Debugging Applications 101

by Dale LaForce

Table of Contents

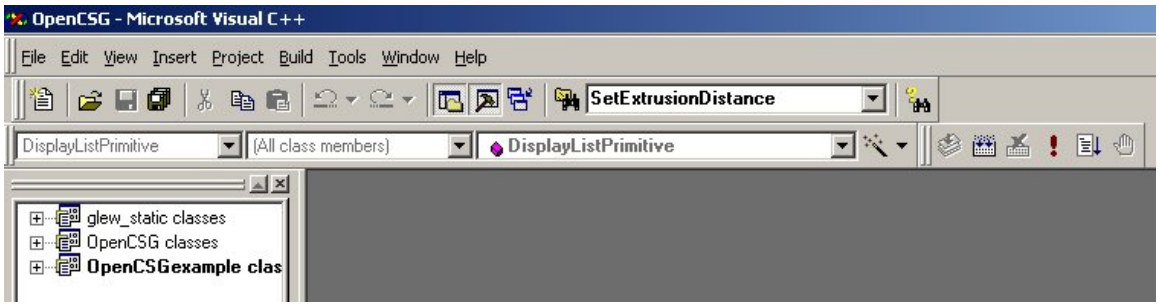
Introduction

When writing new code or modifying existing code, debugging typically consumes a large portion of a programmer's time. In this short article, we are going to try to provide a rudimentary foundation in the use of the "debugger" tool shipped with most development suites. We will be using Visual Studio 6.0 for our examples and explanations but you should find that the concepts translate fairly well to other development environments.

Debugging is the process of correcting coding and logic errors so that your project will build and execute properly, working according to the design goals you had in mind. The debugger is a tool that provides an interface to facilitate this process. It provides special menus, windows, dialog boxes, and spreadsheet fields for this purpose; drag-and-drop functionality is often available as well for moving debug information between components. The debugger essentially allows you to observe and even alter the execution of your program code in a line-by-line fashion. In doing so, you will be able to isolate the source of errors, determine whether or not proper commands are executed, view the current contents of variables, and so on. The application can even be paused during execution, where it will wait for user input after completing a debugging command (like break at breakpoint, step into/over/out/to cursor, break at exception, break after Break command or Restart) which we will see shortly.

An old adage says, "I will not be able to teach you how to remember everything, but I will teach you how to find the information you cannot remember." This simple statement is the key to the pursuit of success in most research fields. No one can know everything, but if you know how to find information, then it is just as good as total knowledge. In this article, we will try to provide a good amount of background material to get you underway, and we will also point you in the direction of additional resources which you should find helpful.

In the next section, we are going to start to look at the toolbars, windows, and other fine trappings associated with the debugger tool.



The Visual Studio 6.0 IDE

Apart from buttons, toolbars, and some other slight variations, most of the information presented here should apply to your debugger. If you are not using Visual Studio 6.0 then you will need to make the necessary adjustments to correlate this information to the

proper button, window or technique. However, the basics generally remain the same regardless of the visual appearance of the environment. Note as well that a number of items have short cut keys assigned so that you can more easily access their functionality, so you will want to look those up as well. If you are new to code development or unfamiliar with an integrated development environment like Visual Studio, then you can be sure that it will take a little time to learn and become familiar with everything. But do not despair. These tools are designed to help make your development time more efficient and once you are familiar with the IDE, you will find your projects coming together a lot quicker.

First, let us take a look at some of the important components that comprise the debugger.

The Debugger Windows

There are several specialized windows that display debugging information for your program. We will review them quickly now and then come back a bit later to see them all in action.

The Watch Window

The Watch window displays names and values of variables and expressions.

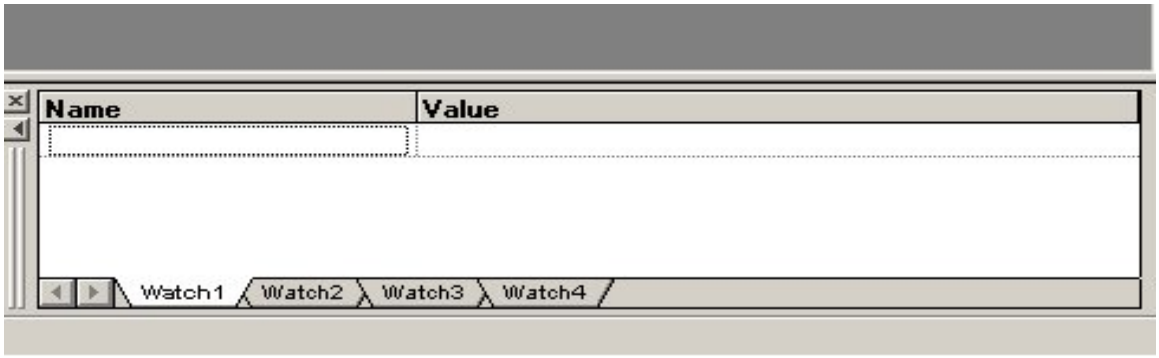


Fig 1.1 Watch window

The Variables Window

The Variables window displays information about variables used in the current and previous statements, function return values (in the **Auto** tab), variables local to the current function (in the **Locals** tab), and the object pointed to by **this** (in the **This** tab).

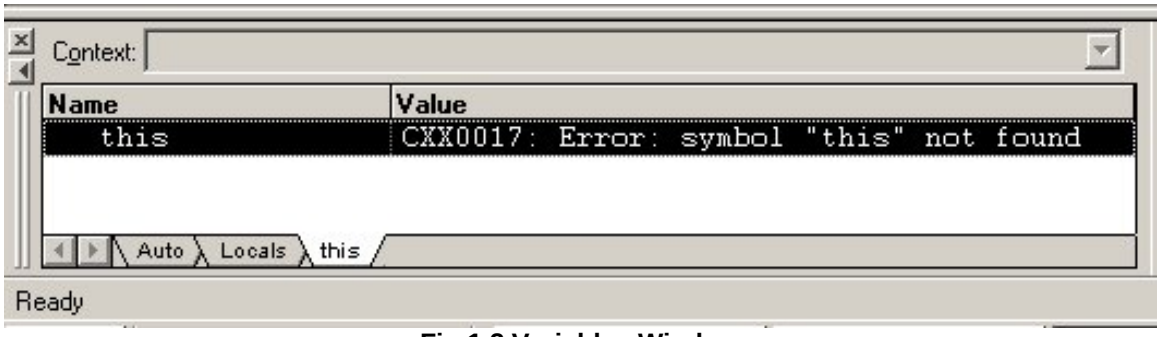


Fig 1.2 Variables Window

The Call Stack Window

The Call Stack window displays the stack of all function calls that have not returned.

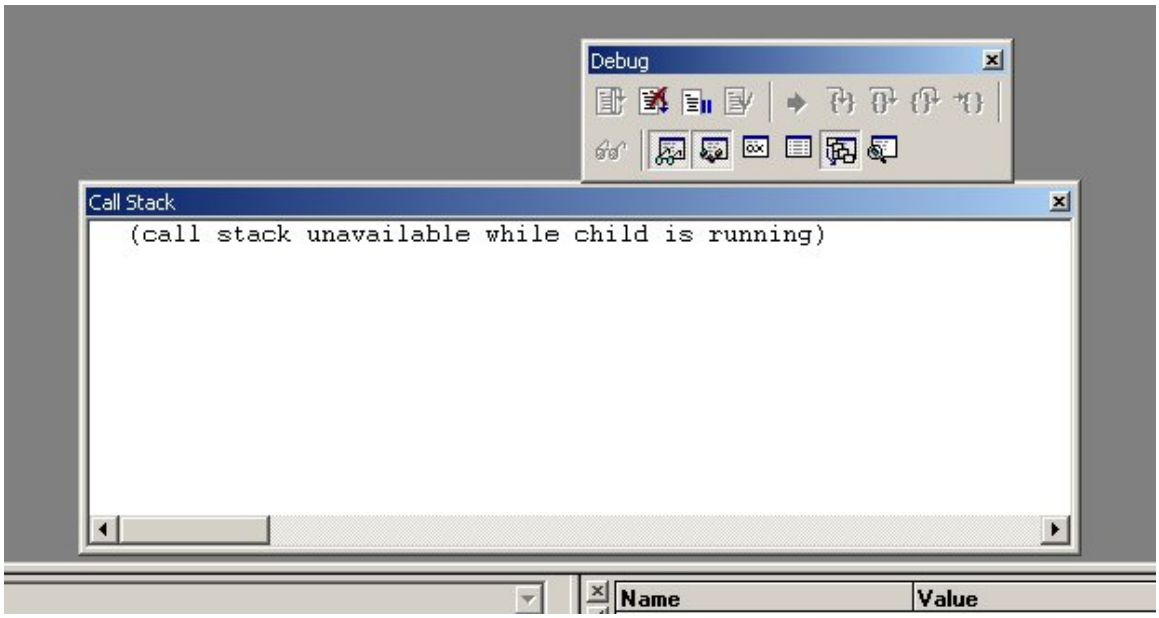


Fig 1.3 Call Stack Window

The Registers Window

The Registers window displays the contents of the general purpose and CPU status registers.

