Getting Started with the Visual C++ Debugger

This tutorial discusses how Visual Studio 6.0’s debugging tools can be used to find and fix problems in C++ programs. It doesn’t assume any experience with either debuggers or Visual Studio’s debugging tools. The debugger is a tool to help correct runtime and semantic errors — note that no debugging tools are useful in solving compiler errors.

Compiler errors are those that show at the bottom of the screen when compiling. Things like "V:\debugger\temp.cpp(11) : error C2146: syntax error: missing ';' before identifier 'cin'" Since these are always associated with a particular line of the program, they’re generally fairly easy to solve.

The difficult problems are the ones that happen while the program is running. The syntax is correct (or it wouldn't have compiled), so the error is that the code doesn't do what you think it does.

Basic Debugging Technique
If the program isn't working correctly, one of two things could be going wrong:
1) Data is corrupt somewhere. Was something added that should be subtracted? Is some user input not being stored properly?
2) The code isn’t correct. Are all the loop conditions set properly? Are the functions called in the right order? Is there an if where there shouldn’t be one?

The idea of debugging is to determine what line is causing the problem so the error can be more easily identified. It's important to note that the line on which a problem evidences itself is not necessarily where it originated.

For example:
```
a = 0;
b = 1;
cout << (b / a) << endl;
```

The error actually happens on the 'cout' line, but the problem is caused two lines up where a is set to 0. In a complex program, the difference between where the error occurs and where it is manifested may be hundreds of lines.

The most basic way to hunt down errors (and the only way in some languages) is to echo information out to the screen:
```
cout << "I'm in getRatio totVal = " << totVal << endl;
```

That works but could take a long time to sift through a thousand lines of code to find the one line causing the problem. That's where the debugger comes in.

A Buggy Program
Trying to debug a program that's working perfectly is rather pointless, so we'll begin with a program that has some obvious problems:
This program compiled cleanly (0 warnings, 0 errors), but obviously some things aren't working right. One and three AREN'T equal, and 0.85 is NOT 0%. So we'll examine the code and determine where the problems are.

The Buggy Code
Copy the following code into Visual Studio to follow the tutorial:

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int toPercent (float decimal);

void main() {
    int a, b;
    float c;
    int cAsPercent;

    cout << "Enter two Integers >";
    cin >> a >> b;

    if (a == b) cout << "They are Equal!\n";
    else if (a > b) cout << "The first one is bigger!\n";
    else cout << "The second one is bigger!\n";

    cout << "Enter a Decimal to be Converted to Percent >";
    cin >> c;
    cAsPercent = toPercent(c);
    cout << "That's " << cAsPercent << "%\n";
    cout << endl << endl;
    system("pause");
}

/* ToPercent():
Converts a given float (eg 0.9) to a percentage (90).
*/
int toPercent (float decimal) {
    int result;
    result = int(decimal) * 100;
    return result;
}
Debug Mode or Not?
None of the debugging tools available are useful if the program isn't being run in debug mode. Rather than just clicking the Exclamation Point icon or pressing Ctrl+F5 to run your program, the debugger must be run from the "Build" menu by clicking "Start Debug" then "Go". The F5 key alone will also run in debug mode.

Visual Studio also makes a distinction between "Build for Debug" and "Build for Release." The default is "Debug," so unless you've changed it you don't need to look for the option. The debugging functions can be accessed from Visual Studio's menus but the Debug Toolbar is usually easier to use. There are also keyboard shortcuts for each function; hover your mouse over one of the toolbar's buttons to see its description and keyboard shortcut.

If the toolbar isn't showing, right-click near an existing toolbar (at the top of the screen) and select "Debug" from the list. The toolbar can be "Docked" and "Undocked" — drag it wherever is most convenient for you.

Breakpoints
Breakpoints are the lifeblood of debugging. Regardless of whether your error is data-related or sequence-related, you'll still need to pause the program at some point and examine what's happening in detail; that's what a breakpoint does (it breaks at a certain point).

In the Buggy Program example, let's set a breakpoint just before the user enters the two numbers that the program thinks are equal. Move to that line (cin >> a >> b;) and either right-click and select "Insert/Remove Breakpoint" or press the F9 key. You can also just click in the gray bar at the left to set a breakpoint. Right click on an existing breakpoint glyph to set properties, disable, or delete the breakpoint.

