Debugging with Visual Studio 2005

The purpose of this handout is twofold: to give you a sense of the philosophy of debugging and to teach you how to use some of the practical tools that make testing and debugging easier.

The philosophy of debugging

Debugging is one of the most creative and intellectually challenging aspects of programming. It can also be one of the most frustrating. To a large extent, the problems that people face debugging programs are not so much technical as they are psychological. To become successful debuggers, you must learn to think in a different way. There is no cookbook approach to debugging, although Nick Parlante's rules below will probably help. What you need is insight, creativity, logic, and determination.

The programming process leads you through a series of tasks and roles:

Design	—	Architect
Coding	—	Engineer
Testing	—	Vandal
Debugging	—	Detective

These roles require you to adopt distinct strategies and goals, and it is often difficult to shift your perspective from one to another. Although debugging is extremely difficult, it can be done. It will at times take all of the skill and creativity at your disposal, but you can succeed if you are methodical and don't give up on the task.

Debugging is an important skill that you will use every day if you continue in Computer Science or any related field. Even though it is the final task of those listed above, it is certainly not the least important. Debugging will almost always take more time than the first three tasks combined. Therefore, you should always plan ahead and allow sufficient time for testing and debugging, as it is required if you expect to produce quality software. In addition, you should make a concentrated effort to develop these skills now, as they will be even more important as programs become more complicated later in the quarter and later in your career as a programmer.

The Eleven Truths of Debugging

1. Intuition and hunches are great—you just have to test them out. When a hunch and a fact collide, the fact wins.

2. Don't look for complex explanations. Even the simplest omission or typo can lead to very weird behavior. Everyone is capable producing extremely simple and obvious errors from time to time. Look at code critically—don't just sweep your eye over that series of simple statements assuming that they are too simple to be wrong.

3. The clue to what is wrong in your code is in the values of your variables and the flow of control. Try to see what the facts are pointing to. The computer is not trying to mislead you. Work from the facts.

4. Be systematic. Be persistent. Don't panic. The bug is not moving around in your code, trying to trick or evade you. It is just sitting in one place, doing the wrong thing in the same way every time.

5. If you code was working a minute ago, but now it doesn't—what was the last thing you changed? This incredibly reliable rule of thumb is the reason your section leader told you to test your code as you go rather than all at once.

6. Do not change your code haphazardly trying to track down a bug. This is sort of like a scientist who changes more than one variable at a time. It makes the observed behavior much more difficult to interpret, and you tend to introduce new bugs.

7. If you find some buggy code which does not seem to be related to the bug you were tracking, fix the bug code anyway. Many times the buggy code is related to or obscures the original bug in a way you had not imagined.

8. You should be able to explain in Sherlock Holmes style the series of facts, tests, and deductions which led you to find a bug. Alternately, if you have a bug but can't pinpoint it, then you should be able to give an argument to a critical third party detailing why each one of your functions cannot contain the bug. One of these arguments will contain a flaw since one of your functions does in fact contain a bug. Trying to construct the arguments may help you to see the flaw.

9. Be critical of your beliefs about your code. It's almost impossible to see a bug in a functions when your instinct is that the function is innocent. In that case, only when the facts have proven without question that the function is the source of the problem will you be able to see the bug.

10. You need to be systematic, but there is still an enormous amount of room for beliefs, hunches, guesses, etc. Use your intuition about where the bug probably is to direct the order that you check things in your systematic search. Check the functions you suspect the most first. Good instincts will come with experience.

11. Debugging depends on an objective and reasoned approach. It depends on overall perspective and understanding of the workings of your code. Debugging code is more mentally demanding than writing code. The longer you try to track down a bug without success, the less perspective you tend to have. Realize when you have lost the perspective on your code to debug. Take a break. Get some sleep. Many times a programmer can spend hours late at night hunting for a bug only to finally give up at 4:00 A.M. The next day, they find

> the bug in 10 minutes. What allowed them to find the bug the next day so quickly? Maybe they just needed some sleep and time for perspective. Or maybe their subconscious figured it out while they were asleep. In any case, the "go do something else for a while, come back, and find the bug immediately" scenario happens too often to be an accident.
>
> — Nick Parlante, Stanford University

Testing incrementally

When dealing with any program containing more than a couple of functions, we highly recommend that you test incrementally. This might include writing test code that passes values to a function, and gets its return value. Doing this allows you to test your functions in isolation, allowing you to hone in on bugs before putting everything together. Realizing there's a bug in a specific 10-line function by seeing the results of a very simple test program is much easier than testing it in the context of the rest of your program, where it is no longer isolated and much harder to tell whether a bug is a result of that function, or any other one. This makes debugging much less painful in the long run.

Assessing the symptoms

For programs that use random numbers, it is important to debug the program in a deterministic environment. A deterministic environment means that a piece of code will behave the same way every time you run it. For this reason, it is essential to take the `Randomize` statement out of the main program, usually by enclosing it in comment markers. You want it there eventually, but not while you're debugging. You want your program to work the same way each time, so that you can always get back to the same situation.

