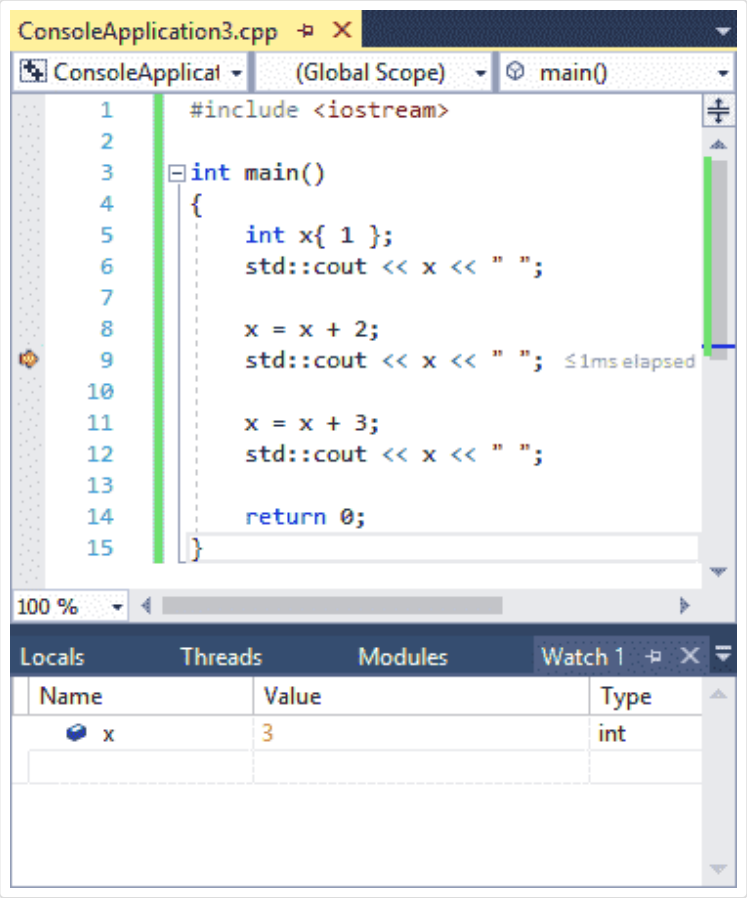
The watches window may or may not contain anything in it already.

There are typically two different ways to add variables to the watch window:

1. Pull up the watch window, and type in the name of the variable you would like to watch in the leftmost column of the watch window.
2. In the code window, right click on the variable you'd like to watch, and choose *Add Watch* (Visual Studio) or *Watch x* (replace x with the variable's name) (Code::Blocks).

If you're not already in a debugging session with the execution marker on line 9 of your program, start a new debugging session and *run to cursor* to line 9.

Now, go ahead and add the variable “x” to your watch list. You should now see this:



Now *step over* twice, or *run to cursor* to line 12, and you should see the the value of *x* change from *3* to *6*.

Variables that go out of scope (e.g. a local variable inside a function that has already returned to the caller) will stay in your watch window, but will generally either be marked as “not available”, or may show the last known value but grayed out. If the variable returns to scope (e.g. the function is called again), its value will begin showing again. Therefore, it’s fine to leave variables in the watch window, even if they’re out of scope.

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

The watch window can evaluate expressions too

The watch window will also allow you to evaluate simple expressions. If you haven’t already, *run to cursor* to line 12. Then try entering *x + 2* into the watch window and see what happens (it should evaluate to 8).

You can also highlight an expression in your code and then inspect the value of that expression via hover or by adding it to the watch window via the right-click context menu.

Warning

Identifiers in watched expressions will evaluate to their current values. If you want to know what value an expression in your code is actually evaluating to, *run to cursor* to it first, so that all identifiers have the correct values.

Local watches

Because inspecting the value of local variables inside a function is common while debugging, many debuggers will offer some way to quickly watch the value of *all* local variables in scope.

For Visual Studio users

In Visual Studio, you can see the value of all local variables in the *Locals* window, which can be found at at *Debug menu* > *Windows* > *Locals*. Note that you have to be in a debug session to activate this window.

For Code::Blocks users

In Code::Blocks, this is integrated into the *Watch* window, under the *Locals* node. If you don't see any, there either aren't any, or you need to uncollapse the node.

If you're just looking to watch the value of a local variable, check the *locals* window first. It should already be there.


3.9 -- Using an integrated debugger: The call stack


Index


3.7 -- Using an integrated debugger: Running and breakpoints

C++ TUTORIAL | PRINT THIS POST

96 comments to 3.8 — Using an integrated debugger: Watching variables

« Older Comments 1 2

HolzstockG
November 23, 2019 at 10:23 pm · Reply

Why after stepping "return" statement do we still have to go through function's ending curly brace? I am really curious about that. Shouldn't function just close after return? I did try to put some lines of code under return and as it should it did avoid them but not curly brace.