A red glyph will appear next to the line, indicating a breakpoint is set:

```
cout << "Enter two Integers >":
  cin >> a >> b;
  if (a == b) cout << "They are Equal!\n";
  else if (a > b) cout << "The first one is bigger!\n";
  else cout << "The second one is bigger!\n";
```

Run the program in Debug mode, and the output window will appear, but the compiler window will 'jump' to the foreground and a yellow arrow will be pointing to the line with your breakpoint.

The yellow arrow means that statement will execute next! That is, it has not executed yet.

```
cout << "Enter two Integers >":
  cin >> a >> b;
  if (a == b) cout << "They are Equal!\n";
  else if (a > b) cout << "The first one is bigger!\n";
  else cout << "The second one is bigger!\n";
```

Watches
Once you've identified the general part of the program where the trouble lies ("the part that asks
for two numbers and compares them”) and you’ve setup a starting breakpoint, you can start
cHECKING FOR DATA PROBLEMS (ONE OF THE TWO TYPES OF PROBLEMS YOU MIGHT HAVE).

The "Watch" window lets you watch the contents of any variables you select as your program
executes. Thus, if the value changes in an unexpected way you’ll see it as it happens. If the
Watch window isn’t already open, open it from the View menu (Debug Windows >
Watch), or by clicking the "Watch" icon in the toolbar, or by pressing Alt+3.

If there are already some variables entered, just click each one and press the Delete key to
clear it. The two variables involved in this particular problem are \(a\) and \(b\). Click the Watch window
and press "a" (a text box will appear when you start typing), then Enter to add \(a\) to your Watch
list:

Add \(b\) the same way and you can examine their values:
These two numbers (while coincidentally equal) are completely useless — they mean only that a and b haven't been initialized (remember, the 'cin' statement hasn't executed yet)! Extremely negative numbers like this are almost sure signs of uninitialized variables.

To determine what's really going wrong, it's necessary to execute some more code.

### Stepping

Once you pause execution you have the ability to step through code, in other words, execute code one line at a time. From the debug menu there is a Step Into command (F11). If you are currently in break mode on a line of code that contains a method call, the Step Into command will enter the method and break again on the first line of code inside the method. In contrast, the Step Over command will execute the entire method call and break on the next line of code in the current method. Use Step Into if you want to see what happens inside a method call; use Step Over if you only want to execute the entire method and continue in the current code block.

If the instruction pointer is currently on a line of code that does not contain a method call, Step Over and Step Into will both move the instruction pointer to the next line of code.

The debug menu also contains a Step Out command, which you can use when you want to execute the rest of the current method and return to the calling method. Step Out will break execution at the return point in the calling function.

Two more tips: right-clicking on a line of code and selecting the 'Run To Cursor' will put the application into run mode until execution reaches the specified line of code. Also, you can click and drag the yellow instruction point with the mouse to skip code, or to re-execute code.

**So for our code:** To execute a line of code, click the "Step Over" icon in the toolbar (also found in the Debug menu) or press F10. Doing this causes the 'cin' to execute.

At first it looks like nothing happens. Rather, the change is just subtle! Switch to your program's window and you'll see it's waiting for your two numbers.

As soon as you enter both numbers, Visual Studio will 'jump' back to the foreground, and the yellow arrow will have moved.

You told it to execute only that one statement, and as soon as it finished executing it paused again.

```cpp
    cout << "Enter two Integers >";
    cin >> a >> b;
    if (a == b) cout << "They are Equal!\n";
    else if (a > b) cout << "The first one is bigger!\n";
    else cout << "The second one is bigger!\n";
```
Examine the Watch window. The values for both $a$ and $b$ are in red, which means they changed during the most recent execution. Since you're only watching two variables this seems obvious, but if you had many variables and you were watching them over many lines of code, such an "obvious" indicator can be very valuable.

Clearly the values are as expected — the same two values "the user" entered. So that particular line isn't the problem. "Step Over" the next line (which won't require any user input) and again examine the Watch window.

The value of $a$ is still in red, which means it changed during the most recent execution. The fact that it changed before doesn't matter anymore; it's red because it changed during the execution of the "if ..." line. (Notice that $b$ is black again because it did not change).

Not only did the value of $a$ change, but clearly it changed to be equal to $b$ which explains the problem. Examine the line you just executed, and the error should be fairly obvious — the condition of the 'if' is $(a = b)$ rather than $(a == b)$. Rather than having to stare at the entire program, the debugger showed you exactly what line was "broken."

