

Debugging with Visual Studio

This handout has many authors including Eric Roberts, Julie Zelenski, Stacey Doerr, Justin Manis, Justin Santamaria, and Jason Auerbach.

Because debugging is a difficult but nonetheless critical task, it is important to learn the tricks of the trade. The most important of these tricks is to get the computer to show you what it's doing, which is the key to debugging. The computer, after all, is there in front of you. You can watch it work. You can't ask the computer why it isn't working, but you can have it show you its work as it goes. Modern programming environments usually come equipped with a *debugger*, which is a special facility for monitoring a program as it runs. By using the Visual Studio debugger, for example, you can step through the operation of your program and watch it work. Using the debugger helps you build up a good sense of what your program is doing, and often points the way to the mistake.

Using the Visual Studio debugger

The Visual Studio debugger is a complicated environment, but with a little patience, you should be able to learn it to the point where you code more efficiently and productively.





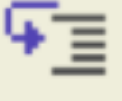

Under the Debug menu, there are two menu items for executing your code: the **Start (F5)** option and the **Start without Debugging (Ctl-F5)** option. You should always use the **Start** option so that you can set breakpoints and debug your code more easily. The debugger gives you the ability to stop your program midstream, poke around and examine the values of variables, and investigate the aftermath of a fatal error to understand what happened. If you set a breakpoint at the first statement of your `main` function, when you choose the **Start** command, the debugger sets up your program and brings up the windows without starting program execution. At this point, you control the execution of the program manually using the buttons on the debugger window. You can choose to step through the code line-by-line, run until you get to certain points, and so on.

The six most important toolbar icons stand for **Continue**, **Break**, **Stop Debugging**, **Step Over**, **Step Into**, and **Step Out**. These icons and their corresponding commands are detailed in Figure 1.

Whenever the program has stopped or has not yet been started, the **Continue** command will start it again from where it left off. The **Break** button is useful if the program is in an infinite loop or if you want to stop the program manually to use the debugger. The **Stop Debugging** command is useful if you want to quit the debugger and return to editing before the program has finished running.

When a program starts with debugging enabled, the debugger window opens and the program begins to execute. If you click on the **Break** button, you can stop the program through mid-execution and see what is happening right at that point. Setting a breakpoint (discussed below) will automatically stop the debugger at a specific line. Once stopped, there are three panes in the window filled with information about the program.

Figure 1. Debugger toolbar icons

Continue		Continues the program from the debugger's current position until you choose Stop or the program encounters a breakpoint, discussed below. The shortcut key for this is F5 .
Break		Stops the program wherever it happens to be and sets up the debugger in that spot.
Stop Debugging		Stops the program and debugger immediately. You can use Shift-F5 .
Step Over		Executes one step in the program, at the current level. If the program calls a function, that function is executed all at once, rather than by stepping through it. Use F10 to do this.
Step Into		Stops at the first line of the first function called on this line. Use F11 here.
Step Out		Runs the program until the current function returns. Use Shift-F11 .

Once your program is paused, the bottom pane of the debugger window will show the current function that is executing and a red arrow to the left of the code shows the next line to be executed. Choosing **Continue** causes your program to continue executing. The three **Step** buttons, by contrast, give you more fine-grained control over how the execution proceeds, which makes it possible to watch exactly what's happening as you search for bugs in your code.

To get a better sense as to how this entire process works, take a look at the example in Figure 2, which shows a program stopped before the call to the `distanceFromOrigin` function. Clicking **Step Over** executes the next line of code, automatically calling any functions invoked by that line. As a result, you can ignore the details of operations that are at lower levels of the code than the part you are debugging. Clicking **Step Over** at this point would execute the `distanceFromOrigin` call and assign it to `distance` without having to step through the details of `distanceFromOrigin`. The **Step Into** command makes it possible to drop down one level in the stack and trace through the execution of a function or a procedure. In Figure 2, the debugger would create the new stack frame for the `distanceFromOrigin` function and return control back to the debugger at the first statement of `distanceFromOrigin`. The **Step Out** command executes the remainder of the current function and returns control to the debugger once that function returns.

Figure 2. Using the Visual Studio debugger

[need to update this screen shot]

In the screenshot of the debugger, there are three panes in the window filled with information:

1. The bottom-right pane shows the call stack list. It lists the functions that have been called in reverse chronological order (last call on top). When you click a function in the stack frame list, the other panes update to show the code and variables for the selected function, allowing you to survey the variables in the entire execution path.
2. The bottom-left pane contains all variables and associated values contained in the stack frame highlighted in the `call stack` list. In Figure 2, `distance` has quite a strange value. Looking at the code (and using your debugging skills), you can see that `distance` has yet to be initialized at this point in execution and thus has a junk value. Also note that some variable types (such as structures, classes, and pointers) have a plus sign to their left. This means that there is more information for that data type that you can view. In Visual Studio, when a variable changes values it is highlighted in red. A highlighted value returns to black once the next line of code is executed.¹
3. The top pane contains the program's code with breakpoint information and current point of execution. The yellow arrow in the top frame indicates the next line of code to be executed. Selecting functions from the `context` list will adjust this window accordingly and change the yellow arrow to indicate the next statement to be executed for that particular function.