Alex
November 25, 2019 at 10:02 pm · Reply

I'm not totally sure, but my guess is that post-function cleanup is happening at this point (e.g. popping the frame off the call stack). It would make sense to have all this code at one place (at the end of the function) rather than associated with each potential return point.

Bruno
August 11, 2019 at 9:16 pm · Reply

Hey!

I think code blocks is getting to close to Visual Studios at the end of this lesson(Local watches). XD. Naughty code blocks.

Alex
August 13, 2019 at 10:48 am · Reply

Hah! Yes, it appears they got a bit cozy. They've been separated and instructed to leave their door open at least 3 inches at all times.

Joe
May 20, 2019 at 3:11 am · Reply

Hi, just some nitpicking.

Near the end of section Watching variables, it says to "step over 3 times" in order to get to line9 when in fact only 2 step overs are needed.

Edit: similar thing occurs under The watch window. Says "step over 3 times" when only two are needed

In the examples, we were asked to move exactly 3 lines down hence why we were asked to step over 3 times. However, between the statements is a blank line so only 2 steps really are needed

Anyway, really appreciate the guide :D

Alex
May 20, 2019 at 10:20 am · Reply

Thanks for pointing this out. The lesson has been amended.

Len
March 21, 2019 at 6:31 am · Reply

Just leaving a quick suggestions:

In Visual Studios Community 2017 the option to show the Locals window in Debug > Windows > Locals is only available when you are currently in a debug session. In the next tutorial you mention this for the call stack but here you do not. You might consider adding that in to avoid someone being confused.

Alex
March 23, 2019 at 8:10 pm · Reply

Lesson amended. Thanks for the callout!

AA
January 21, 2019 at 6:46 am · Reply

Hi there,
Thanks you all for publishing these awesome tutorials. I'd love them.
I recommend to you that add Qt Creator in them, Because it's really greater than Code::Blocks and you know it's cross-platform.

I have a problem about initializing the variable 'x'.
As I said I use Qt Creator 5.7 on Mint(GNU/Linux), but when I'm using "Run to cursor" command to debug first

```
1 std::cout << x << " ";
```

, the value X is initialized with the value 0 !.
What's the matter ?
I know there are differences between compilers but in this case I don't know anymore !
Thank you so much ;)

Alex
January 23, 2019 at 9:30 pm · Reply

I'll have to check out Qt Creator again.

I have no idea when you're getting a value of zero when the previous line explicitly initializes the variable with value 1.

Alireza
January 25, 2019 at 9:27 am · Reply

Thank you, I hope it will be in these tutorials. It's awesome.
How long is it ?

YK
January 9, 2019 at 8:16 pm · Reply

Hi Alex,

Thank you so much for putting this tutorials out. They are awesome!
I know it takes a lot of time and thoughts to put all this information together.
And it's not about just putting the information out, but putting it out in a manner so that it's easy to understand.
Your concise explanation supported with examples makes it very easy to follow along.
Thanks again for taking time to spread the knowledge!

Android
November 26, 2018 at 3:42 am · Reply

Thank you for this blog..This is really very helpful for me as I am preparing for courses in Development..I used to prefer your blog since long time for keeping me updates..looking forward for new blogs..
ALL THE BEST

Derick
November 18, 2018 at 4:17 am · Reply

Just want to check whether I missed something or it hasn't been covered.

In the first block of example code x is initialized with the line

```
1 int x =1;
```

But in all the subsequent versions of the same program it's shown as

```
1 int x(1);
```

Is this a different syntax for declaring and initializing a variable, or is this something visual studio does during debuging mode? I don't recall it being in any of the earlier tutorials.

nascardriver
November 18, 2018 at 4:28 am · Reply

Hi Derick!

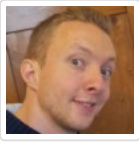
The first version is copy initialization, the second direct initialization. They, along with the preferred uniform initialization, are discussed in lesson 2.1.

Lazar
November 14, 2018 at 9:05 am · Reply

Hi Alex :) Just wanted to let you know before the call stack window headline you wrote now chose, it should be now choose.

Alex
November 16, 2018 at 3:58 pm · Reply

It always surprises me how long some of these typos have survived in the wild. Thanks for catching this one!



Nick Boots
August 20, 2018 at 3:46 am · Reply

I'm a bit confused by the differing call stack output. You mentioned the outputs can differ, but they point at quite the different positions in the example above and in my program.

Both my Script and the one in the example match in line numbers. With mine being done in VS 2017 Pro.

In yours Main and CallA() are marked at the line after the closing bracket, in mine a line after each Function that performs the call. (Line 25 and 18 respectively.) Finally, the last two calls are on line 13 for CallB() and line 5 for CallC() for me.

