UMass

Electrical and Computer
Engineering Department

ECE 264, Object-Oriented Software Development

Lab Assignment 1 **Monday Session:** Due Time: 5:00 pm, 01/30/2013 (Wednesday)
Wednesday Session: Due Time: 5:00 pm, 02/01/2013 (Friday)

1. Introduction and Acknowledgements

This lab assignment helps you to learn how to set up a Microsoft Visual Studio C++ project, using the debugger, and understand some questions you may encounter throughout the term. This document, which was revised by Professor Honggang Wang, in fall 2012, is based heavily on the previous ECE 264 Lab Manual, written by Prof. Michael Geiger in spring 2011, and by Makia Powell, an ECE 264 TA under Professor Hong Liu, in Spring 2008.

2. Deliverables

Repeat following the procedures described from section 4 to section 5 and output all the similar figures from figure 1 to figure 18. To acquire these figures, you can do screen copy (using "PrintScr" key) and put all the screen copies (figures) in a single word file. Submit your word format file to your class directory on the M:\ ECE-264. Call your project *lab1_VisualStudio*, and ensure that your source file name is ***lab1_** VisualStudio**.docx***. You should place only the .docx file on the M:\ECE264. Failure to meet this specification will reduce your grade, as described in the ECE 264 lab grading handout, which you are strongly encouraged to read before starting the lab.

3. Getting Started

First, log into your Engineering account. Most of your work should be stored on your Z:\ drive, but lab submissions will be placed in your class folder on the M:\ drive. You can get to both of these by double-clicking on "My Computer."

From outside the labs, or via VPN, you can access your M:\ drive by using the "Run" program and entering:

\\134.88.53.57\class\ECE-264\<username>

You can access your Z:\ drive by entering:

\\134.88.53.58\<username>

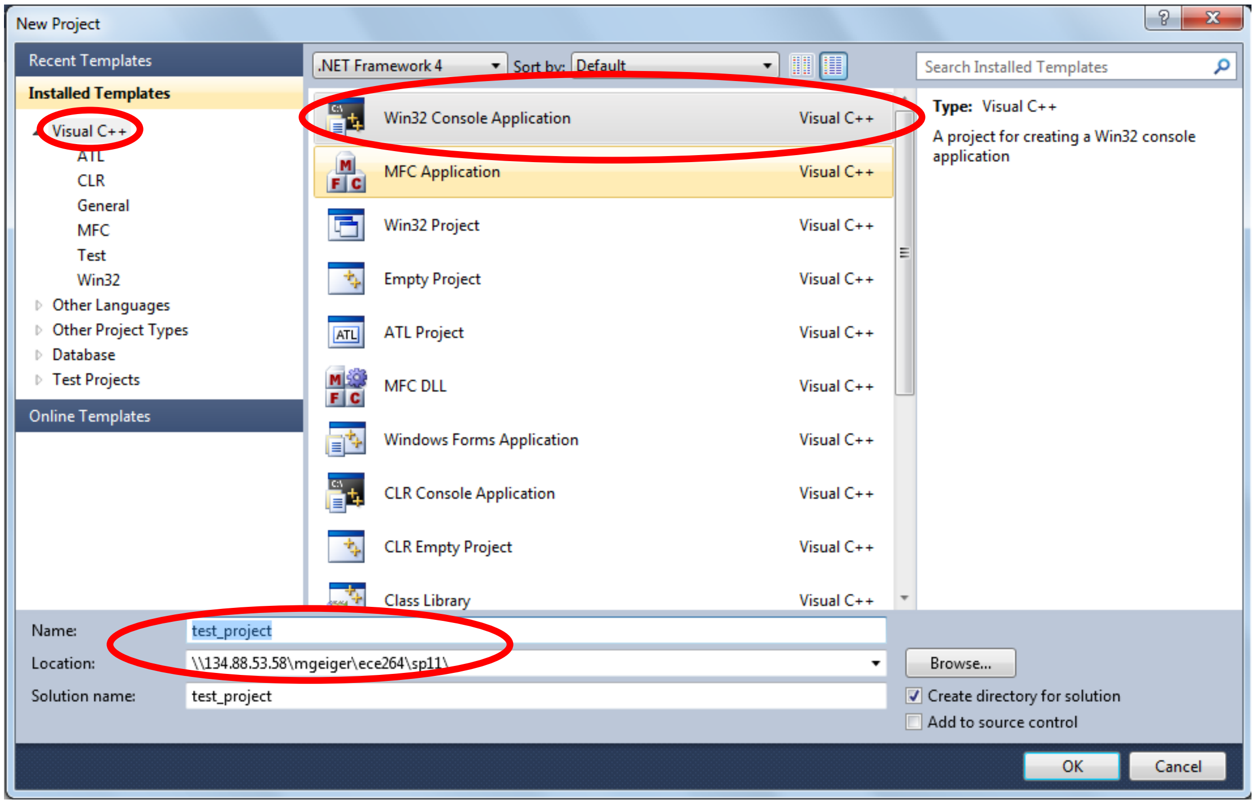
In both cases, replace <username> with your UMass Dartmouth username.

4. Launching Visual Studio

We will create a sample project to help illustrate the use of Visual Studio. Note that this tutorial assumes the use of Visual Studio 2010.

After starting Visual Studio, select **File→New→Project** from the main menu.