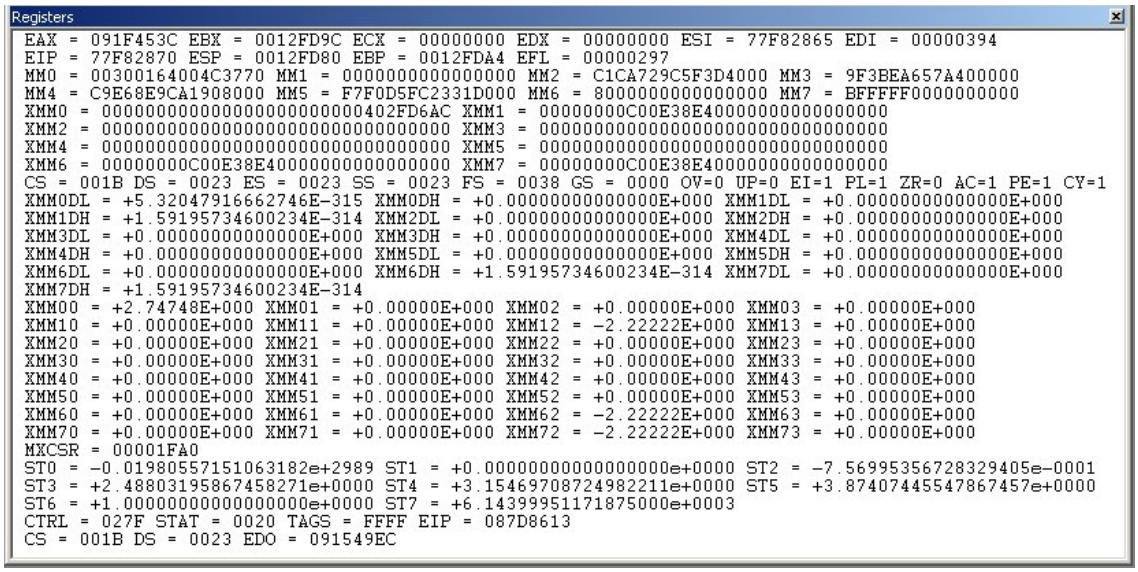


Fig 1.4 Registers Window

The Disassembly Window

The Disassembly window displays assembly-language code derived from disassembly of the compiled program.

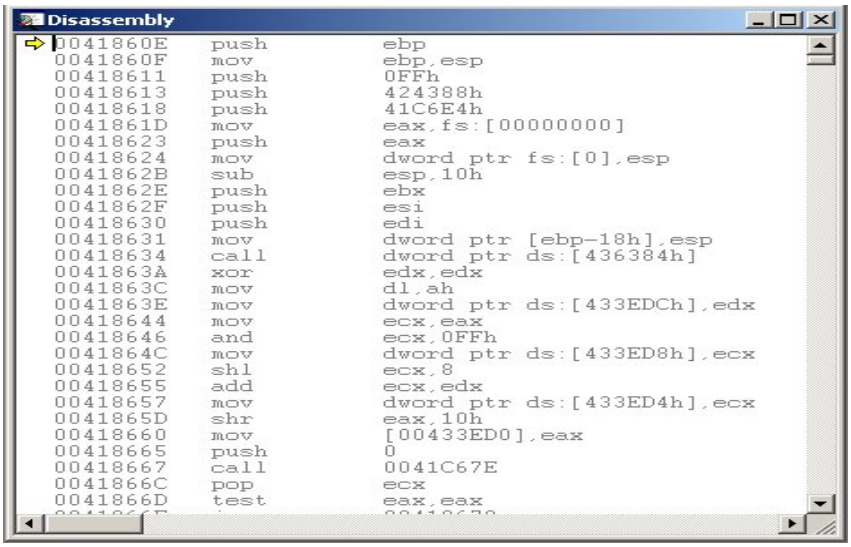


Fig 1.5 Disassembly Window

The Memory Window

The Memory window displays the current memory contents.

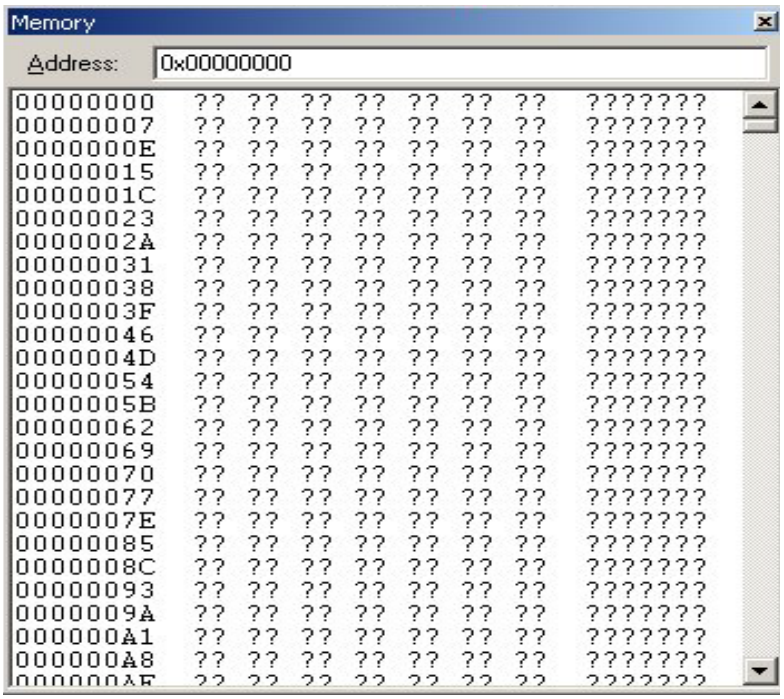


Fig 1.6 Memory Window

All of the Debugger windows can be docked or floating. When a window is in floating mode, you can resize or minimize the window to increase/decrease the visibility of other windows. You can also copy information from any debugger window, but you can only print information from the Output window.

Debugger Window Formatting

To set formatting and other options for the Debugger windows, use the **Debug** tab in the **Options** dialog box. Figure 1.7 shows the Debug sheet in the Options dialog box which can be accessed by choosing Tools from the Main menu and then selecting Options, which will display the dialog box. Then, just select the tab at top labeled Debug as seen in Figure 1.7.

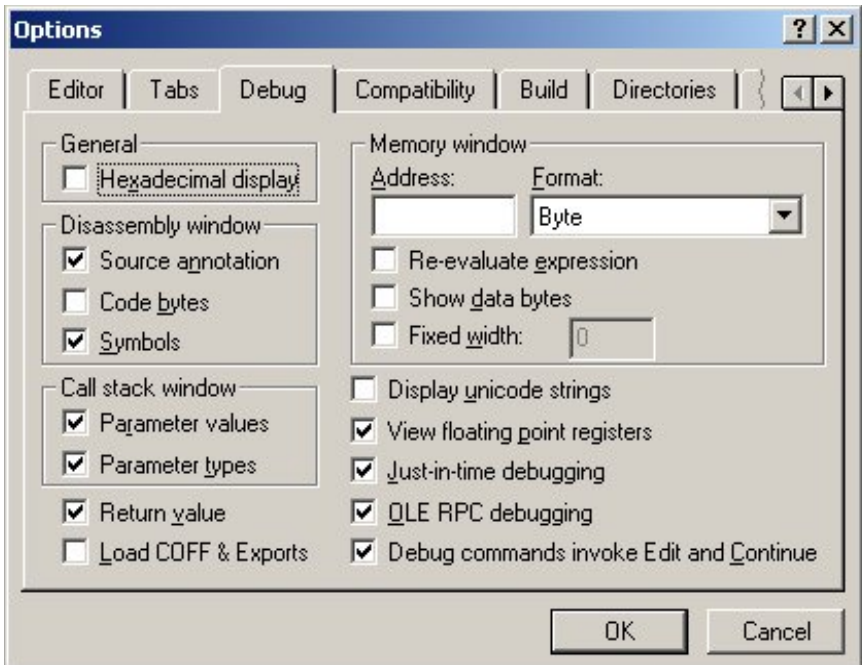


Fig 1.7 Debug Sheet of Options Dialog

If we check “Hexadecimal display” under the General heading, it displays values in hexadecimal format and parses user input in hexadecimal format in all debugger windows and dialogs. When using the debugger in hexadecimal mode, it is possible to enter decimal numbers using the prefix 0n (read : zero “n”), for example: 0n1000.

