3.7 — Using an integrated debugger: Running and breakpoints

BY ALEX ON FEBRUARY 1ST, 2019 | LAST MODIFIED BY ALEX ON OCTOBER 7TH, 2019

While stepping (covered in lesson **3.6 -- Using an integrated debugger: Stepping**) is useful for examining each individual line of your code in isolation, in a large program, it can take a long time to step through your code to even get to the point where you want to examine in more detail.

Fortunately, modern debuggers provide more tools to help us efficiently debug our programs. In this lesson, we'll look at some of the debugger features that let us more quickly navigate our code.

Run to cursor

The first useful command is commonly called *Run to cursor*. This **Run to cursor** command executes the program until execution reaches the statement selected by your cursor. Then it returns control to you so you can debug starting at that point. This makes for an efficient way to start debugging at a particular point in your code, or if already debugging, to move straight to some place you want to examine further.

For Visual Studio users

In Visual Studio, the *run to cursor* command can be accessed by right clicking a statement in your code and choosing *Run to Cursor* from the context menu, or by pressing the ctrl-F10 keyboard combo.

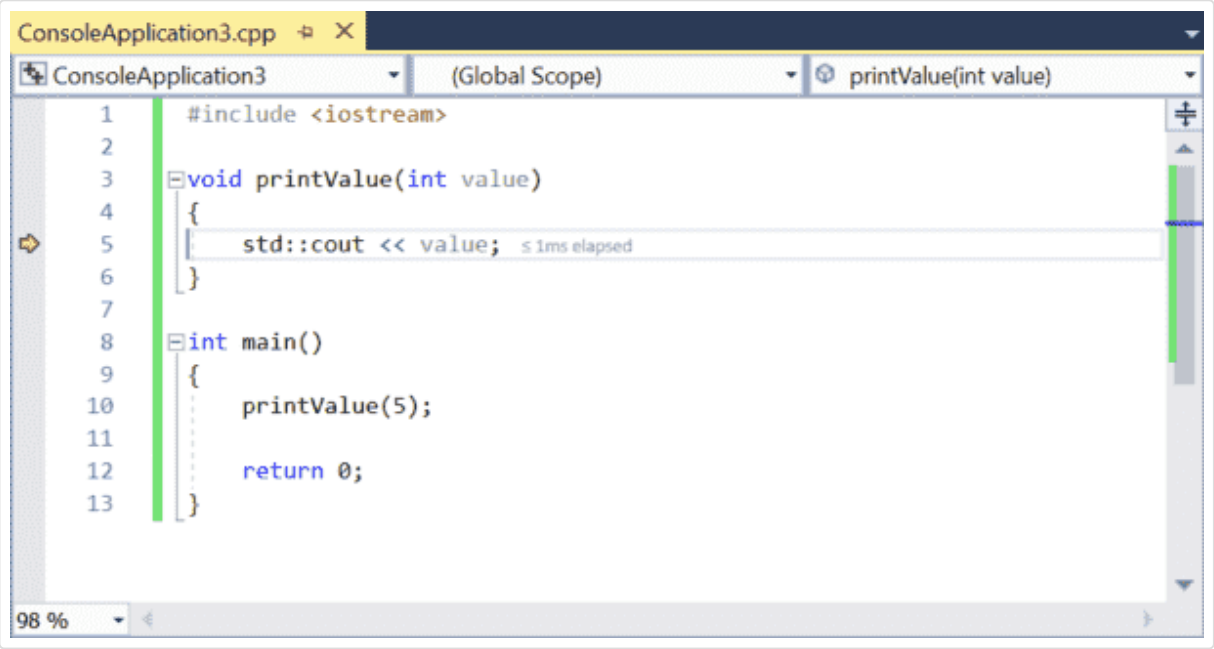
For Code::Blocks users

In Code::Blocks, the *run to cursor* command can be accessed by right clicking a statement in your code and choosing either *Run to cursor* from the context menu or *Debug menu* > *Run to cursor*, or by pressing the F4 shortcut key.

Let's try it using the same program we've been using:

```
1  #include <iostream>
2
3  void printValue(int value)
4  {
5      std::cout << value;
6  }
7
8  int main()
9  {
10     printValue(5);
11
12     return 0;
13 }
```

Simply right click anywhere on line 5, then choose "Run to cursor".