The dialog window that appears allows you to choose the type and name of your project. After selecting **Visual C++** in the list of templates on the left, choose **Win32 Console Application** from the list of project types in the middle. Use the boxes at the bottom of this window to specify a name and location for your project. Remember, you should store your work on the Z:\ drive until you are ready to submit your files.



.NET Framework 4 | Sort by: Default

Type: Visual C++

A project for creating a Win32 console application

Name: test_project

Location: \\134.88.53.58\mgeiger\ece264\sp11\

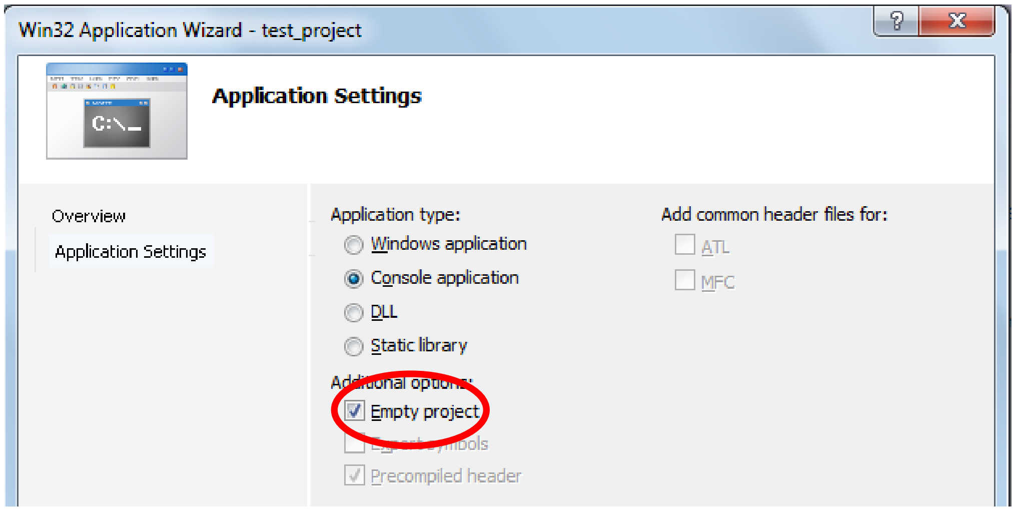
Solution name: test_project

[✓] Create directory for solution

[ ] Add to source control

Figure 1: Creating a new Win32 Console Application

After accepting these settings, a window appears that you can use to set application settings. Click **Next**, then select the check box next to **Empty project**, which is under **Additional options**, in the following window. Click **Finish** to create your project.



Application type: [✓ Console application]

Additional options: [✓ Empty project]

Figure 2: Initial application settings

To create your first C++ file, right click on **Source Files**, in the **Solution Explorer** window. Choose **Add → New Item**. In the list that appears, choose **C++ file (.cpp)**, then name your file. Note that, in your lab assignments, file names are specified for you.

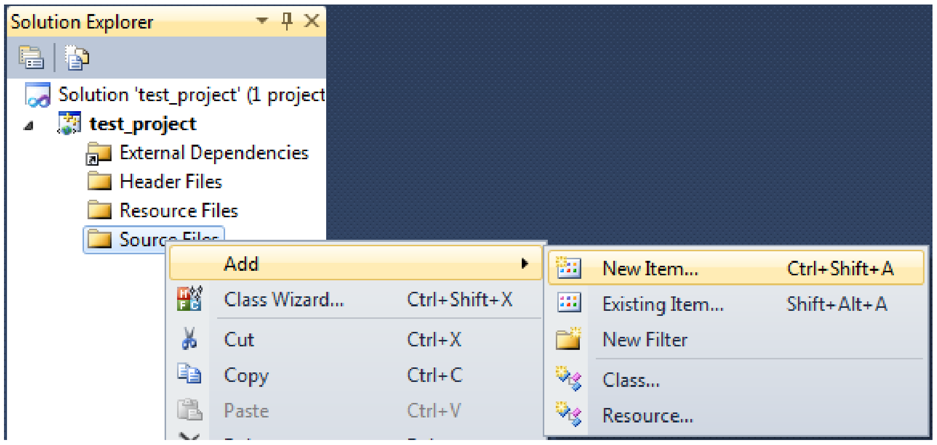


Figure 3: Adding a new source file to your project

Note that you can add existing files to your project, which can be useful when reusing code from previous projects. Right click on the appropriate files folder (typically Source Files or Header Files), then choose **Add → Existing Item**. Find the desired file, then click **Add** to add it to the project.

5. Writing, Compiling, and Executing Programs

Let's create a simple application that handles basic input and output to illustrate the process of creating a Visual Studio application. Enter the following in the code window:

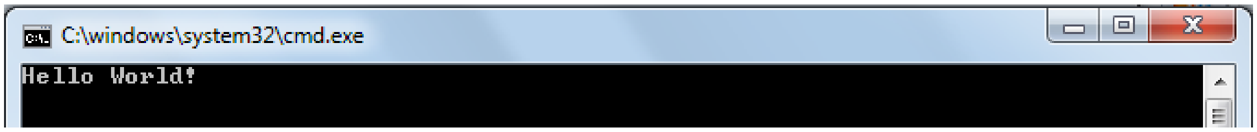
```
// Test program
#include <iostream>

using std::cout;
using std::endl;
using std::cin;

int main() {
    int a;
    cout << "Hello World!" << endl;
    cin >> a;
    cout << "a = " << a << endl;
    return 0;
}
```

Notice that Visual Studio color-codes parts of your program—C++ keywords are blue, strings and library names are red, and comments are green.