Dropping down to the Disassembly window label, we first see the check box labeled Source annotation. When checked, this displays source code within the listing of assembly-language code. Next we have a check box labeled Code bytes, which displays bytes corresponding to each assembly-language instruction (if checked). Finally, under the Disassembly window we have the Symbols check box. This displays symbolic names (such as CGIApp::OnGIEvent) for addresses in the Disassembly window.

Just below the Disassembly window settings box, we find the Call Stack window settings box. The first check box we encounter is labeled Parameter values. If checked, the debugger will display the values passed to each parameter for each function call shown in the Call Stack window. The remaining check box labeled Parameter types, displays type information for each parameter for each call shown in the Call Stack window.

Below the Call Stack window settings box, we see two check boxes: one labeled Return value and one labeled Load COFF & Exports. The check box labeled Return value, if checked, will display function return values in the Variables window. The check box labeled Load COFF & Exports, if checked, will enable the debugger to load COFF-format debugging information, or DLL Exports when debugging information is not available. Selecting this option may affect debugger performance when loading the application to be debugged.

NOTE: In 32-bit programming, COFF is a format for executable and object files that is portable across platforms. The Microsoft implementation of COFF is derived from the UNIX specification for COFF, but includes additional headers for compatibility with MS-DOS and 16-bit Windows. This Microsoft version is sometimes called the "portable executable (PE) file format."

Moving up and to the right, we come to the Memory window settings box and the First item we encounter is the text box labeled Address. This box will display the beginning address for the block of memory to be displayed. Just to the right of the Address box, we see the Format box. This is a drop down box, with a large selection to choose from. Basically, it determines the display format for memory contents. It is a good idea to click this box and scroll up and down to see the various formats you may choose from (Fig 1.8).

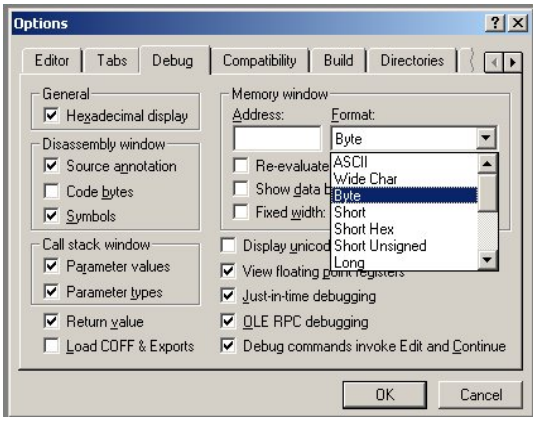


Fig 1.8 Format Drop Down Box

Just below the Address text box, we find the Re-evaluate expression check box. When checked, the debugger dynamically evaluates an expression entered in the Memory window. Select this option if you want to enter a pointer, for example, and have the memory window display the address pointed to, even when the pointer changes. Do not select this option if you want the pointer to be evaluated once and the Memory window to continue to point to the evaluated address even when the pointer changes.

Next we have Show data bytes check box. If enabled, the debugger displays data as raw bytes as well as in the selected format (as given by the drop down box mentioned above).

The final item in our Memory window settings box is the Fixed width check box. This displays memory contents in a fixed-width format when the checkbox is selected. We cab use the textbox to the right to specify the width. Width units are determined by the format selected in the Format box. For example, if you choose short and set the width to 4, each row in the Memory window will display four short values.
There are five more check boxes on this dialog to review.

Display unicode strings should be mostly self-explanatory. It displays Unicode-format strings for debugging international programs. Only check it if you need it.

Next we see the View floating point registers check box. When checked, the debugger displays contents of floating-point registers in the Registers window.

The Just-in-time debugging check box enables an application launched from the desktop to call the debugger when an error occurs. Thus, if you want the application you are creating to be able to call up the debugger when (however slim the chance) it errors out, then checking this box is the way to do that.

The OLE RPC debugging check box enables debugging of remote procedure calls. (That is what RPC stands for -- Remote Procedure Call).

Finally, we have the Debug commands invoke Edit and Continue check box. When this option is selected, Edit and Continue applies code changes automatically when you choose a Step, Go, or Run command. Otherwise, code changes are applied only when you choose Apply Code Changes.

“Edit and Continue” is a debugging feature introduced in the Microsoft Visual C++® version 6.0 development system. It allows you to make changes to source code during a debugging session, and to apply the code changes to the application being debugged, without having to stop debugging, rebuild, restart the debugger, and return that application to the state it was in when the bug occurred. For typical debugging sessions, Edit and Continue saves time by shortening the code, compile, and debug cycle and by allowing the programmer to maintain his or her concentration.

The Edit and Continue feature is enabled by default in all newly generated Visual C++ 6.0 debug configuration projects (through the /ZI compiler switch). Additionally, when using Visual C++ 6.0 to open a project generated with a previous version of Visual C++, users are prompted to convert the project to the 6.0 format, in which case the /ZI compiler switch replaces the use of /Zi for each project. Note that the /ZI switch is quite different from the /Zi compiler switch. Both of these switches configure the compiler to build a program database (.pdb) file with debug information, but when the /ZI switch is used, the .pdb file contain information necessary for performing Edit and Continue operations, *in addition to* the debug information generated with /Zi. To enable the /ZI compiler switch for a project within the Visual C++ IDE, select **Settings** from the Project menu after activating the project for which Edit and Continue will be enabled. In the Project Settings dialog box that appears, click the **C/C++** tab. Select **General** from the Category drop down box. In the Debug info drop down box, select **"Program Database for Edit and Continue."**

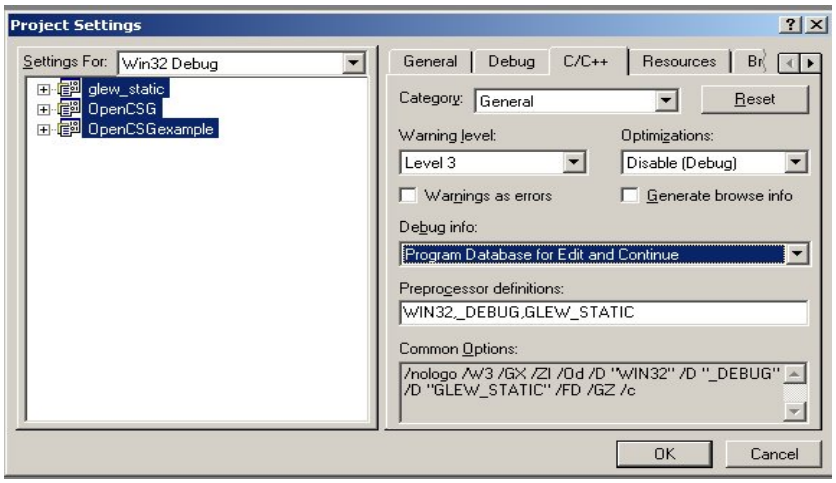


Fig 1.9 /ZI Switch and Debug Info Hi-lighted

The /ZI compiler switch will appear in the Project Options box at the bottom of the dialog box. Any other selection in the Debug info group box will disable the /ZI compiler switch. Click **OK** to accept the settings.

The Debugger Menus

Now that we have had a brief overview of the debugger windows we will encounter, let us move on to the debugger menu items. Commands for debugging can be found on the **Build** menu, the **Debug** menu, the **View** menu, and the **Edit** menu located on the Main menu of the IDE. The **Build** menu contains a command called **Start Debug**, which contains a subset of the commands on the full **Debug** menu (Fig 1.10). These commands start the debugging process (**Go**, **Step Into**, **Run To Cursor** and **Attach to Process**). The **Debug** menu appears in the menu bar while the debugger is running (even if it is stopped at a breakpoint). From the **Debug** menu, you can control program execution and access the QuickWatch window. When the debugger is not running, the **Debug** menu is replaced by the **Build** menu.