When you are tired of stepping through the code, F5 resumes execution.

**Stopping the Debugger**

When you've found a problem to correct, it may be tempting to press Ctrl+C in your program window to end the program. *DON'T DO THAT!* If you end your program, and then try to end the debugger, Visual Studio fails and closes.
Instead, select "Stop Debugging" from the Debug menu or on the toolbar or press Shift+F5. That will both end your program and end the debugger, which makes Visual Studio happy.

### Conditions and Hit Counts

Most of the time you'll add a plain breakpoint to an application and not set any special properties. If you open up the breakpoint window (from the Debug menu, select Windows -> Breakpoints), you'll notice a breakpoint can use conditions and hit counts. Conditions and hit counts are useful if you don't want the debugger to halt execution every time the program reaches the breakpoint, but only when a condition is true, or a condition has changed, or execution has reached the breakpoint a specified number of times.

Conditions and hit counts are useful when setting breakpoints inside of a loop. For example, if your code iterates through a collection of Customer objects ith a for each loop, and you want to break on the 10th iteration of the loop, you can specify a hit count of 10. If something bad only happens when the Customer object's Name property is equal to "Scott", you can right click the breakpoint red glyph, select Condition from the context menu, and enter the expression customer.Name == "Scott" into the breakpoint condition textbox. Intellisense is available in this textbox to ensure you are using the correct syntax.

### Viewing State

#### DataTips

One of the common techniques to view the data inside a variable is to place the mouse cursor over the variable in code and allow Visual Studio to display a DataTip. DataTips are only available when the program is in break mode. The data tip will display information about the variable.

If the object you are inspecting is a complex object, structure, or array, there will be a plus sign (+) to the left of the tip. If you hover over the + you can expand the DataTip to view additional fields and properties of the object in a tree like view. If the object you are inspecting has a property that itself represents another complex object, you can continue to expand the nodes of the tree and drill further and further into the object.

When viewing a DataTip you can edit writeable values of the object by right clicking the tip and selecting "Edit Value", or by left clicking the value itself. Press the Ctrl key to temporarily hide the DataTip.
So, for example, using the code below, I can see the contents of the all array.

```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

class Pair
{
public:
    int x, y;
    Pair(int x1=0, int y1=0){x = x1; y = y1;}
};

void main() {
    int const SIZE=10;
    int a, b;
    Pair all[SIZE];
    int tot=0;
    for (int i=0; i < SIZE; i++)
    {
        cout << "Enter two Integers >";
        cin >> a >> b;
        all[i] = Pair(a, b);
        tot += all[i].x;
    }
}
```

**Variable Windows**

There are other options besides the watch window. You can also open a locals window from the Debug menu (Windows -> Locals). The locals window will automatically display all local variables in the current block of code. If you are inside of a method, the locals window will display the method parameters and locally defined variables. The Autos window (Debug -> Windows -> Autos) will display variables and expressions from the current line of code, and the preceding line of code.

**Step Into**

To examine the second bug in the "Buggy Program" example, some code inside a function may or may not need to be examined. Sometimes you can assume a certain function is working
correctly, so it would be a waste of time to step through it. (Ideally you'll have tested certain functions before you try writing others; a lot of the time you'll know the older functions are still working). (For example, you should never step into `cin.get()` because even if it wasn't working, there'd be nothing you could do.

When you don't want to step through an entire function, use the "Step Over" command. When you do want to step into a function, use the "Step Into" command.

In the "Buggy Program" example, let's assume again that the value for 'c' is stored correctly and set a breakpoint on the next line — the one that calls the `ToPercent()` function. Then run the program until that point is reached.

In the Watch window, you can delete the two Watches that are set (a and b) by clicking on each line and pressing "Delete" (on the keyboard), or just type over the existing names. Enter 'c' and 'cAsPercent'.

'cAsPercent' clearly has garbage data, but it hasn't been assigned yet, so that doesn't matter. If you step over the function call, the value of 'cAsPercent' changes:

Whatever is happening in `ToPercent()` isn't working right, since it's returning a 0! Unfortunately, the entire function has already executed. It's far more hassle than it's worth to "Undo" the execution of a line; instead either click the "Restart" toolbar icon, or "Stop Debugging" and then run it again.