To compile your code, choose **Build→Build Solution** in the main menu (or press **F7**).

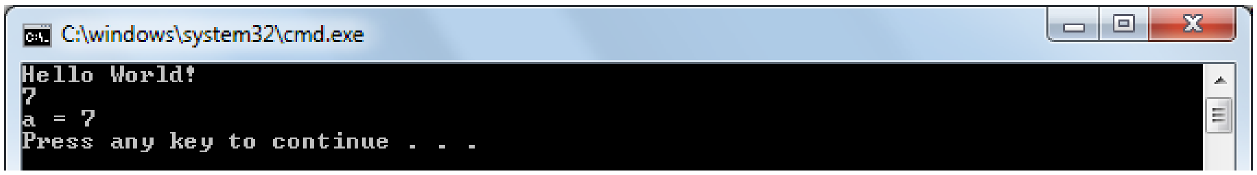
To run your program, choose **Debug→Start Without Debugging** (**Ctrl+F5**). This option forces your program to pause when finished, allowing you to view all output, as shown in Figure 5. If you simply press the green "play" button, which corresponds to **Debug → Start Debugging** (**F5**), your program will run until it encounters a breakpoint. If you have not set any breakpoints, the program will run to completion and exit without allowing you to view the output.

Note that your program requires input—the statement cin >> a; causes the program to wait for the user to enter at least one non-whitespace character and press Enter.



```
Hello World!
```

Figure 4: Output of basic program prior to user input



```
Hello World!
7
a = 7
Press any key to continue . . .
```

Figure 5: Final output of basic program. Note that the '7' is user input.

6. Debugging in Visual Studio

Executing Code Line-by-Line

You have two basic options for line-by-line code execution: **Step Into** (**F11**) or **Step Over** (**F10**). For most code, these operations will behave the same. **Step Into** can be used to enter and step through functions, which can then be exited using a third option: **Step Out** (**Shift+F11**). **Step Over** will execute each line without entering any functions. All three options can be found in the **Debug** menu; note that **Step Out** is only available when actively debugging a program.

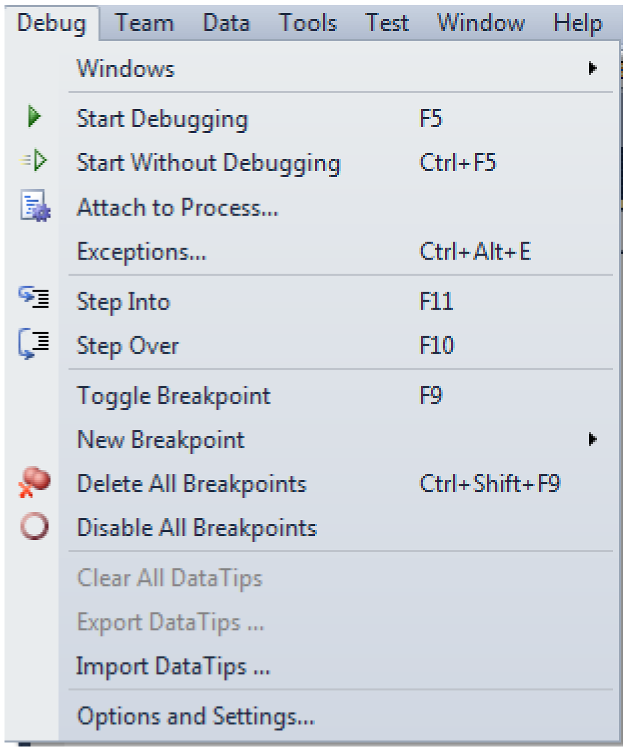


Figure 6: The Debug menu. Note that the Step Out command is not shown because no program is being actively debugged

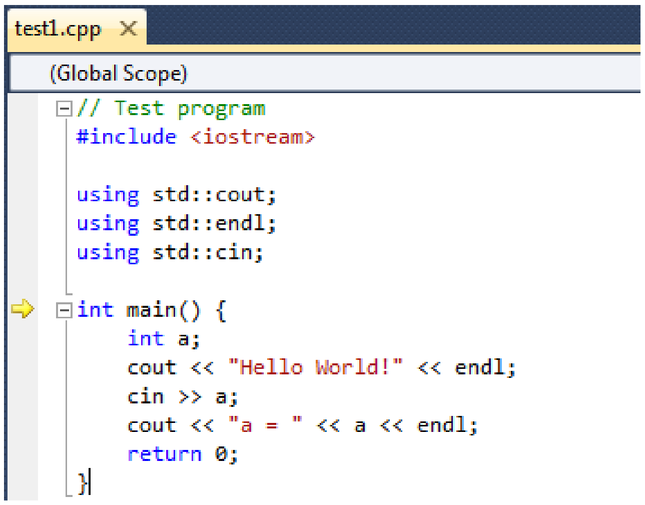
Using Breakpoints

Breakpoints are preset stopping points within your code used for debugging purposes. Setting these points allows you to go through code more quickly than line-by-line execution, making it easier to efficiently identify the source of a problem.