Keeping an open mind

One of the most important techniques to master about debugging is keeping an open mind. So often, the problems that keep your code from working are easy to see if you can simply overcome the psychological blinders that keep you from seeing them. The more you are sure that some piece of code is correct, the harder it is to find the bugs in it.

Dealing with pointers

Programs involving pointers can be very tricky to debug, because errors in pointer handling can lead to very strange consequences. With pointers, you can delete something twice and/or use a variable after it has been freed, run off the end of the array, etc., and you can get away with it at the time, but later the ill effects begin to show. This is why it is so important to have your memory model straight before you start coding, and it doesn't hurt to make a memory diagram to make sure you understand what you want, and compare it to what you have actually coded. Also, be extra careful when scrutinizing allocations and initializations of pointers.

Debugging with `cout`

Interspersing code with `cout` statements can be a quick and easy way to check the value of a variable name at a certain point in a program, or to see whether or not a specific point of execution is being reached. It is important to note that the way `cout` works may not be quite what you would expect. Usually there is a buffer that stores the data piped to calls to `cout`. The

data in the buffer is only displayed on the screen once **endl** is sent, or a function requiring input is called (like **GetInteger()** or **GetLine()**).

For example:

```
int main()
{
      int i, int j;

      i = ComplicatedFn();
      cout << "got past ComplicatedFn(); i = " << i << ".";       // debug

      j = ComplicatedFn2();
      cout << "got past ComplicatedFn2(); j = " << j << ".";      // debug

      ComplicatedFn3(&i, &j);
      cout << "done: i = " << i << ", j = " << j << endl;       // debug

      return 0;
}
```

Assuming no other I/O calls are in the other functions, no **cout** statements will print until the last **cout** call is executed, since it has the first (and only) **endl** call. Therefore, if your program crashes in the middle of **ComplicatedFunction2()**, the output window will not have printed out anything, even though a **cout** call had been passed. Therefore, it is always a good idea to attach **endl** to all debugging statements, so the buffer is flushed and the debug statements print to the screen when they are passed.

Even though **cout** statements may seem like a sufficient means for debugging, it becomes largely inadequate for more complicated programs, especially those which make many iterations over the same code during a single program run. In cases where **cout**-style debugging is largely inefficient and/or inadequate, we turn to the debugger.

Using an online debugger

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 Microsoft Visual Studio debugger