You will notice the program starts running, and the execution marker moves to the line you just selected. Your program has executed up to this point and is now waiting for your further debugging commands. From here, you can step through your program, *run to cursor* to a different location, etc…

If you *run to cursor* to a location that doesn't execute, *run to cursor* will simply run your program until termination.

Continue

Once you're in the middle of a debugging session, you may want to just run the program from that point forward. The easiest way to do this is to use the *continue* command. The **continue** debug command simply continues running the program as per normal, either until the program terminates, or until something triggers control to return back to you again (such as a breakpoint, which we'll cover later in this lesson).

For Visual Studio users

In Visual Studio, the *continue* command can be accessed while already debugging a program via *Debug menu* > *Continue*, or by pressing the F5 shortcut key.

For Code::Blocks users

In Code::Blocks, the *continue* command can be accessed while already debugging a program via *Debug menu* > *Start / Continue*, or by pressing the F8 shortcut key.

Let's test out the *continue* command. If your execution marker isn't already on line 5, *run to cursor* to line 5. Then choose *continue* from this point. Your program will finish executing and then terminate.

Start

The *continue* command has a twin brother named *start*. The *start* command performs the same action as *continue*, just starting from the beginning of the program. It can only be invoked when not already in a debug session.

For Visual Studio users

In Visual Studio, the *start* command can be accessed while not debugging a program via *Debug menu* > *Start Debugging*, or by pressing the F5 shortcut key.

For Code::Blocks users

In Code::Blocks, the *start* command can be accessed while not debugging a program via *Debug menu* > *Start / Continue*, or by pressing the F8 shortcut key.

If you use the *start* command on the above sample program, it will run all the way through without interruption. While this may seem unremarkable, that's only because we haven't told the debugger to interrupt the program. We'll put this command to better use in the next section.

Breakpoints

The last topic we are going to talk about in this section is breakpoints. A **breakpoint** is a special marker that tells the debugger to stop execution of the program at the breakpoint when running in debug mode.

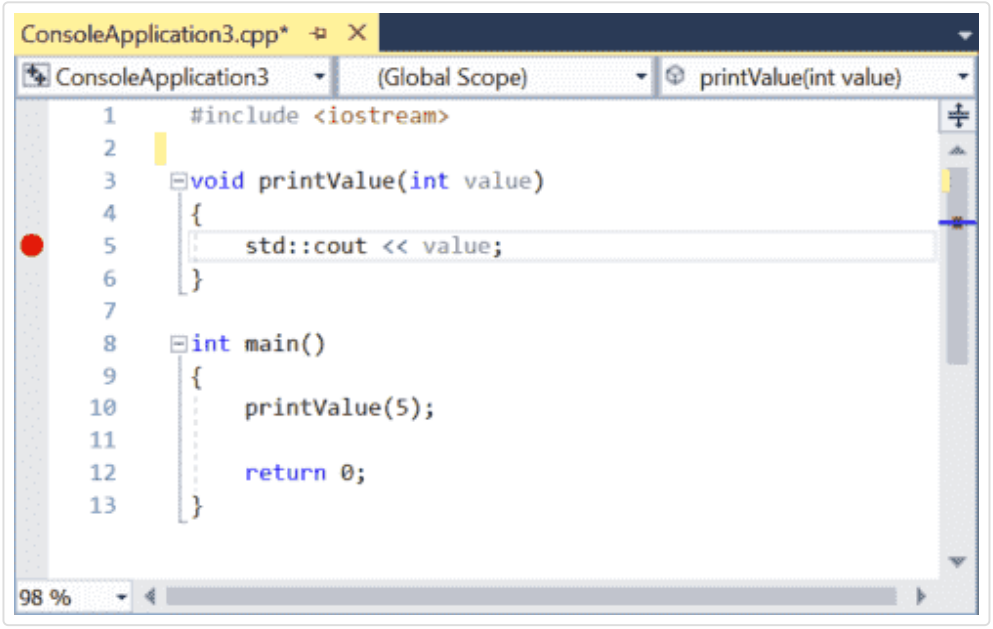
For Visual Studio users

In Visual Studio, you can set or remove a breakpoint via *Debug menu* > *Toggle Breakpoint*, or by right clicking on a statement and choosing *Toggle Breakpoint* from the context menu, or by pressing the F9 shortcut key, or by clicking to the left of the line number (in the light grey area).