Breakpoints can be set or cleared in two ways: by choosing **Debug→Toggle Breakpoint** (**F9**) to toggle a breakpoint at a selected line, or by clicking the gray area directly to the left of the desired line. As shown in Figure 8, a red circle to the left of a line indicates a breakpoint has been set.

To practice the above techniques, start executing your program by using the **Step Into** command once. As shown in Figure 7, the yellow arrow that appears to the left of the main function indicates which statement will execute next. (Since this "statement" is the start of a function, what will actually execute is the first line of that function.) Note your console window has no output—the cout statement has not yet executed.

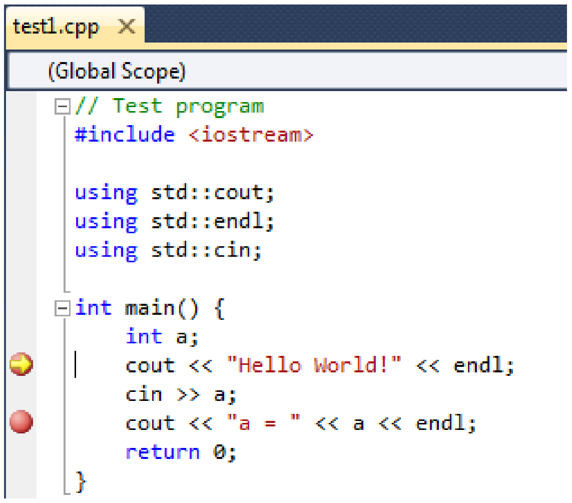
If you use **Step Into** again, you will see that the arrow moves to the second line of code. Your program has executed its first line—declaring the integer a.



```
// Test program
#include <iostream>

using std::cout;
using std::endl;
using std::cin;

int main() {
    int a;
    cout << "Hello World!" << endl;
    cin >> a;
    cout << "a = " << a << endl;
    return 0;
}
```

Figure 7: Program after executing "Step Into" once. The arrow indicates the next statement to be executed—in this case, the first line of the designated function

Now, set a pair of breakpoints as shown in Figure 8, using either of the methods described above. Use the **Continue** command; the program will wait for your input and then run to the next breakpoint. Use **Continue** one more time to complete the program. Note that your breakpoints remain set for future use.

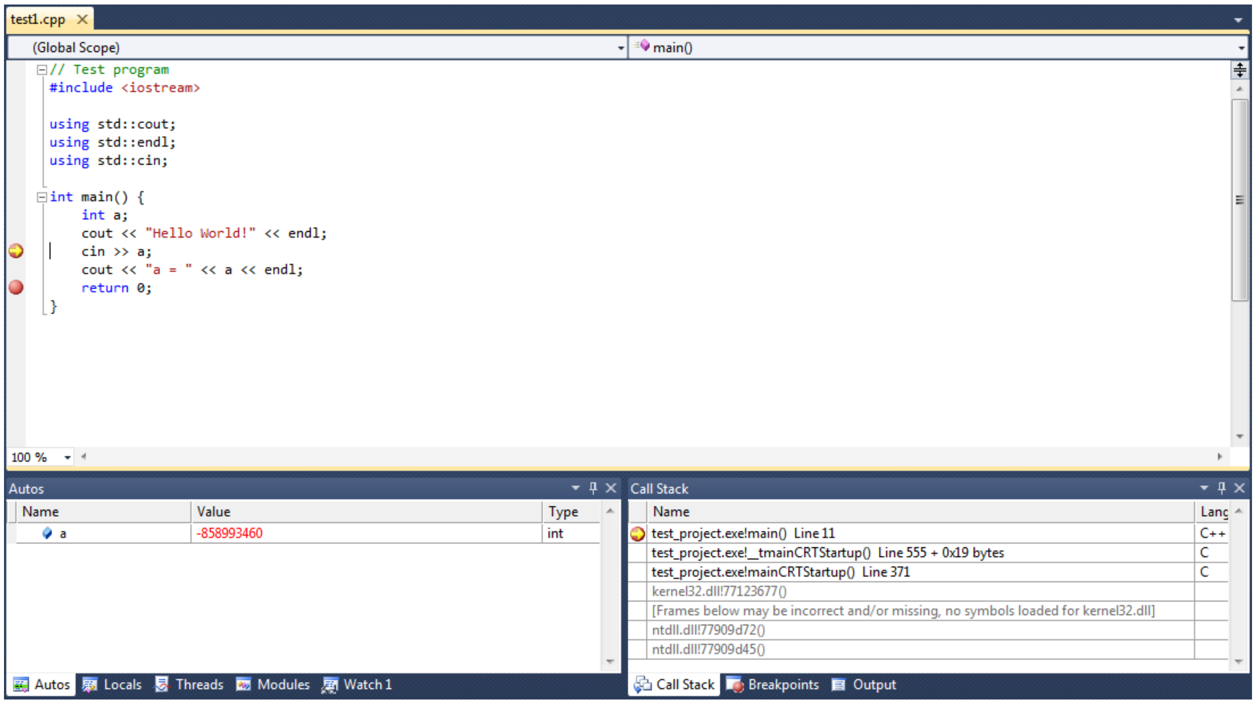


```
// Test program
#include <iostream>

using std::cout;
using std::endl;
using std::cin;

int main() {
    int a;
    cout << "Hello World!" << endl;
    cin >> a;
    cout << "a = " << a << endl;
    return 0;
}
```

Figure 8: Basic program with breakpoints set

Tracking Variables Automatically

Visual Studio automatically provides two debugging windows. The Call Stack window shows what functions have been called, with the most recent function at the top. The Autos window shows all variables used in the current and previous statements, which allows you to see how their values change during program execution. Both windows are shown in Figure 9.