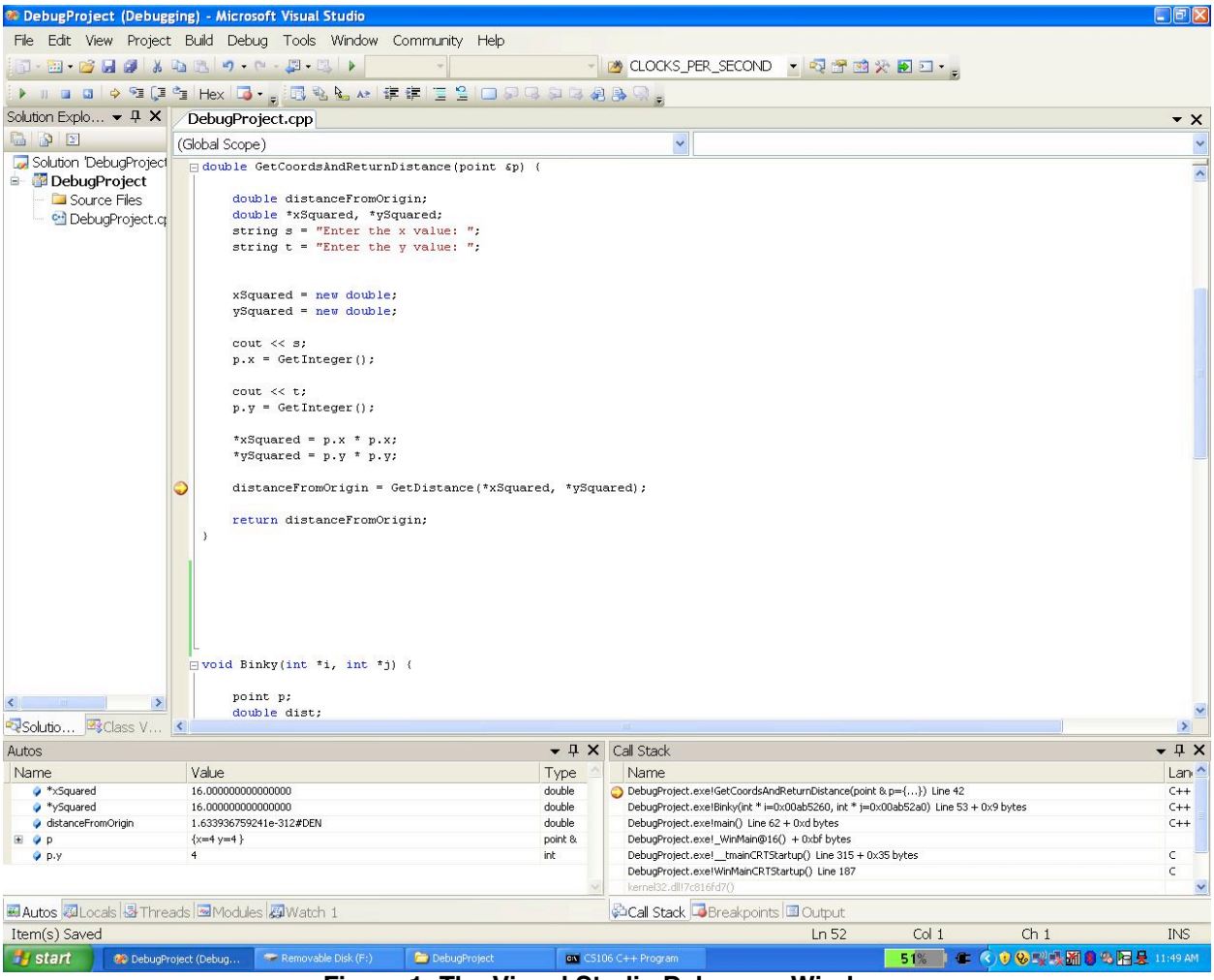

Figure 1: The Visual Studio Debugger Window

The Visual Studio debugger is a complicated debugging environment, but with a little patience, you should be able to learn it to the point where you can produce correct 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 mid-stream, 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**, respectively:

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**.

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 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.

Once your program is stopped, the top pane of the debugger window will show the current function that is executing and a yellow arrow to the left of the code shows the next line to be executed. Choosing **Continue** would cause your program to continue executing freely while the three Step buttons allow you to manually control what to execute and then return control to the debugger. **Step Over** is useful whenever you know that the function on that line works and that the bug is further on. For example, you would hate to trace through the steps involved in each call to `cout`, and the **Step Over** button allows us to skip all of the details. In Figure 1, calling **StepOver** would execute the `GetDistance()` call and assign it to `distanceFromOrigin` before control returns to the debugger after executing that line. 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 1, the debugger would create thenew stack frame for the `GetDistance()` function and return control back to the debugger at the first statement of `GetDistance()`. The **Step Out** command executes the remainder of the current function and returns control to the debugger once that function returns.

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

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); Our current function is the topmost entry. If we look into the list, we see that `main()` called function `Binky()` which took two `int *`s, which called `GetCoordsAndReturnDistance()` which took a reference to a `point`. When we click a function in the stack frame list, the other panes update to show the code and variables for the selected function. This allows us to look at the variables and execution path of all

The bottom-left pane contains all variables and associated values contained in the stack frame highlighted in the `Call Stack` list. In Figure 1, we see `distanceFromOrigin` has quite a strange value. Looking at the code (and using our debugging skills) we can see that `distanceFromOrigin` has yet to be initialized at this point in execution and thus has a junk value. Also note that some variable types (such as structs, classes, and pointers) have a plus sign to their left. This means that there is more information for that data type that can be viewed. By clicking on the plus sign next to `p`, which is a variable of type `point`, we see the values for the x and y variables, which are members of the `point` struct. Note that `ySquared` is listed as `*ySquared`. This is because `ySquared` is a pointer and therefore its actual, literal value is not what we are interested in. It's value is the location of the chunk of memory it points to; what we are really interested in is the value stored there. This value is `*ySquared`, which is why the debugger displays this instead of `ySquared`'s literal value. 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.

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.

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`**. 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).

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. You can try stepping 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. Wash, 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 the stack frame and the variable values in earlier function calls, and do some serious detective work to see what went wrong.

A call to **`Error()`** actually quits the program, and thus renders the debugger useless, since **`Error()`** basically short-circuits the program to immediately exit. When this happens, you should put a breakpoint right on the line causing **`Error()`** statement to be triggered, so you can poke around and see what's up immediately before **`Error()`** is called.

Sometimes it is not obvious at all as to what is going on and you don't know where to start. **`Error()`**s 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? Divide and conquer to the rescue! Set a breakpoint at roughly halfway through your program's code path. 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/incorrect value, you know that there's a problem in code that led to this point – restart and set a breakpoint halfway between the beginning of the code path and the first breakpoint. If everything looks ok 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 `Histogram(Vector<int> &values, Vector<int> scores)` where each bucket represents a range from the minimum score to the maximum score. When a score falls in the range of that specific bucket, the bucket is incremented by one.

An example of an edge case to test would be to have 0 scores. Another would be to have 0 buckets. Does the function handle the case where one or more of the score values may be zero or negative? More than 100? What if the difference between the lowest score and highest score is an odd number? Thinking of the assumptions you've made about the input to a function and writing tests that violate those assumptions can lead to a healthy testing of edge cases.

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.

Source: http://see.stanford.edu/materials/icspacs106b/H07P-DebuggingWithVS.pdf