For Code::Blocks users

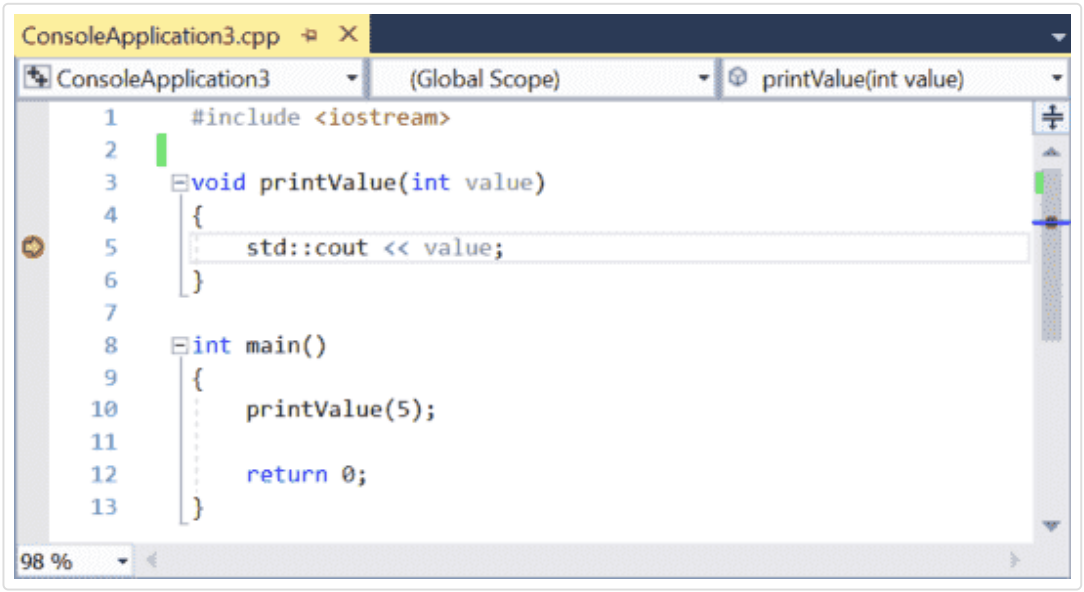
In Code::Blocks, you can set or remove a breakpoint via *Debug menu* > *Toggle breakpoint*, or by right clicking on a statement and choosing *Toggle breakpoint* from the context menu, or by pressing the F5 shortcut key, or by clicking to the right of the line number.

When you set a breakpoint, you will see a new type of icon appear. Visual Studio uses a red circle, Code::Blocks uses a red octagon (like a stop sign):



Go ahead and set a breakpoint on the line 5, as shown in the image above.

Now, choose the *Start* command to have the debugger run your code, and let's see the breakpoint in action. You will notice that instead of running all the way to the end of the program, the debugger stops at the breakpoint (with the execution marker sitting on top of the breakpoint icon):



It's just as if you'd *run to cursor* to this point.

Breakpoints have a couple of advantages over *run to cursor*. First, a breakpoint will cause the debugger to return control to you every time they are encountered (unlike *run to cursor*, which only runs to the cursor once each time it is invoked). Second, you can set a breakpoint and it will persist until you remove it, whereas with *run to cursor* you have to locate the spot you want to run to each time you invoke the command.

Note that breakpoints placed on lines that are not in the path of execution will not cause the debugger to halt execution of the code.

Let's take a look at a slightly modified program that better illustrates the difference between breakpoints and *run to cursor*:

```
1 #include <iostream>
2
```

```
3  void printValue(int value)
4  {
5      std::cout << value;
6  }
7
8  int main()
9  {
10     printValue(5);
11     printValue(6);
12     printValue(7);
13
14     return 0;
15 }
```

First, start a new debugging session and then do a *run to cursor* to line 5. Now choose *continue*. The program will continue to the end (it won't stop on line 5 again, even though line 5 is executed twice more).

Next, place a breakpoint on line 5, then choose *start*. The program will stop on line 5. Now choose *continue*. The program will stop on line 5 a second time. Choose *continue* again, and it will stop a third time. One more *continue*, and the program will terminate. You can see that the breakpoint caused the program to stop as many times as that line was executed.

Set next statement

There's one more debugging command that's used fairly uncommonly, but is still at least worth knowing about, even if you won't use it very often. The **set next statement** command allows us to change the point of execution to some other statement (sometimes informally called *jumping*). This can be used to jump the point of execution forwards and skip some code that would otherwise execute, or backwards and have something that already executed run again.