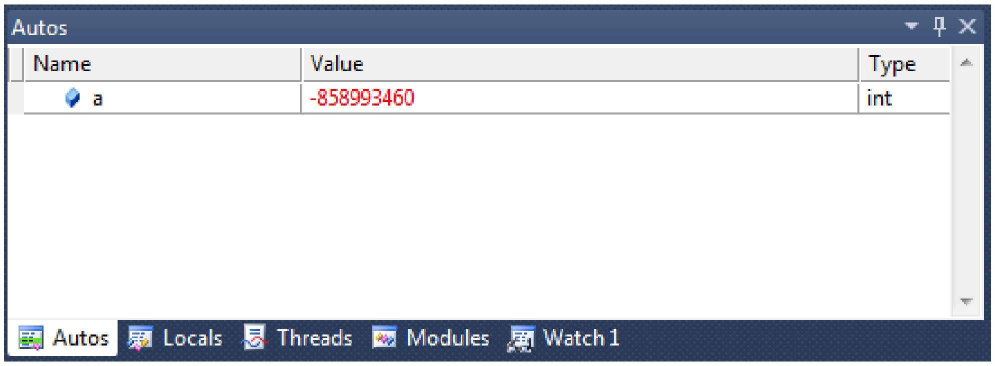
```
// Test program
#include <iostream>

using std::cout;
using std::endl;
using std::cin;

int main() {
    int a;
    cout << "Hello World!" << endl;
    cin >> a;
    cout << "a = " << a << endl;
    return 0;
}
```

Name	Value	Type
a	-858993460	int

Name	Lang
test_project.exe!main() Line 11	C++
test_project.exe!__tmainCRTStartup() Line 555 + 0x19 bytes	C
test_project.exe!mainCRTStartup() Line 371	C
kernel32.dll!77123677()	
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dll]	
ntdll.dll!77909d72()	
ntdll.dll!77909d45()	

Figure 9: Basic program at the start of debugging. The Autos window is on the bottom left; the Call Stack window is on the bottom right

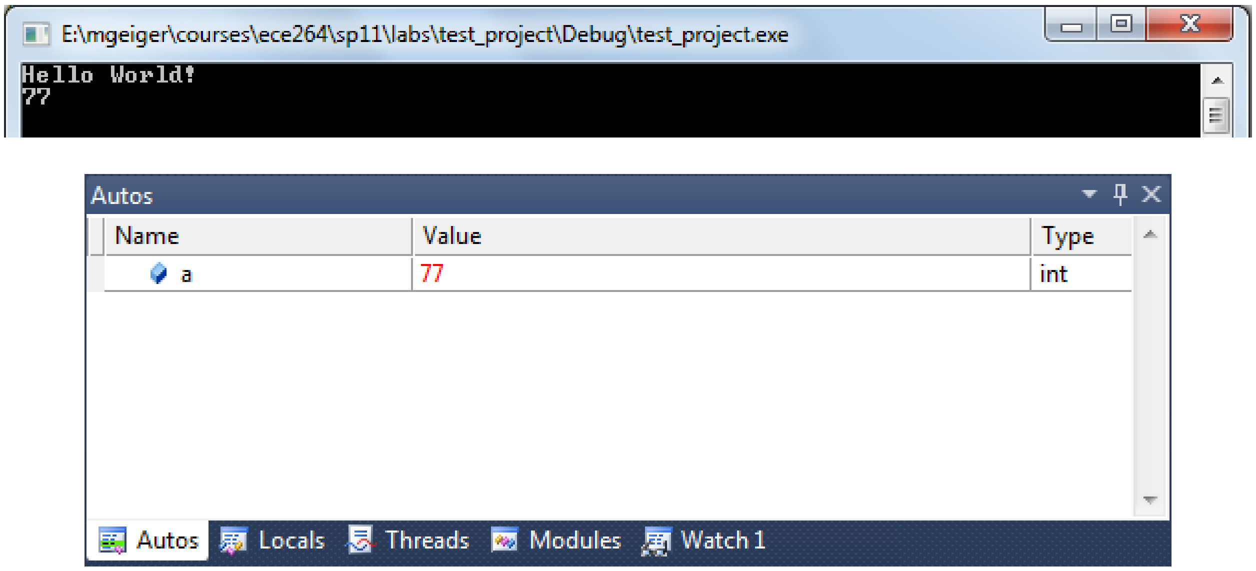
We will use the Autos window to track the value of the integer a. Choose **Debug→Start Debugging** (**F5**); the program will run to the first breakpoint. a now appears in the Autos window. Its value is red, which77 indicates that it has recently changed; however, since the variable is uninitialized, the current value is meaningless.