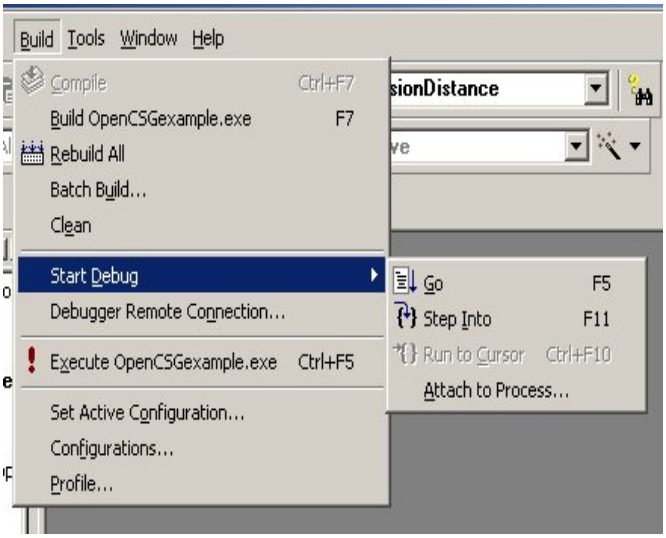


Fig 1.10 Subset of Start Debug

The **View** menu contains commands that display the various debugger windows, such as the Variables window and the Call Stack window discussed previously.

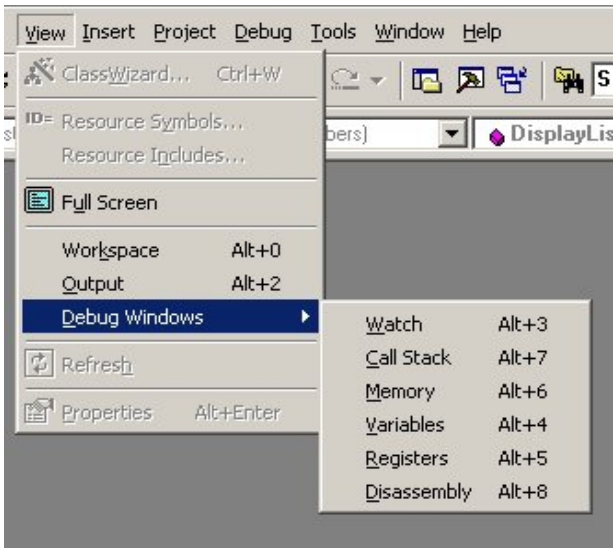


Fig 1.11 Debug Windows from View menu

From the **Edit** menu, you can access the **Breakpoints** dialog box, from which you can insert, remove, enable, or disable breakpoints. We will talk about breakpoints in more detail in the next section.

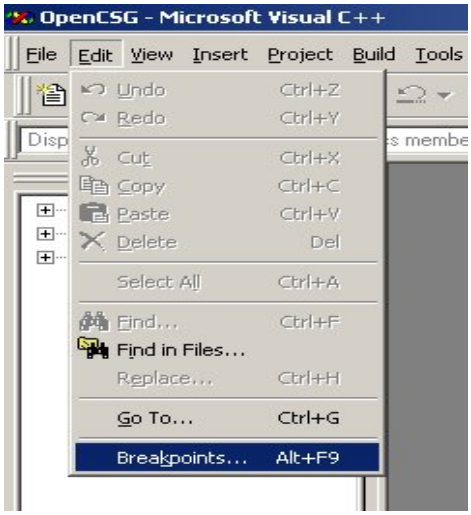


Fig 1.12 Breakpoints from the Edit Menu

In addition to windows, the debugger uses a number of dialog boxes to manipulate breakpoints, variables, threads, and exceptions. You can access the **Breakpoints** dialog box using the **Breakpoints** command on the **Edit** menu. You can access the other dialog boxes using commands from the **Debug** menu.

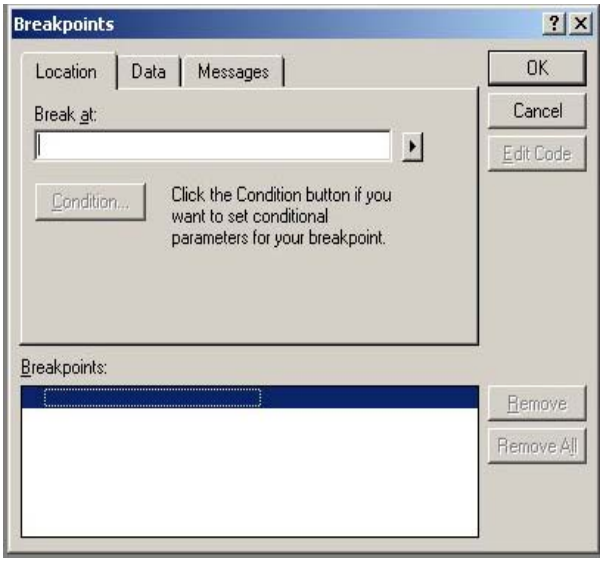


Fig 1.13 Breakpoints Dialog Box

Breakpoints

Before concluding our discussion of the debugger interface, let us discuss the concept of breakpoints, since they are quite important and still fresh in our minds.

Breakpoints are locations in the source code that tell the debugger where to halt (break) program execution. For obvious reasons, breakpoints represent one of the most important elements of the debugging process. They allow us to say to the debugger, "run the program up to this point and then stop here, so that I can look at the current state of the application". In this way we can check the values of our variables, return values from functions, and so on, to try to find out where things might be going wrong. Additionally, we can find out whether or not a particular line of code is ever even reached (for example, the code inside an if/else statement), as perhaps our logic is faulty somewhere or our error happens much sooner than the expected line we want to break on.

We can set breakpoints (one or more) in either the source window or the Call Stack window. A red dot next to the line of source identifies breakpoints in our code.

> **NOTE:** You must have a project open before you can set a breakpoint. With no project open, the Breakpoints command does not appear on the Edit menu.

Breakpoints can be set in various places:

- on a source-code line
- at the beginning of a function
- at the return of a function
- at a label

> **NOTE:** If you want to set a breakpoint on a source statement extending across two or more lines, you must set the breakpoint on the last line of the statement.

You can also set data breakpoints that halt execution when an expression changes value or evaluates to true. The following list describes instances that are common:

- when a variable changes value
- when an expression changes value
- when an expression is true
- break on a variable outside the current scope
- when the initial element of an array changes value
- when the initial element of an array has a specific value
- when a particular element of an array changes value
- when any element of an array changes value
- when any of the first *n* elements of an array change value
- when the location value of a pointer changes
- when the value at a location pointed to changes

- when an array pointed to by a pointer changes
- when the value at a specified memory address changes
- when a register changes
- when a register expression is true

The Start Debug Menu

To conclude our discussion of the debugger interface, let us quickly review the options that we can select through the Start Debug menu.

Attach to Process allows us to attach to a running process either locally or across the world on another machine.

Go executes code from the current statement until a breakpoint or the end of the program is reached, or until the application pauses for user input. (Equivalent to the **Go** button on the toolbar.)

Step Into, single-steps through instructions in the program, and enters each function call that is encountered along the way.

Run To Cursor executes the program as far as the line that contains the insertion point. This is equivalent to setting a temporary breakpoint at the insertion point location. In other words, where ever you place the cursor in the code, the execution of the application will run until it reaches the line that has the cursor on it, giving us a fast non-committal breakpoint that can be modified during execution just by clicking the line we wish to stop at.

QuickWatch window is supported only in Visual C++ Enterprise Edition. You can use **QuickWatch** to quickly examine the value of SQL variables and parameters. You can also use **QuickWatch** to modify the value of a local variable or to add a variable to the Watch window. You cannot modify the value of a global from the QuickWatch window.

That wraps up our overview of the debugger environment and some common terms and phrases. The next thing we shall do is look at the debugger in action and talk about some simple steps and routines to get us started in the debugging arena. While we will be brief, you can be sure that throughout your coding life, you will find ample time and instances to practice debugging techniques.

Debugging Examples

Debugging, while having a snappy name, is really our old friend “troubleshooting” with a new set of clothes. Troubleshooting is an art form in itself, and requires a combination of natural ability/curiosity, a basic skill set in the area of concern, and a relentless desire to find out why something will not function correctly, how it should function in an ideal setting, and what elements might contribute to problems that are encountered.

I have spent most of my adult life as a “troubleshooter” in the fields of advanced avionics, computer systems, and networks. It turns out that the basic skills I learned in one area were always assets in the others. One helpful procedure for troubleshooting, that once learned may be applied to any system you can think of, is called the Divide and Conquer or Half-Step Method. Fundamentally, the idea is to treat a system as a “black box”. Starting at either the input end or output end, divide the box in half, test, then repeat the division until the area of concern is narrowed down.