This time, step into `ToPercent()` and add 'decimal' and 'result' to the Watch window (the two variables local to `ToPercent()`). Notice also that 'c' and 'cAsPercent' have error messages now — they belong to `main()` so you can't inspect their values if you're in
You can step over or into the next line, since no functions are called; when you do the value of 'result' becomes 0! Looking closely at the line, 'decimal' is being truncated to an int before it's multiplied.

**Stepping Into Assembly**

Some statements that don't look like function calls really are -- the most common being `cin >> variable;` and `cout << "Text";` Be careful when you "Step Into" lines involving `cout`, `cin`, or other system functions!

If you do, you'll get a prompt to find the source code for those functions:
If you click Cancel (which is what you should do), the Disassembly window will appear:

Not only is this not your source code, it's not even C++! Obviously this isn't somewhere fun to be. Performing a "Run to Cursor" is usually the easiest way to get out of this situation.

**Run to Cursor [not in 2005?]**

A "Run to Cursor" is like a temporary breakpoint. Right-click on the line just after the one you accidentally stepped into and select "Run to Cursor" (or use the toolbar button). This will tell the program to run until it gets to that point unless another breakpoint is encountered. If one is encountered, the program will break like normal and will not remember where the cursor was.

The "Run to Cursor" command is particularly good at starting the debugger (perform a "Run to Cursor" before you've started the debugger to get to the point you want to start debugging), and getting out of the Disassembly window. It's not very good if you already have a lot of breakpoints already set.

**Visual Studio Debugger - Watch Window**

There are many useful commands available in the watch window, which can alter the way variables and error values are displayed by the debugger.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>Displays the return value of a function after it returns.</td>
</tr>
<tr>
<td>@err</td>
<td>Display the current Win32 GetLastError value</td>
</tr>
<tr>
<td>string, su</td>
<td>Display the specified string as Unicode.</td>
</tr>
<tr>
<td>value, hr</td>
<td>Treat the specified value as a HRESULT and display it's string representation</td>
</tr>
<tr>
<td>value, wm</td>
<td>Decode the specified value as a Windows Message.</td>
</tr>
<tr>
<td>value, x</td>
<td>Display value in Hexadecimal</td>
</tr>
<tr>
<td>array, 23</td>
<td>Display exactly 23 elements of the specified array</td>
</tr>
<tr>
<td>address, mb</td>
<td>Display the specified address/variable as a memory-block hex-dump.</td>
</tr>
</tbody>
</table>

**Tip: Call Stack Window Debugging:** You can create a call stack window by selecting Debug|Window|Call Stack at a break point. For those unfamiliar with the Call Stack Window, this is how you find out "how did I get here" - it enables you to go back up the call stack to find out "from where" the method that you are in was called. Not only can you see where it was called from, you can actually go "Back there" and trace the exact execution path in the debugger. If you haven't discovered it yet, please do so now. Many developers are not aware that you can set breakpoints in the Call Stack Window. Just highlight the call you want to stop on and either press F9 or right-click on the line and select Insert Breakpoint from the shortcut menu. You can also right-click on any breakpoint in the Call Stack Window to enable, disable or set the properties of the breakpoint.

**Tip: Use Run to Cursor for one-shot breakpoints:** All you need to do is right-click on the line you want and choose "Run to Cursor" from the menu, and program execution will immediately continue to the point where you are at. Many developers are not aware that this is available from both debugging and editing. In edit mode, this will actually start the debugger and run your program to the cursor location. As with breakpoints, right-clicking in the Call Stack window pops up the shortcut menu that also offers this feature.

**Tip: Use the BreakPoints Window:** If you know the name of the class and method you want to break on, you can choose New from the Breakpoints window and type it directly into the Function area of the window. Developers can spend 15 minutes wandering through their projects opening files just so they can position the cursor over a line of code and hit the F9 key to set a breakpoint. If you know the name of the class and method, doing it this way will find it and set the breakpoint. In addition, you can click it in the Breakpoints list of the window and go to the location automatically.

You can also set a breakpoint by just entering the method name, if it is unique in your application. And, if it isn't you will actually get a list of ALL the locations where the method can be found,
which is extremely useful for setting multiple breakpoints on a commonly used function throughout your project. It also displays for overloaded methods in the same class.

**Tip: Use Assertion Debugging:** The idea behind asserts is very simple: When you are writing code, you understand that some condition is always expected to be true. If that condition is ever not true, then there is a bug. You need to include `<assert.h>` for it to work.

```
assert (myValue >=0);
```

A very powerful, yet highly under-utilized feature.