Name	Value	Type
a	-858993460	int

Figure 10: The Autos window after the integer a has been declared. Note that this variable is currently uninitialized

Now, use **Continue** to run to the second breakpoint. Use the console window to enter an input value, then return to Visual Studio. Your program has stopped at the second breakpoint, and the Autos window shows the value of a that you entered.


E:\mgeiger\courses\ece264\sp11\labs\test_project\Debug\test_project.exe

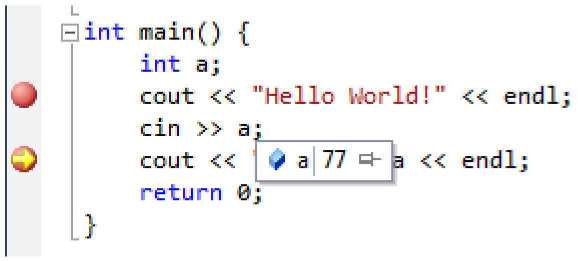
```
Hello World!
77
```

Name	Value	Type
a	77	int

Autos
Locals
Threads
Modules
Watch 1

Figure 11: The console and Autos window at the second breakpoint. The value of a matches the user input in the console window

Note that you can also see the value of a variable at any time during debugging by pointing your cursor at the variable, as shown in Figure 12.



```
int main() {
    int a;
    cout << "Hello World!" << endl;
    cin >> a;
    cout << ' a 77 a << endl;
    return 0;
}
```

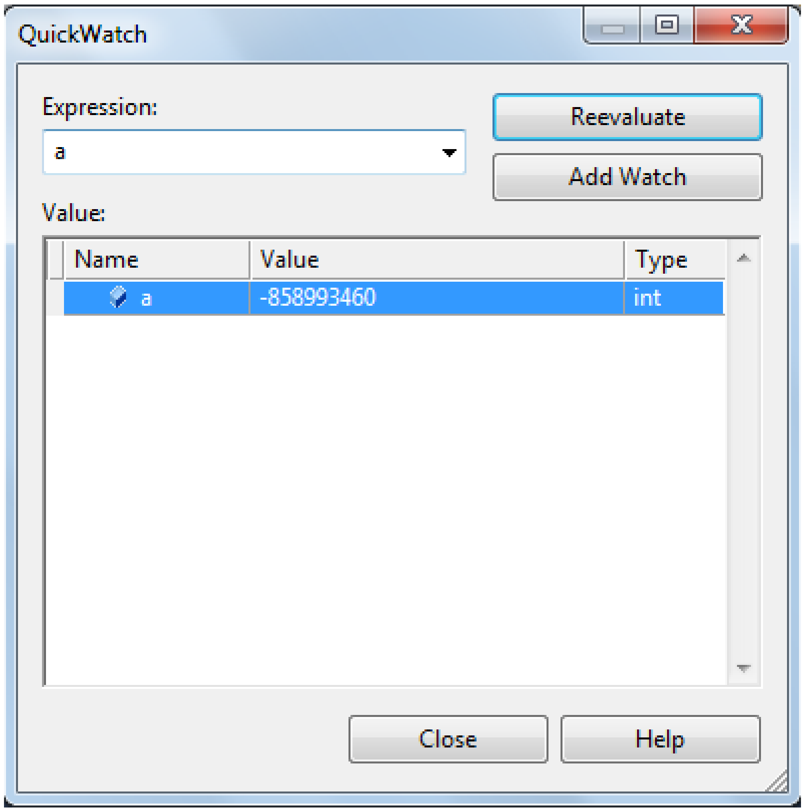
Figure 12: Viewing variable value using cursor

Tracking Variables using Watches

Recall that the Autos window only shows variables used in the current and previous statement. Visual Studio allows you to "watch" a variable to ensure that you can always see its value within a window.

QuickWatch provides one way of watching variables. To use this tool, select **Debug → QuickWatch** while debugging a program. The resulting window allows you to enter any expression involving your variables and get its current value.

To test this method, **Start Debugging** your basic program; once it reaches the first breakpoint, select **Debug → QuickWatch**. Enter "a" as the **Expression** and click **Reevaluate**. The current value of a will appear.