For example, consider an ordinary lamp where the input is electrical energy coming from the wall outlet and the output is light energy emitted into the environment. Let us try going from output to input first. We can divide the lamp in half (not physically of course) and check to see if electrical energy is present at that point. If we find that electrical energy is present at that point, we have eliminated the first half of our system in one step. Now from this point to the output we will once again divide it in half. Since the only object in the system that is left in this example is the light bulb itself, we can be fairly certain that it is at fault. Common sense would probably have led us to the bulb initially of course, and the same can be said in other systems as well. There are usually common “things” that go wrong or break in all systems and we can capitalize on this knowledge when we have it handy. Often this knowledge is gained through experience and trying to remember the mistakes you made on past projects (you should not be too surprised to find that the same bugs keep cropping up over and over again across most projects). Keeping up with the latest literature in our field of concern will also keep us apprised of common problems so that we know what to look for when things do not behave as expected.

The point here is that if you run into a problem and have no idea where to start, remember the half-step method. It can be applied to many other areas as well, not just debugging. Virtually all systems have an input and an output, so you can divide the whole in half, working towards the input or output end, repeating the process until you narrow down the culprit.

Let us put some of these ideas into practice by looking at a few typical debugging examples. Of course, as with everything we do in life, with practice and repetition (and a willingness to accept that we will make mistakes) we will become more comfortable with solving problems using the debugger. And to be sure, if you are hoping to land a job as a software programmer, keep in mind that the debugger tool is the most important tool you have in your toolset. Familiarity with the tool will be assumed at any job you get.

Debugging Example 1

Our first example is very simple and will get us started with using the debugger and get us acquainted with where everything is and what it looks like. This will save some frustration and time.

Start a new empty console project in Visual Studio and give it a snazzy title. Then create a new source file in your project and cut and paste the following code into the source file you just created.

> **NOTE:** When building an application, always create a Debug build first. Then when you have all the issues in order (no errors, no warnings), build the Release version.

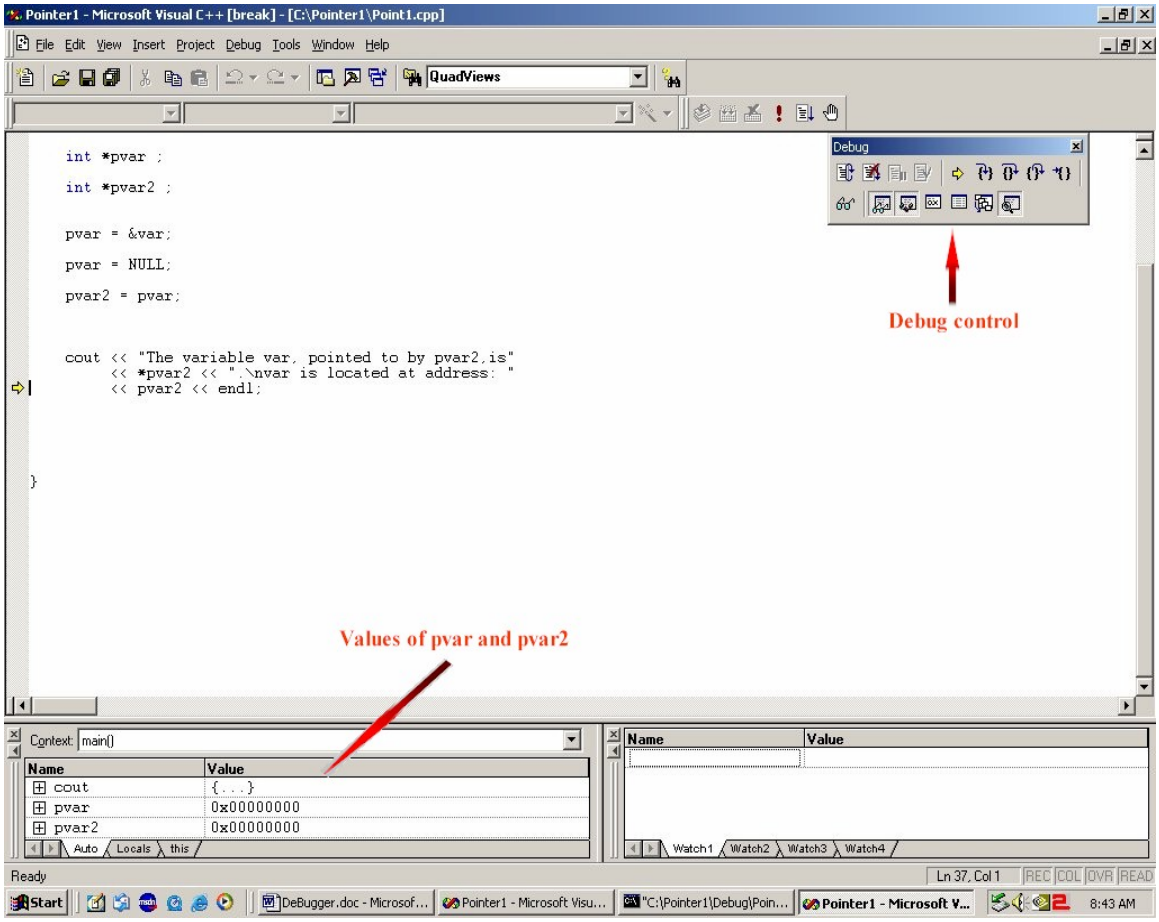
```
////////////////////////////////////////////////
//
//  Module: Point1.cpp
//
//  Purpose: To demonstrate the NULL pointer
//           effects in a program and provide
//           a simple demonstration of debugging
//           the app.
//
//
////////////////////////////////////////////////

#include <iostream.h>

void main ()
{
      int var = 50;
      int *pvar;
      int *pvar2;
      pvar  = &var;
      pvar  = NULL;
      pvar2 = pvar;

      cout << "The variable var, pointed to by pvar2, is "
       << *pvar2 << ".\nvar is located at address: "
       << pvar2 << endl;

}
```

Save your workspace at this point and check the build options to make sure that you are in the debug mode.

Next, under the Tools menu at the top, choose Options and then the Debug tab. If it is checked, uncheck the box next to just in time debugging. This will prevent our app from opening another instance of Visual Studio. On final build, if we want to give the user the ability to bring up the debugger in the event of an error, we can go back and recheck this box. This is useful if we are on final build and want to do extensive testing of our app -- it is a quick way to access the debugger from our app, without the need to start Visual Studio and load the app by hand.

Now build the application.

Finally, hit Ctrl+F5 to run the application. When prompted, hit cancel to start the debugger.



Notice the debug window in the upper right (if it is not present, then right click on an empty space on the main menu bar and select it from the drop down that appears). The Output window at the bottom shows the value of our two pointer variables. We can see immediately that they are NULL (or zero). Just above the Output window, we notice a yellow arrow pointing to the pvar2. This tells us the point that the program "choked". But be very careful here in making assumptions, as this is *not* the location of the error; it is simply the point at which we *noticed* the error because the application failed on this line. In fact, the error could have occurred 10,000 lines of code before the yellow arrow if we were working in a more complex program. This is something that is very important to keep in mind. We now know the location in the program where things broke down, but we do not yet know why the application broke down.

Of course, you have probably already spotted the error by now -- the line that sets pvar to NULL. If you comment out or delete this line and then rebuild the app and run it again, the application should run properly and our output to the DOS window is correct. It is

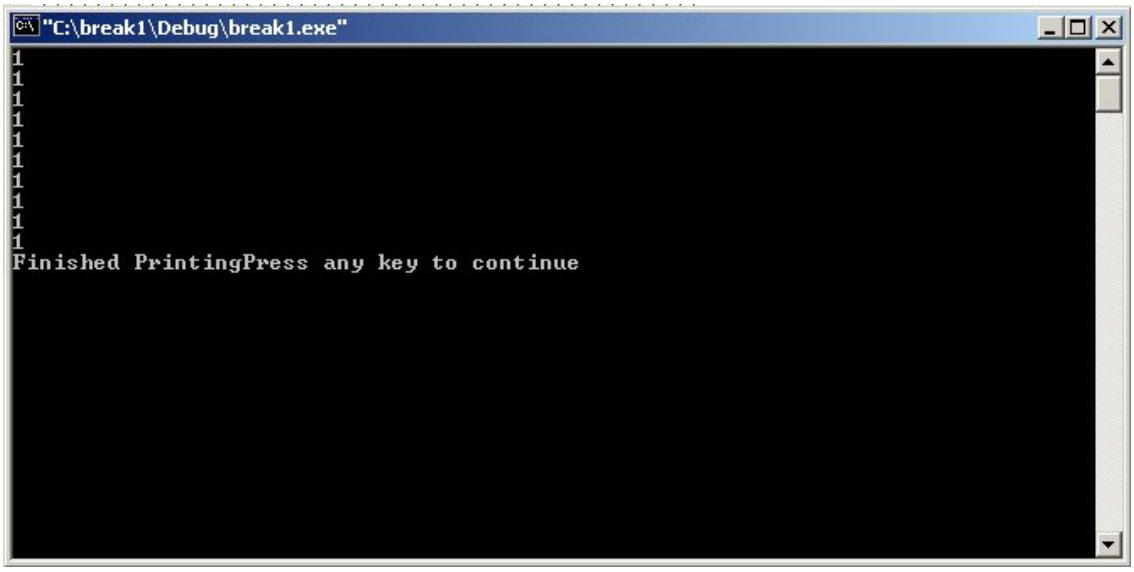
worth saying that NULL pointer errors (which can cause very nasty General Protection Faults) are one of the most common types of bugs and will likely come across your radar many times in the future. Fortunately, they are generally very easy to spot and fix.

