1.11 — Debugging your program (stepping and breakpoints)

BY ALEX ON NOVEMBER 16TH, 2007 | LAST MODIFIED BY ALEX ON MARCH 29TH, 2017

Syntax and semantic errors

Programming can be difficult, and there are a lot of ways to make mistakes. Errors generally fall into one of two categories: syntax errors, and semantic errors (logic errors).

A **syntax error** occurs when you write a statement that is not valid according to the grammar of the C++ language. This includes errors such as missing semicolons, undeclared variables, mismatched parentheses or braces, and unterminated strings. For example, the following program contains quite a few syntax errors:

```
#include <iostream>; // preprocessor statements can't have a semicolon on the end

int main()
{
    std:cout < "Hi there; << x; // invalid operator (:), unterminated string (missing "), and undeclared
 variable
    return 0 // missing semicolon at end of statement
}
```

Fortunately, the compiler will generally catch syntax errors and generate warnings or errors, so you easily identify and fix the problem. Then it's just a matter of compiling again until you get rid of all the errors.

Once your program is compiling correctly, getting it to actually produce the result(s) you want can be tricky. A **semantic error** occurs when a statement is syntactically valid, but does not do what the programmer intended.

Sometimes these will cause your program to crash, such as in the case of a divide by zero:

```
#include <iostream>

int main()
{
    int a = 10;
    int b = 0;
    std::cout << a << " / " << b << " = " << a / b; // division by 0 is undefined
    return 0;
}
```

Sometimes these will just produce the wrong value:

```
#include <iostream>

int main()
{
    std::cout << "Hello, word!"; // spelling error
    return 0;
}
```

or

```
#include <iostream>

int add(int x, int y)
{
    return x - y; // function is supposed to add, but it doesn't
}

int main()
{
    std::cout << add(5, 3); // should produce 8, but produces 2
    return 0;
}
```

Unfortunately, the compiler will not be able to catch these types of problems, because the compiler only knows what you wrote, not what you intended.

In the above example, the errors are fairly easy to spot. But in most non-trivial programs, many semantic errors will not be easy to find by eyeballing the code.

Fortunately, that's where a debugger can really come in handy.

The debugger

A **debugger** is a computer program that allows the programmer to control how a program executes and watch what happens as it runs. For example, the programmer can use a debugger to execute a program line by line, examining the value of variables along the way. By comparing the actual value of variables to what is expected, or watching the path of execution through the code, the debugger can help immensely in tracking down semantic errors.

Early debuggers, such as **gdb**, had command-line interfaces, where the programmer had to type arcane commands to make them work. Later debuggers (such as Borland's **turbo debugger**) came with their own "graphical" front ends to make working with them easier. Almost all modern IDEs available these days have **integrated debuggers** -- that is, the debugger is built-in to the editor, so you can debug using the same environment that you use to write your code (rather than having to switch programs).

Nearly all modern debuggers contain the same standard set of basic features -- however, there is little consistency in terms of how the menus to access these features are arranged, and even less consistency in the keyboard shortcuts. Although our examples will be from Microsoft Visual Studio 2005 Express, you should have little trouble figuring out how to access each feature we discuss no matter which development environment you are using.

Before proceeding: Make sure your program is set to use the ***debug build configuration****.*

Stepping

Stepping is a debugger feature that lets you execute (step through) your code line by line. This allows you to examine each line of code in isolation to determine whether it is behaving as intended.

There are actually 3 different stepping commands: step into, step over, and step out. We will cover each one in turn.

Step into

The **step into** command executes the next line of code. If this line is a function call, "step into" enters the function and returns control at the top of the function.

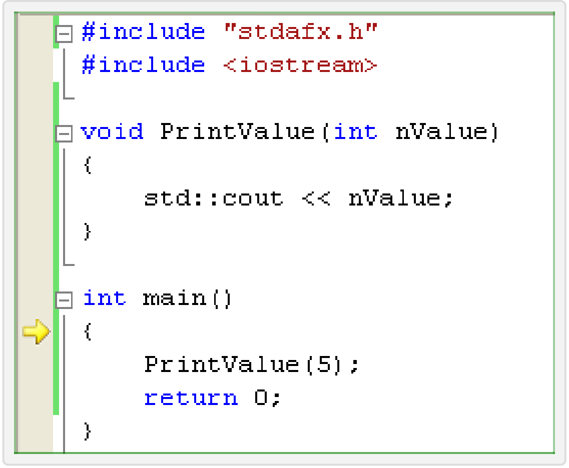
Let's take a look at a very simple program:

```
#include <iostream>

void printValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    printValue(5);
    return 0;
}
```

As you know, when running a program, execution begins with a call to main(). Because we want to debug inside of main(), let's begin by using the "Step into" command.