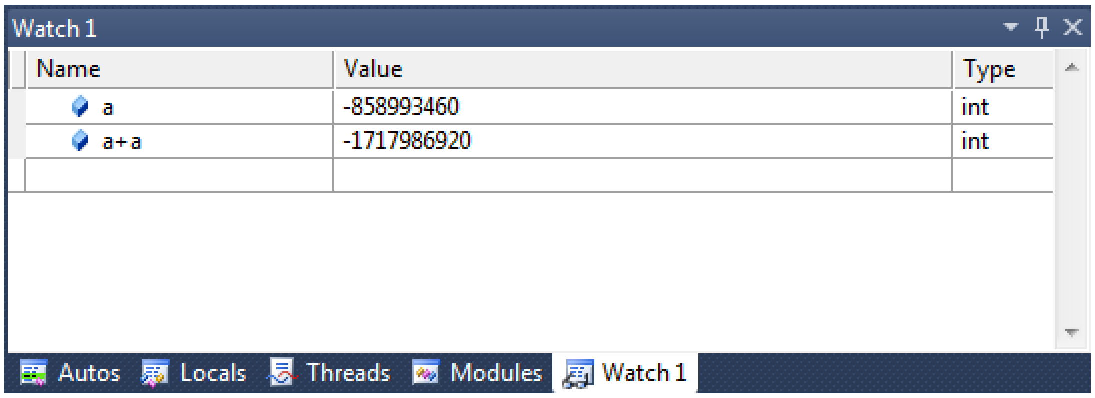


Expression: a

Name	Value	Type
a	-858993460	int

Figure 13: The QuickWatch window showing the value of a at the first breakpoint

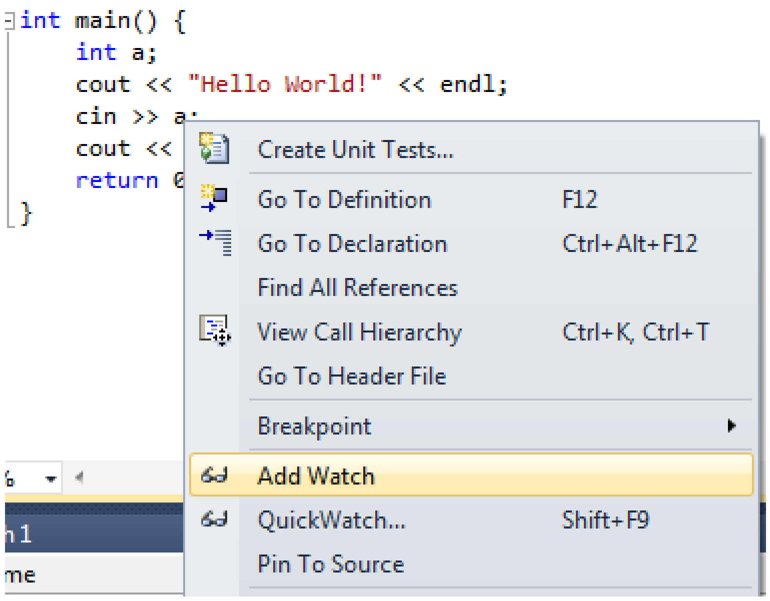
Using the QuickWatch window halts program execution. To track the value of an expression throughout the program, select the expression and click **Add Watch**. The expression appears in the **Watch 1** window, which shares the same space as the Autos window. You can toggle between the two using the tabs below that window, as shown in Figure 14.



Name	Value	Type
a	-858993460	int
a+a	-1717986920	int

Figure 14: The Watch window after adding two expressions--the value of a alone, and the sum a+a. Note that the tabs below this window allow you to switch between the Watch and Autos windows

To watch a variable without opening QuickWatch, you can highlight the variable, right click, and choose **Add Watch**; the variable will once again appear in the Watch window.



```
int main() {
    int a;
    cout << "Hello World!" << endl;
    cin >> a;
    cout <<
    return 0
}
```

Figure 15: Adding a watch for a variable without using QuickWatch

To remove an expression from the Watch window, right click on the expression and select **Delete Watch**.

Tracking Array Variables

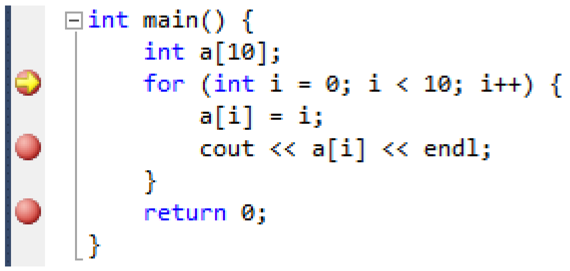
The debugger also allows you to view all values within an array. To test this functionality, enter the program below:

```
// Test program 2
#include <iostream>

using std::cout;
using std::endl;

int main() {
    int a[10];
    for (int i = 0; i < 10; i++) {
            a[i] = i;
            cout << a[i] << endl;
    }
    return 0;
}
```

You may overwrite your existing code or start a new file; if starting a new file, be sure to remove the first file from your project by right clicking it and selecting either **Exclude From Project** or **Remove**. A project that has two source files with the main() function will generate a linker error.

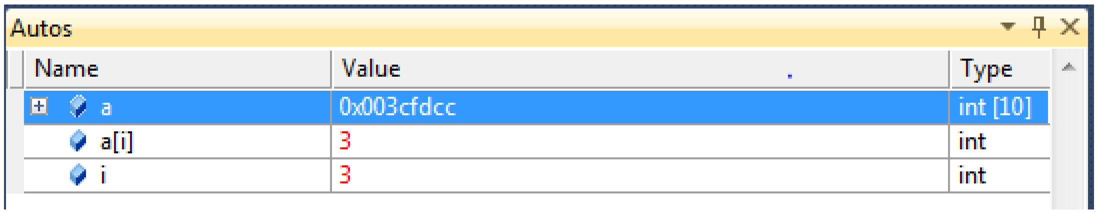
Set breakpoints as shown in Figure 16, then **Start Debugging** to run to the first breakpoint—the start of the for loop. Note that the Autos window now shows two variables: the array a and the loop index i.

The "+" to the left of the array name can be used to expand the array and display all array element values, as shown in Figure 17. The figure shows the values of the array during the fourth loop iteration (i = 3), after executing the statement a[i] = i.