Another tab in the bottom-left window is labeled `watch 1`, which contains a useful tool for examining complex expressions and array elements. First, switch to that tab within the debugger by clicking on it. To create a watch, click once in the dashed box underneath the `name` label; it will expand and give you a cursor to type with. Type in any arbitrary expression that you want to get evaluated, anything from `x * 2` to `array[497]` and so on. The value of that expression will appear on the same line underneath the `value` label. You can create more labels by repeating the process in each dashed box. Be careful though! No syntax checking is provided, so your entries must be syntactically correct in order for them to evaluate correctly.

Using breakpoints

To set a breakpoint, place your cursor on a line of code and right click. Select `Breakpoint->Insert Breakpoint` from the menu, which will set a *breakpoint* on that particular line. You can also accomplish this by pressing `F9` or by clicking in the breakpoint margin. When the program is started by the `Continue` command, it will stop at every line that has a breakpoint, allowing you to debug easily. To remove a breakpoint, place your cursor on the line of code and right click, selecting `Breakpoint->Delete Breakpoint` from the pop-up menu (or you can press `F9` again).

¹ **Note:** In order to get all the variables for a given function's stack frame, you will need to click on the `Locals` tab in the bottom-left window. Otherwise, the debugger will only show you the variables that pertain to the current line of code.

Breakpoints are an essential component of debugging. A general strategy is to set a few breakpoints throughout your program; usually around key spots that you know may be bug-prone, such as computationally intensive or pointer-intensive areas. Then, run your program until you get to a breakpoint. Step over things for a few lines, and look at your variable values. Maybe step out to the outer context, and take a look to see that the values of your variables are still what they should be. If they're not, then you have just executed over code that contains one or more bugs. If things look well, continue running your program until you hit the next breakpoint. Lather, rinse, repeat.

Getting to the scene of the crime

While running the debugger, it is usually easy to see where exactly a program crashes. On a memory error where a dialog box pops up (such as an “Access fault exception”), the program immediately halts, and the debugger window presents the current state of the program right up to the illegal memory access. Even though the program has terminated, you can see exactly where it stopped, dig around and look at variable values, look at other stack frames and the variable values in those calls, and do some serious detective work to see what went wrong.

A call to `Error` has a similar behavior. If there are cases which shouldn't happen when the code is running correctly but might if there is a bug, you can add checks for them and call `Error` if the checks turn out true. This means that if one of those cases occurs, the debugger will stop on the `Error` line so you can look around and see what's going wrong.

Sometimes it is not obvious at all as to what is going on and you don't know where to start. Errors aren't being triggered, and there aren't memory exceptions being raised, but you know that something's not right. A great deal of time debugging will not be spent fixing crashes so much as trying to determine the source of incorrect behavior.

Imagine you have an array of scores. It is perfectly fine when you created and initialized it, but at some later point, its contents appear to have been modified or corrupted. There are 1000 lines executed between there and here—do you want to step through each line-by-line? Do you have all day to work on this? Probably not! Divide and conquer to the rescue! Set a breakpoint halfway through those 1000 lines in question. When the program breaks at that point, look at the state of your memory to see if everything's sane. If you see a corrupted or incorrect value, you know that there's a problem in code that led to this point. Just restart and set a breakpoint halfway between the beginning of the code path and the first breakpoint. If everything looks okay to this point, repeat the process for the second half of the code path. Continue until you've narrowed the bug down to a few lines of code. If you don't see it right away, take a deep breath, take a break, and come back and see if it pops out at you.

Building test cases

Once your program appears to be working fine, it's time to really turn up the heat and become vicious with your testing, so you can smoke out any remaining bugs. You should be hostile to your program, trying to find ways to break it. This means doing such things as entering values a user might not normally enter. Build tests to check the edge-cases of your program.

For example, assume you've written a function

```
generateHistogram(Vector<int> & buckets, Vector<int> & scores)
```

where the **buckets** vector represents uniformly sized ranges that together cover the range from the minimum score to the maximum score, which are given by constants. When a score falls in the range of that specific bucket, the bucket is incremented by one.

Seeing the process through

One of the most common failures in the debugging process is inadequate testing. Even after a lot of careful debugging cycles, you could run your program for some time before you discovered anything amiss.

There is no strategy that can guarantee that your program is ever bug free. Testing helps, but it is important to keep in mind the caution from Edsger Dijkstra that “testing can reveal the presence of errors, but never their absence.” By being as careful as you can when you design, write, test, and debug your programs, you will reduce the number of bugs, but you will be unlikely to eliminate them entirely.