For Visual Studio users

In Visual Studio, you can jump the point of execution by right clicking on a statement and choosing *Set next statement* from the context menu, or by pressing the Ctrl-Shift-F10 shortcut combination. This option is contextual and only occurs while already debugging a program.

For Code::Blocks users

In Code::Blocks, you can jump the point of execution via *Debug menu* > *Set next statement*, or by right clicking on a statement and choosing *Set next statement* from the context menu. Code::Blocks doesn't have a keyboard shortcut for this command.

Let's see jumping forwards in action:

```
1  #include <iostream>
2
3  void printValue(int value)
4  {
5      std::cout << value;
6  }
7
8  int main()
9  {
10     printValue(5);
11     printValue(6);
```

```
12      printValue(7);
13
14      return 0;
15  }
```

First, *run to cursor* to line 11. At this point, you should see the value of *5* in the console output window.

Now, right click on line 12, and choose *set next statement*. This causes line 11 to be skipped and not execute. Then choose *continue* to finish executing your program.

The output of your program should look like this:

57

We can see that `printValue(6)` was skipped.

This functionality can be useful in several contexts.

In our exploration of basic debugging techniques, we discussed commenting out a function as a way to determine whether that function had a role in causing an issue. This requires modifying our code, and remembering to uncomment the function later. In the debugger, there's no direct way to skip a function, so if you decide you want to do this, using *set next statement* to jump over a function call is the easiest way to do so.

Jumping backwards can also be useful if we want to watch a function that just executed run again, so we can see what it is doing.

With the same code above, *run to cursor* to line 12. Then *set next statement* on line 11, and *continue*. The program's output should be:

5667

Warning

The *set next statement* command will change the point of execution, but will not otherwise change the program state. Your variables will retain whatever values they had before the jump. As a result, jumping may cause your program to produce different values, results, or behaviors than it would otherwise. Use this capability judiciously (especially jumping backwards).

Warning

You should not use *set next statement* to change the point of execution to a different function. This will result in undefined behavior, and likely a crash.

Conclusion

You now learned the major ways that you can use an integrated debugger to watch and control how your program executes. While these commands can be useful for diagnosing code flow issues (e.g. to determine if certain functions are or aren't being called), they are only a portion of the benefit that the integrated debugger brings to the table. In the next lesson, we'll start exploring additional ways to examine your program's state, for which you'll need these commands as a prerequisite. Let's go!


3.8 -- Using an integrated debugger: Watching variables


Index


3.6 -- Using an integrated debugger: Stepping

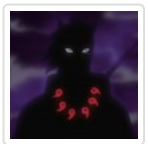
C++ TUTORIAL | PRINT THIS POST

21 comments to 3.7 — Using an integrated debugger: Running and breakpoints

OLGD
November 30, 2019 at 10:16 am · Reply

"Set Next Statement" skips lines that modify any variables already modified - is this intended? Code below. I'm using latest ver. Visual Studio Community 2019 on Win-10 OS, debugging in debug configuration

```
1 // SNS == Set Next Statement; SO = Step Over; RTC = Run To Cursor; cout == Console output
2 int i{5};
3 printValue(i); // [2] SNS + SO --> cout: "566"
4 i = i + 1;
5 printValue(i); // [1] RTC + SO --> cout: "56" // [3] SO --> cout: "5666"
```

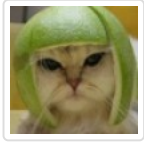


OLGD
December 1, 2019 at 7:41 am · Reply

Solved: problem stemmed from using Release configuration (auto-reverted from Debug for some reason).

HolzstockG
November 20, 2019 at 10:14 pm · Reply

So what's the difference between "step back" and "set next statement"? Isn't that quite the same when "set next statement" is used backwards?



Alex
November 21, 2019 at 11:12 am · Reply

"Step back" rewinds the state of everything, as if you'd never gone past that point in the first place. Any changes to variable values or other program state is undone.

"Set next statement" when used to jump backwards only changes the point of execution. Any changes to variable values or other program state is not changed.

HolzstockG
November 21, 2019 at 11:11 pm · Reply

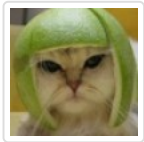
Thx :)

Bombi Barlsson
October 4, 2019 at 8:51 pm · Reply

I just wanted to add that you can click to the right of the line number in Code::Blocks to toggle a breakpoint

Paul
September 23, 2019 at 6:57 am · Reply

When the debugger finishes executing the .exe window closes and i cant see the output of the program. Is there any way to change this on code::blocks?