Debugging Example 2

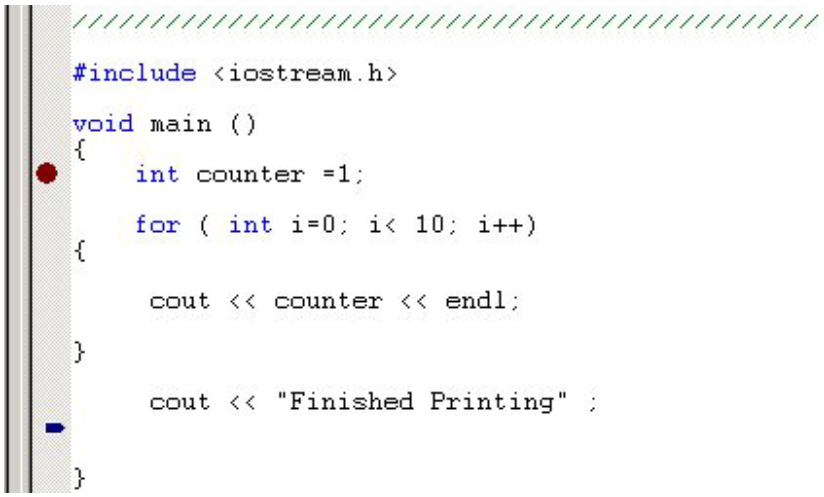
Now let us try a new example. Please start a new console project just like in the last example and cut and paste the following code into your source file.

```
////////////////////////////////////////////////
//
//  Module: breakpoint.cpp
//
//  Purpose: To demonstrate setting breakpoints
//  in an app.
//
////////////////////////////////////////////////

#include <iostream.h>

void main ()
{
    int counter =1;
    for ( int i=0; i < 10; i++)
    {
         cout << counter << endl;
    }

     cout << "Finished Printing" ;
}
```

After you build and run the program, your output should look like this:

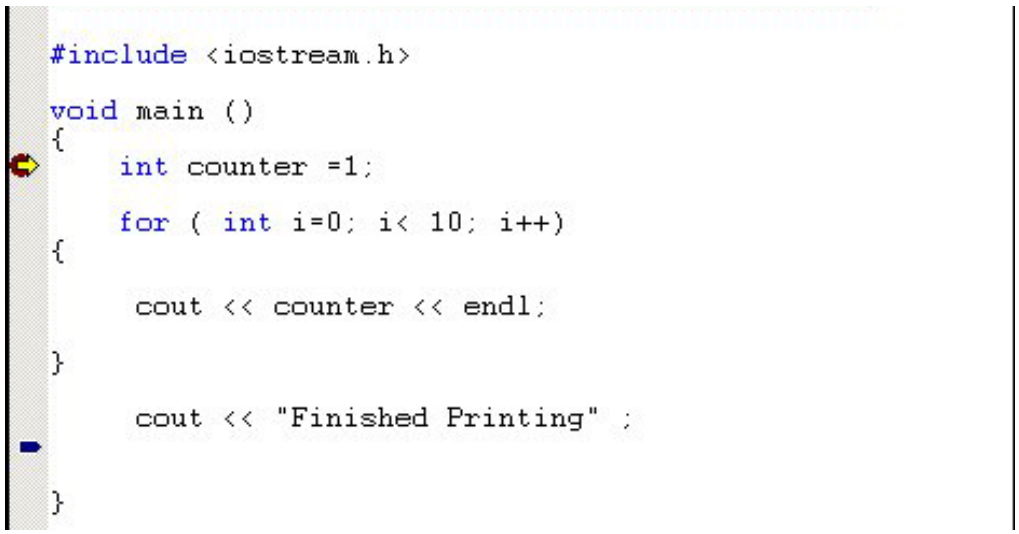


Now we will set a breakpoint and step through the code to see what is taking place. In the source window, just behind int counter = 1; right click and choose insert/remove breakpoint:



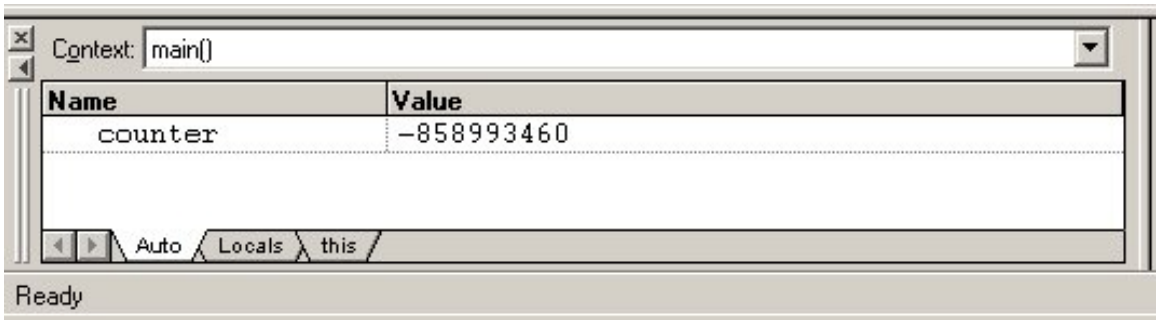
```
//////////////////////////////////////////////////
#include <iostream.h>

void main ()
{
    int counter =1;

    for ( int i=0; i< 10; i++)
{

     cout << counter << endl;

}

     cout << "Finished Printing" ;


}
```

Notice the red dot located to the left of int counter =1;. This red dot shows us the location of our breakpoint. Now from the Build menu choose debug -> Go, or just hit the F5 key to start the debugger.



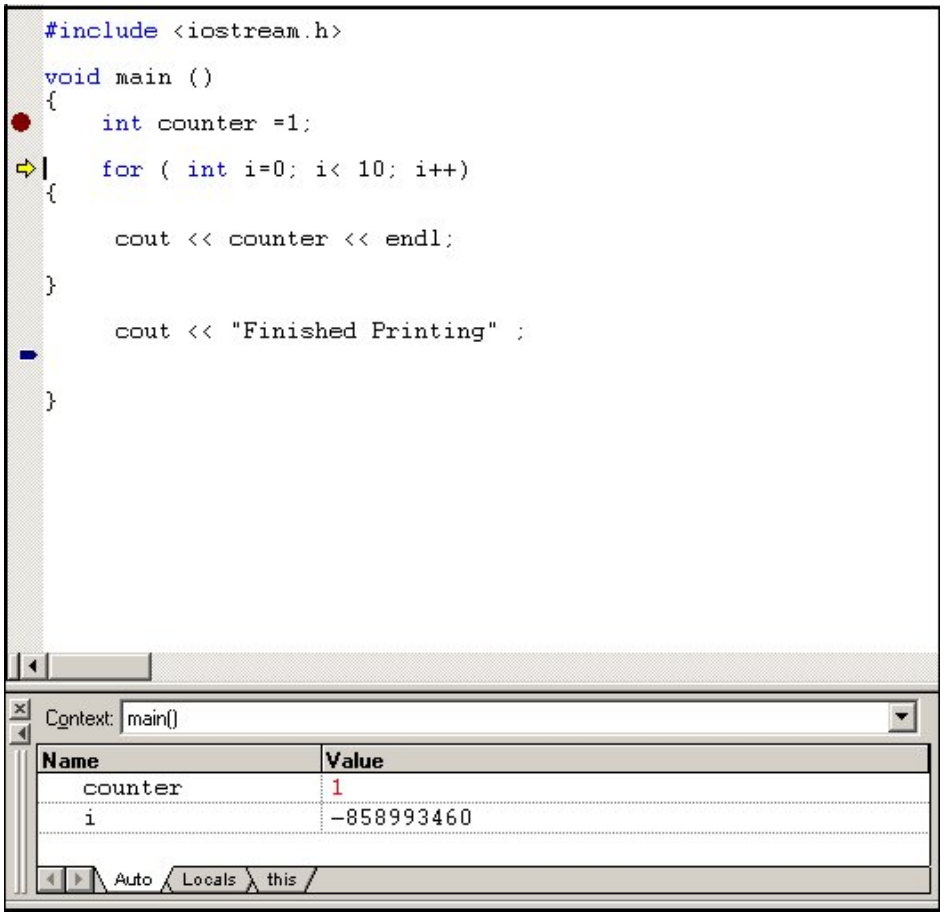
```
#include <iostream.h>

void main ()
{
    int counter =1;

    for ( int i=0; i< 10; i++)
{

     cout << counter << endl;

}

     cout << "Finished Printing" ;


}
```

The yellow arrow now points to the position where the break in execution took place (our previously placed breakpoint).