I'm a bit confused as to why it differs so much and would like to understand it, did something change?

Alex
August 22, 2018 at 12:14 pm · Reply

Yeah, I think the example evolved and the screenshots didn't. I'll flag this for an update. Thanks for pointing out the inconsistency!

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.

Jack
February 6, 2019 at 7:26 am · Reply

In case anyone still has this problem....I had it using code::blocks, ie not stopping at breakpoints (although it was possible to step through the code with 'step into') After spending a day trying various ideas I found I had been too profligate in my naming of the project directory -- using too many spaces and non-ascii characters. Changing to simpler file names cured it. I subsequently found this post. http://forums.codeblocks.org/index.php?topic=19476.0

Alex
February 7, 2019 at 8:01 pm · Reply

Thanks. I added a note to a prior lesson about this, in case anybody else runs into the issue.

Louis Cloete
May 23, 2018 at 5:54 pm · Reply

error line [below 2nd screenshot of watch window]: present tense verb "choose" expected, past tense verb "chose" found ;-)

himanshu
April 7, 2018 at 7:58 pm · Reply

Hi Alex!
I unable to understand what call stack try to say
I am using code::block
It is completely different than your's call stack
and unfortunately i unable to copy it so i can not paste it but the first line is:

0x4010fd _mingw_CRTSartup()

Alex
April 10, 2018 at 10:11 pm · Reply

It's okay if the bottom of your call stack is different than mine. Everything from main() and above should be the same though.

Audrius
December 16, 2017 at 8:01 am · Reply

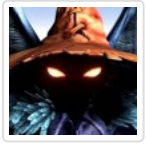
Hello,
Is there any way to watch array values inside a function like this one?

```
void skaic(int (Mok)[Cmax], int (Slyv)[Cmax])
{
    for(int j=0;j<10;j++)
    {
        slink(Mok,j);
        {
            for(int i=j;i<Cmax;i++)
            {
                if(Mok[i]!=0)
                {
                    Slyv[i]+=1;
                    Mok[i]-=1;
                }
            }
        }
    }
}
```

Thanks, keep up the good work :)

Alex
December 16, 2017 at 6:37 pm · Reply

Yes, just put a watch on the array variables and your compiler should let you expand the array out so you can see all of the elements.



William
October 16, 2017 at 9:24 am · Reply

Hey.

I'm not sure If I understand what the difference between "run to cursor" and breakpoints exactly is. Both execute code until you hit that one specific line, no? With the "run to cursor" command, you can then continue to debug your code. However with a breakpoint, the program just stops executing and closes? Is this correctly understood?

Alex
October 21, 2017 at 9:40 am · Reply

A breakpoint is a pre-set spot where the debugger will always stop the code from executing. Run to cursor is essentially the same as setting a breakpoint, choosing run, and then

removing the breakpoint you just set when any breakpoint is hit. Breakpoint are typically placed in places where you always want to stop. Run to cursor is used when you want to go to a particular spot once.

Georges Theodosiou
September 27, 2017 at 3:50 am · Reply

My dear c++ Teacher,
Could you please add some instructions for code::blocks users?
With regards and friendship.

Mick
September 18, 2017 at 8:10 am · Reply

Hey Alex,

This could be me doing something incredible stupid but my program won't build. I get an error: 'x': redefinition, multiple initialization every time x is mentioned in the code.

Thanks in advance!

```
1  // deguggingexample.cpp : Defines the entry point for the console application.
2  //
3
4  #include "stdafx.h"
5  #include <iostream>
6
7  int main()
8  {
9          int x = 1;
10         std::cout << x << " ";
11
12         int x = x + 1;
13         std::cout << x << " ";
14
15         /*int x = x + 2;
16         std::cout << x << " ";
17
18         int x = x + 4;
19         std::cout << x << " ";
20
21
22     return 0;
23 }
```

Alex
September 18, 2017 at 10:50 pm · Reply

This is an easy one. You only need to tell the compiler that x is an int the first time you define x. After that, you just use x. e.g. "x = x + 1" instead of "int x = x + 1".

Nakul
July 14, 2017 at 10:13 pm · Reply

Hey Alex.
You said in this that VS takes you to the next line after the function call.What does that mean and

how to do it?
Plus when i press Step Over it jumps to some other format of code instead of my code. Pls help

Alex
July 15, 2017 at 10:03 am · Reply

So consider these two lines of code:

```
1 someFunction();
2 x = y;
```

If your cursor is on the someFunction line, and you step over the function, the someFunction() function will execute, and the program will stop just before x = y executes.

I'm not sure why Step Over is jumping you to another bit of code. What function and/or file is this code in?

« Older Comments 1 2