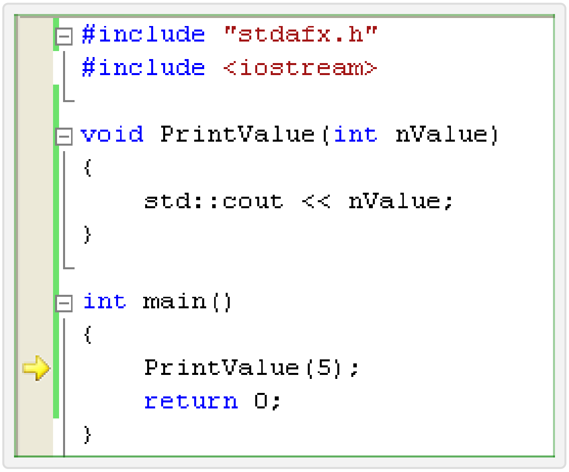
In Visual Studio 2005 Express, go to the debug menu and choose "Step Into", or press F11.
If you are using a different IDE, find the "Step Into" command in the menus and select it.

When you do this, two things should happen. First, because our application is a console program, a console output window should open. It will be empty because we haven't output anything yet. Second, you should see some kind of marker appear to the left of the opening brace of main. In Visual Studio 2005 Express, this marker is a yellow arrow. If you are using a different IDE, you should see something that serves the same purpose.



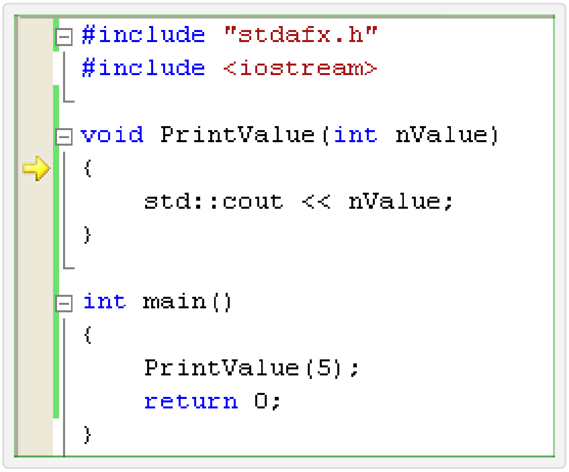
```
#include "stdafx.h"
#include <iostream>

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

This arrow marker indicates that the line being pointed to will be executed next. In this case, the debugger is telling us that the next line that will be executed is the opening brace of main(). Choose “Step into” again to execute the opening brace, and the arrow will move to the next line.



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

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

This means the next line that will be executed is the call to printValue(). Choose “Step into” again. Because printValue() is a function call, we “Step into” the function, and the arrow should be at the top of the printValue() code.



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

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

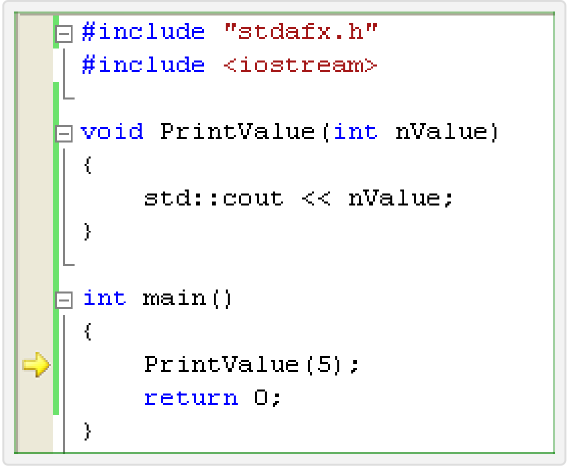
Choose “Step into” to execute the opening brace of printValue().

At this point, the arrow should be pointing to `std::cout << nValue;`.

Choose *“Step over”* this time (this will execute this statement without stepping into the code for operator<<). Because the cout statement has now been executed, you should see that the value 5 appears in the output window.

Choose “Step into” again to execute the closing brace of printValue(). At this point, printValue() has finished executing and control is returned to main().

You will note that the arrow is again pointing to printValue()!



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

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

While you might think that the debugger intends to call printValue() again, in actuality the debugger is just letting you know that it is returning from the function call.

Choose “Step into” twice more. At this point, we have executed all the lines in our program, so we are done. Some debuggers will terminate the debugging session automatically at this point. Visual Studio does not, so if you’re using Visual Studio, choose “Stop Debugging” from the debug menu. This will terminate your debugging session.

Note that “Stop Debugging” can be used at any point in the debugging process to end the debugging session.

Step over

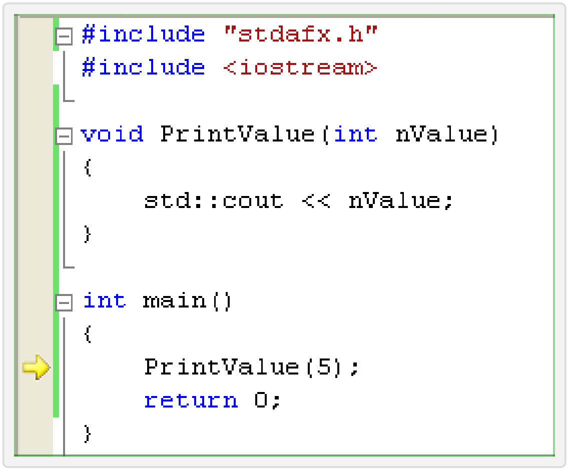
Like “Step into”, The **Step over** command executes the next line of code. If this line is a function call, “Step over” executes all the code in the function and returns control to you after the function has been executed.

Note for Code::Blocks users: In Code::Blocks, “Step over” is called “Next Line”.

Let’s take a look at an example of this using the same program as above:

```
#include <iostream>

void printValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    printValue(5);
    return 0;
}
```

“Step into” the program until the next statement to be executed is the call to printValue().



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

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

Instead of stepping into printValue(), choose “Step over” instead. The debugger will execute the function (which prints the value 5 in the output window) and then return control to you on the next line (`return 0;`).

Step over provides a convenient way to skip functions when you are sure they already work or do not need to be debugged.

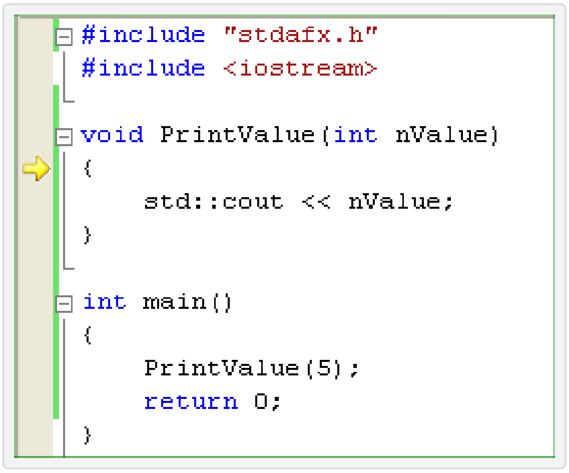
Step out

Unlike the other two stepping commands, “Step out” does not just execute the next line of code. Instead, it executes all remaining code in the function you are currently in, and returns control to you when the function has finished executing.

Let’s take a look at an example of this using the same program as above:

```
#include <iostream>

void printValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    printValue(5);
    return 0;
}
```

“Step into” the program until you are inside printValue().



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

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

Then choose “Step out”. You will notice the value 5 appears in the output window, and the debugger returns control to you after the function has terminated.

Run to cursor

While 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 just to get to the point where you want to examine in more detail.

Fortunately, modern debuggers provide a few more tools to help us efficiently debug our programs.

The first useful command is commonly called **Run to cursor**. This command executes the program like normal until it gets to the line of code selected by your cursor. Then it returns control to you so you can debug starting at that point. Let’s try it using the same program we’ve been using:

```
#include <iostream>

void printValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    printValue(5);
    return 0;
}
```

Simply put your cursor on the `std::cout << nValue;` line inside of printValue(), then right click and choose “Run to cursor”.

You will notice the arrow indicating the line that will be executed next moves to the line you just selected. Your program executed up to this point and is now waiting for your further debugging commands.

Run

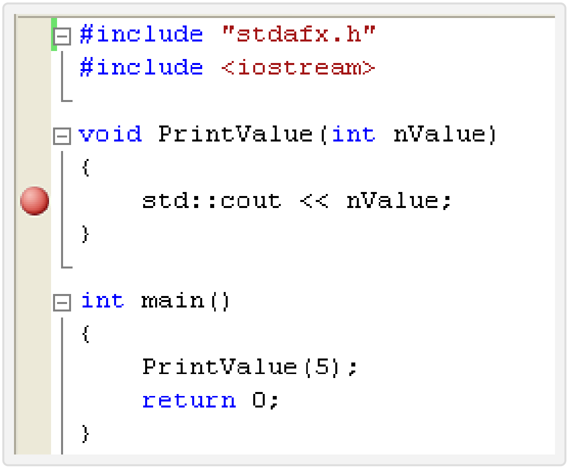
Once you’re in the middle of debugging a program, you can tell the debugger to run until it hits the end of the program (or the next breakpoint, which we’ll discuss in a second). In Visual Studio 2005 Express, this command is called “Continue”. In other debuggers, it may be called “Run” or “Go”.

If you have been following along with the examples, you should now be inside the printValue() function. Choose the run command, and your program will finish executing and then terminate.

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.

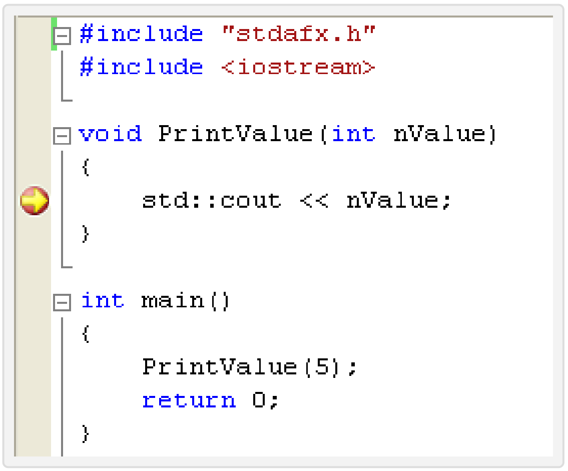
To set a breakpoint in Visual Studio 2005 Express, go to the Debug menu and choose “Toggle Breakpoint” (you can also right click, choose Breakpoint -> Insert Breakpoint). You will see a new type of icon appear:



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

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

Go ahead and set a breakpoint on the line `std::cout << nValue;`.

Now, choose “Step into” to start a debugging session, and then “Continue” 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 stopped at the breakpoint!



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

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5);
    return 0;
}
```

Breakpoints are extremely useful if you want to examine a particular section of code. Simply set a breakpoint at the top of that section of code, tell the debugger to run, and the debugger will automatically stop every time it hits that breakpoint. Then you can use the stepping commands from there to watch your program run line by line.

One last note: Up until now, we’ve been using “step into” to start debugging sessions. However, it is possible to tell the debugger to just start running to the end of the program immediately. In Visual Studio 2005 Express, this is done by picking “Start debugging” from the Debug menu. Other debuggers will have similar commands. This, when used in conjunction with breakpoints, can reduce the number of commands you need to use to get to the spot you want to debug in more detail using the stepping commands.

Conclusion

Congratulations, you now know all of the major ways to make the debugger move through your code. However, this is only half of what makes debuggers useful. The next lesson will talk about how to examine the value of variables while we are debugging, as well as a couple of additional windows of information we can make use of to help debug our code.

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

Index

1.10b -- How to design your first programs

Share this:

118 comments to 1.11 — Debugging your program (stepping and breakpoints)

« Older Comments 1 2

karna bc
May 14, 2018 at 3:52 am · Reply

#define max 10
int main()
{
char a[max];
int total=1;
char ch;
for(int i=0;i
{
cin>>a[i];
total=total*a[i];
cout<<"Do you want to add?\n";
cin>>ch;
if(ch=='n')
break;
}
for(int i=0;i
cout<
cout<
return 0;
}

solve error