The Output window shows the current value of our variable called counter. The reason we do not see the assigned value of 1 is that the program was halted before the value was assigned. Thus we are seeing un-initialized memory (which incidentally is another very common bug that you will encounter down the road, although not problematic in this particular example).

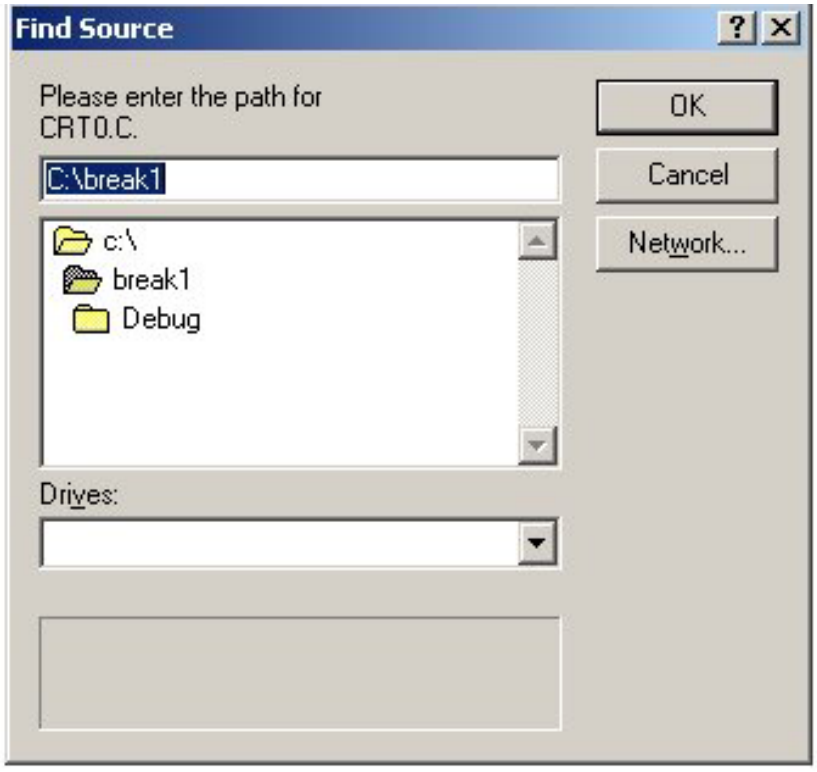
Using the F10 key, we can step past the line and observe what is taking place. Press the F10 key once and you should now see the following:



The yellow arrow has dropped down one line, and now counter stores the assigned value of 1 in the Output window. Continue to hit F10 to step through the remainder of the code

and observe the Output window to see what is taking place. It is also a good idea to expand the entries in the output window when you see the "X in a box", and observe all that is taking place. We will not go into much detail here, as we will save that for the next example. This is just to get you familiar with the procedures.

NOTE: While stepping through the code you may encounter a dialog such as this:



Simply hit cancel and you will be shown the Disassembly window at that point. Select Stop Debugging from the debugger window or the main menu access point. If you need to, run through the procedure several times, until you are comfortable with the procedure of stepping from line to line through your code and watching the local variables change in the Output window.

Debugging Example 3

Please start a new project and copy and paste the following code into the new source file. (Do not forget to check and make sure you are in debug mode for your build.)

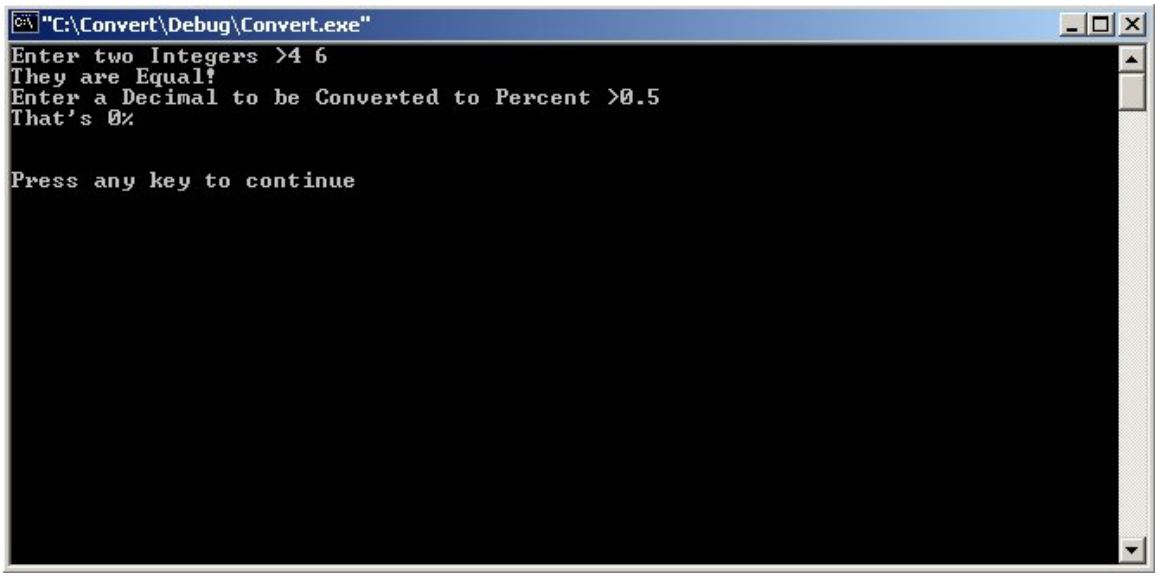
```
#include <iostream.h>
#include <stdlib.h>

int Convert (float decimal);

void main()
{
        int a, b;
```

```
        float c;
        int ThePercent;

        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;
        ThePercent = Convert(c);
        cout << "That's " << ThePercent << "%\n";
        cout << endl << endl;
        system("pause");
}

int Convert (float decimal)
{
        int result;
        result = int(decimal) * 100;
        return result;
}
```

Now build and run the program.

For the input to the convert decimal please use 0.5 as your input. This is important since we want to see the error that we will encounter with our Convert function. Also, please use 4 and 6 for the integer input in the first step.

The output tells us that our integers are equal and that our percentage is 0%. This obviously cannot be correct as we know that 4 and 6 are not equal, and we also know that 0.5 is not 0%. We've got bugs, so let us see how we can track them down.



First, let us set a breakpoint just before the user enters the two Integers:

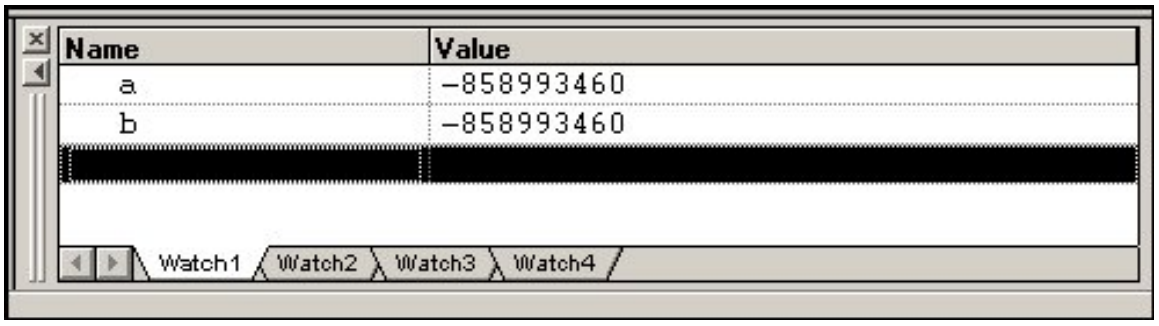
```
void main() {
        int a, b;
        float c;
        int ThePercent;

        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";
```

Now run the program in Debug mode (F5).

```
        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";
```

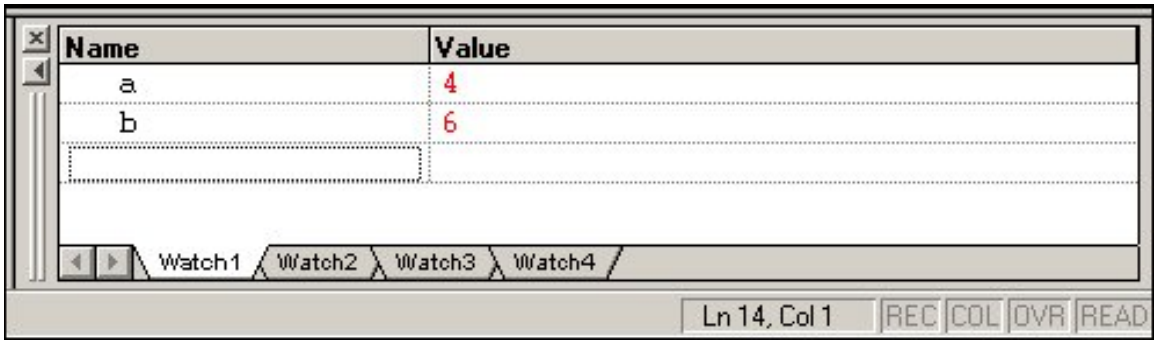
The yellow arrow means that this statement will execute *next*! Also, notice the word [break] in Visual Studio's title bar just before the title of the source file.