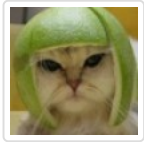


Alex
September 23, 2019 at 3:51 pm · Reply

One way: put a breakpoint on main's return statement and look at your output before continuing.

Logan
May 25, 2019 at 6:18 am · Reply

Sorry I have used the "Breakpoint", but I cannot find the option for, "Set next statement."
I am using the an updated version of Visual Studios. When I right click I get a window showing options for, quick actions and refactoring. There is no listing for the command under this, nor in the debugging window. Please help I pressed the keys and nothing either.



Alex
May 29, 2019 at 12:47 pm · Reply

"Set next statement" only occurs in the right click menu when you've already started debugging a program. Step into your program once and try again. I've updated the lesson text to note this.

Logan
May 30, 2019 at 6:05 am · Reply

Thanks, I missed the step to select Run to Cursor, when this is selected and you right click the code again it reveals the menu option now showing Set Next Statement. I noted it does not appear if you just select start debugging; far as I can tell this is the only way it appears.

Alex
June 1, 2019 at 9:26 am · Reply

Start with debugging will run the program to completion if you don't have any breakpoints, so by the time you have a chance to right click, your program will probably have already terminated and you will no longer be in debug mode.

Red Lightning
April 11, 2019 at 10:36 pm · Reply

"In Visual Studio, the continue command can be accessed while not debugging a program via Debug menu > Start Debugging, or by pressing the F5 shortcut key."

This line is under the title of START, I noticed continue and start are accessed in the same way, but people which don't notice it can get a bit confused because of that.
The same for the Code::Blocks line under the visual studio line.

blazk
March 10, 2019 at 4:09 pm · Reply

In the "set next statement" section

> Now, right click on line 13, and choose set next statement. This causes line 12 to be skipped and not execute. Then choose continue to finish executing your program.

> The output of your program should look like this:

> 57

> We can see that printValue(6) was skipped.

Isnt the output supposed to be 5?
if so, line 11 & 12 skipped and printValue(6) and printValue(7) skipped

> With the same code above, run to cursor to line 13. Then set next statement on line 12, and continue. The program’s output should be:

> 5667

Isnt it supposed to be 5677?

Alex
March 11, 2019 at 10:07 pm · Reply

I've fixed a couple of line numbering typos, so I think the answers should be correct now.

Lorenz
March 4, 2019 at 1:17 pm · Reply

Please note, Set next statement shortcut is Shift+Ctrl+F10 on Visual Studio 2017, not F9

Alex
March 4, 2019 at 9:21 pm · Reply

Yikes. Error fixed. Thanks for pointing that one out.

WSLaFleur
February 19, 2019 at 4:35 am · Reply

"Note that breakpoints placed on lines that do not execute will not cause the debugger to halt execution of the code."

I could be misunderstanding something, but when I tried putting a breakpoint on the empty line 11 of your example code, Microsoft Visual Studio's default (I haven't messed with any setting of which I'm aware) behavior was to shift the break point to the next following line of code and halt it there.

Alex
February 19, 2019 at 8:05 pm · Reply

Does this update clarify things? "Note that breakpoints placed on lines that are not in the path of execution will not cause the debugger to halt execution of the code.", or does it need further refinement?

WSLaFleur
February 20, 2019 at 5:13 am · Reply

Ah, okay. So, for instance, if I place a breakpoint on line 18 (after EOF for int main()) then it won't halt execution.

My understanding was that the compiler begins execution at the top of int main() and then proceeds through the code sequentially, but then why, if I put a breakpoint on an empty line above int main() and outside of a function, does it still halt execution at the start of int main()?

The clarification does help a bit! Sorry for niggling over such a minor detail. Thanks for all your help!

Alex
February 21, 2019 at 5:54 pm · Reply

> So, for instance, if I place a breakpoint on line 18 (after EOF for int main()) then it won't halt execution.

Correct.

> why, if I put a breakpoint on an empty line above int main() and outside of a function, does it still halt execution at the start of int main()?

It looks like Visual Studio temporarily relocates any breakpoints defined in the global scope to the start of the next function defined in the file.