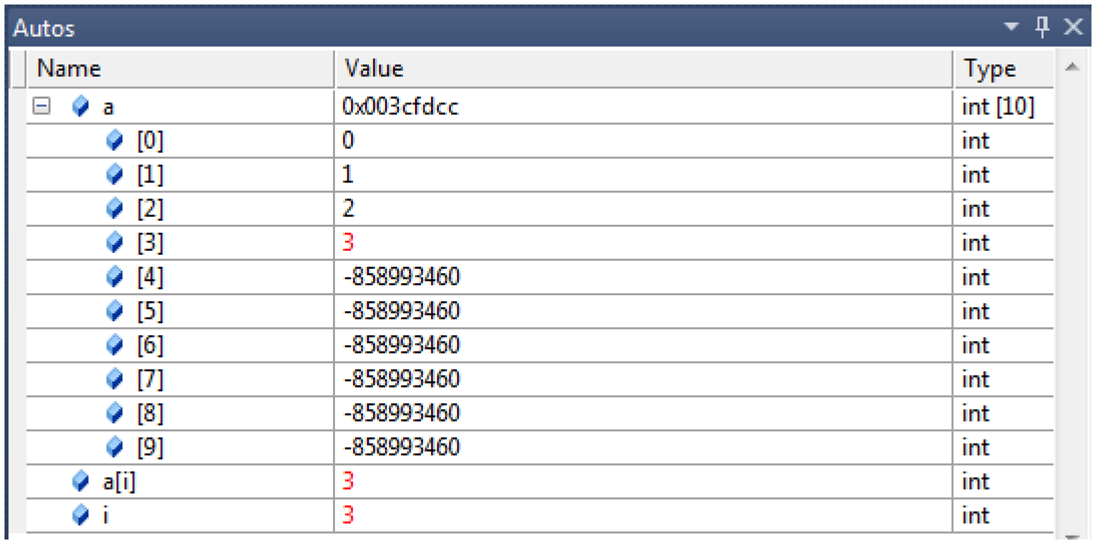


```
int main() {
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = i;
        cout << a[i] << endl;
    }
    return 0;
}
```

Figure 16: Second test program with appropriate breakpoints



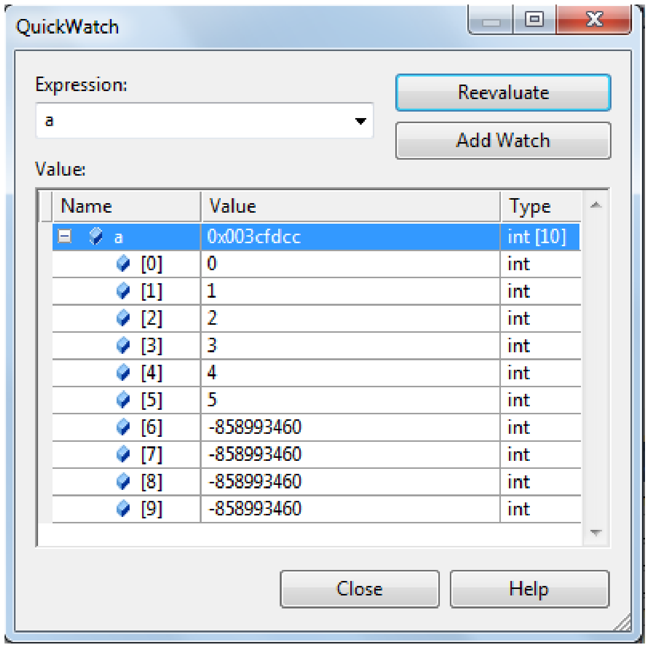
Name	Value	Type
a	0x003cfdcc	int [10]
a[i]	3	int
i	3	int



Name	Value	Type
a	0x003cfdcc	int [10]
[0]	0	int
[1]	1	int
[2]	2	int
[3]	3	int
[4]	-858993460	int
[5]	-858993460	int
[6]	-858993460	int
[7]	-858993460	int
[8]	-858993460	int
[9]	-858993460	int
a[i]	3	int
i	3	int

Figure 17: Autos window shown before and after clicking the "+" symbol to show all elements of array a, as well as the current value of i and a[i]. These snapshots were taken during the fourth loop iteration, after a[3] was set to 3

Note that watches can be set for arrays as well, using the exact same methods described above. Figure 18 shows the QuickWatch window during the sixth loop iteration (i=5).



Expression: a

Value:

Name	Value	Type
a	0x003cfdcc	int [10]
[0]	0	int
[1]	1	int
[2]	2	int
[3]	3	int
[4]	4	int
[5]	5	int
[6]	-858993460	int
[7]	-858993460	int
[8]	-858993460	int
[9]	-858993460	int

Figure 18: QuickWatch window showing array a during the sixth loop iteration

7. FAQ (to be added)

Note: Additional sections may also be added to this document at a later date.

Appendix: Useful Keyboard Shortcuts

Shortcut	Command	Description
F7	Build Solution	Compile and link all project files
Ctrl+F5	Start Without Debugging	Basic program execution—program will run to completion (assuming no errors), ignoring all breakpoints
Debugging Commands		
F5	Start Debugging	Start program execution; run until next breakpoint
	Continue	Run until the next breakpoint (if program already running)
F9	Toggle Breakpoint	Set/remove a breakpoint at the current line of code
F10	Step Over	Execute the current line of code without entering any functions called within that line
F11	Step Into	Execute the current line of code; enter any functions called within that line to allow the user to step through that code
Shift+F11	Step Out	Exit the current function and return to the line of code in which the function was called