We now need to set up a “Watch” on the two Integer variables.
If there are already values in the Watch window, simply click the boxes under the Name column and hit the delete key to remove them. Now click the empty box and add ‘a’ as the first Name entry. Hit the return key and then enter ‘b’ in the empty box just below the ‘a’ entry.

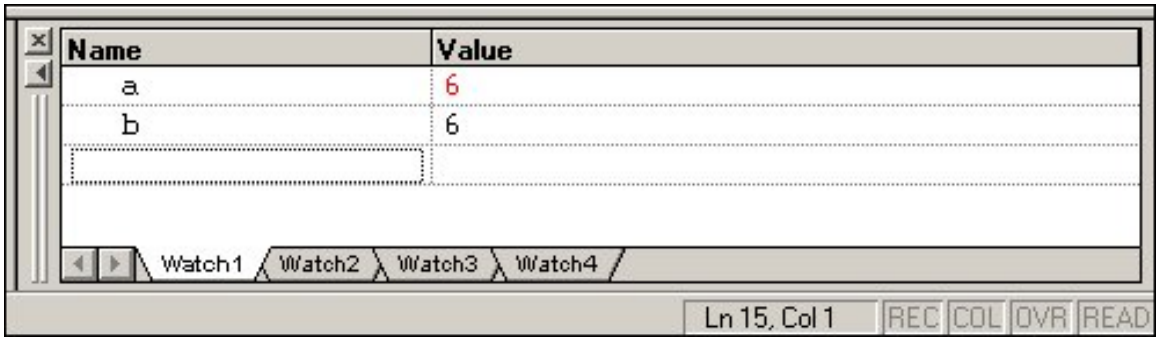


The values you see are just garbage right now, as we have not entered our two integers for comparison yet and the memory is still un-initialized.

Now press the F10 key to Step Over this line. Notice that in the Visual Studio title bar, that [break] has now changed to [run]. Since the program has been minimized to the taskbar, click once on the icon, and then enter 4 and 6 for the Integer values (place a space between 4 and 6 for readability). Next press the Enter/Return key and the Visual Studio title bar will reflect that the process is once again in the [break] state. More importantly, the Watch window now reflects the values of the two Integers that you just entered.



Step Over (F10) once more and notice the yellow line has dropped down one line, and that the value of “a” has changed in the watch window. We see that “a” and “b” are now the same value. Our program has just changed the value that we entered, so something is going on here that we certainly do not want.



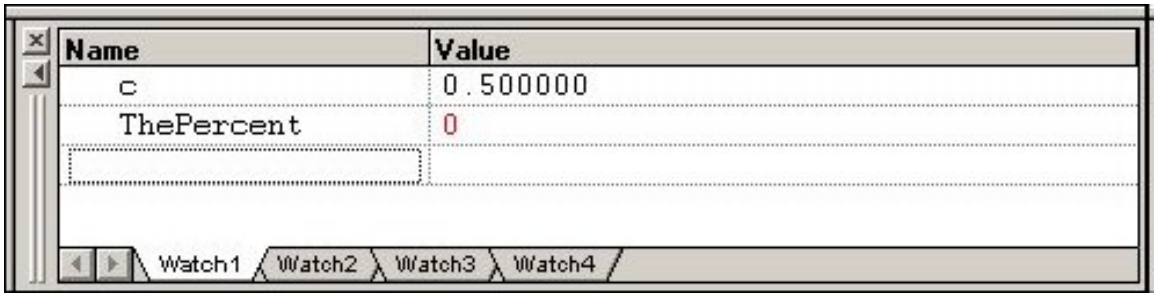
A quick look at the code shows us that our Integer problem is due to the fact that instead of using “is equal to” (==) operator, we are in fact assigning “b” to “a” in the following statement:

```
        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";
```

Stop the debugger from the debug window or the main menu access point and change the statement to reflect “is equal to” rather than an assignment. Re-build the app and run again. The first problem (assignment versus equals is another very common problem that you should be on the lookout for) should now be solved, but we still have the percentage problem to fix.

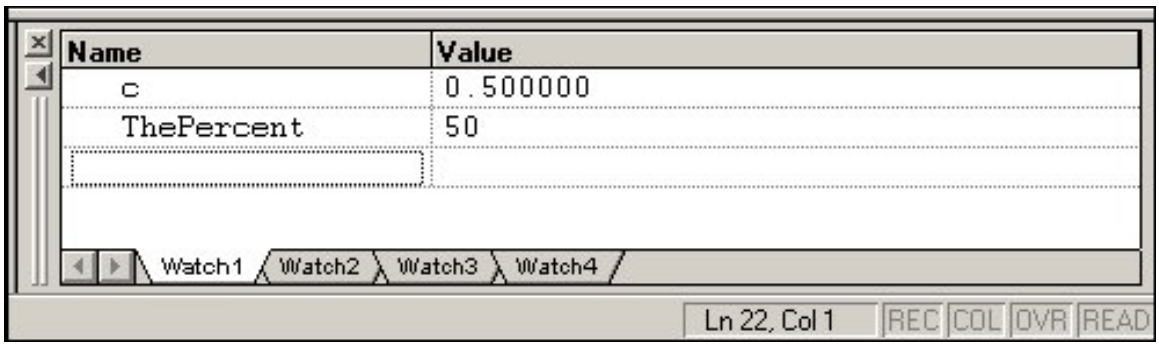
In the Watch window, you can delete the two Watches that are set (a and b) by clicking on each line and pressing "Delete", or you can just type over the existing names. Enter 'c' and 'ThePercent' in the Watch window Name columns, just as you did for “a” and “b” previously.

Now set a breakpoint at “ThePercent” just below the “cin >> c;” statement. Press F5 to start the debug run and enter values of 4 and 6 for the Integers, and 0.5 for the decimal input to the Convert portion. After you press Enter/Return, you will notice that the yellow arrow is in place and the title bar shows [break]. We could use the F10 key to Step Over, but instead we will use the F11 key to Step Into. This way we can single step into and through the function and see everything that takes place. Single step through (F11) until the yellow arrow is at the first “cout” statement. Notice the values in the watch window:



Our input is correct, but the result returned from ThePercent function is definitely off target. So the fault must lie somewhere inside the function.

As it turns out, the function is rounding off the decimal value that we input and truncating to zero prior to the multiplication by 100. We can easily remedy this by changing the location of the parentheses that are enclosing the decimal input. Place the parentheses just before the terminating semicolon on the same line. Remove any breakpoints that may remain, re-build the app, and set a breakpoint on the last cout statement. Press F5 to start the debug run and enter 4 and 6 for the integers and 0.5 for the decimal input. The app will break at the cout statement and you can observe the results in the Watch window. As you can see, it displays the correct value for “ThePercent” return result:



Stop the debugger, remove any breakpoints and run the program using Ctrl+F5. The program now functions in a normal fashion. If desired, you can now create a Release build, now that all of the bugs are cleaned up.

Conclusion

This wraps up our brief foray into the world of debugging. Hopefully you have gleaned at least a working knowledge of the processes involved in debugging apps. While being very simple, the example programs we looked at are good practice to get you up to speed. Run through the samples as many times as you need to until you are comfortable with what has been presented here. You should make changes and experiment with them as much as you can stand. The more time you put in now working with smaller examples, the better off you will be down the road. Later on, as you begin debugging more complex applications, you will find that the process is basically the same.

It is worth noting that MSDN (online or from the local install if you have it) includes a series of very good articles by John Robbins called “Bugslayer.” You should find these articles to be very helpful in continuing to expand your knowledge in this area and they are highly recommended reading.

Good luck to you with your projects!