Introduction to Programming in C for EGR140 Computer Utilization in Engineering

Most of the work done in the earlier parts of the course use MATLAB (Matrix Laboratory) as an environment within which one can do a variety of things ranging from calculations similar to a calculator, to writing programs, to executing library packages to do elaborate image processing, modeling, or simulations. The prerequisite for doing all this is having a PC running MATLAB. Buying a PC is no longer the big financial challenge it once was, but MATLAB is expensive. (As a student you can get a good deal, but licenses once you graduate are expensive, perhaps even more than the computer.)

An alternative to doing engineering analysis in MATLAB is to do so using a general purpose programming language. Often compilers are either free or are much less expensive than MATLAB. Furthermore, a general purpose programming language can do things that are beyond what MATLAB is intended to do, such as text handling. (You can probably do text handling in MATLAB but it would be a bit unnatural, since the moist basic form of data is the real number.)

To explore the use of a general purpose programming language, we will start with the language "C". You perhaps do not hear C mentioned very often, since programmers more commonly use the elaboration of C called "C++". However, C is often the better choice for problems of mathematical analysis. It is simpler and easier to understand. C++ is the better choice for applications featuring mouse clicks, icons, buttons, and other fancy user interface features.

We will use the "Integrated Development Environment" (IDE) called "Microsoft Visual Studio 2008". This program is not free, but similar products can be found that are free. Visual Studio 2008 is available for free to students. If you are programming a Mac, the IDE "XCode" for that platform is free. If you are programming microcontrollers, the Codewarrior for Microcontrollers IDE is available for free for configurations up to a certain size.

What an IDE is and what it does is perhaps best illustrated by example. Follow the directions below:

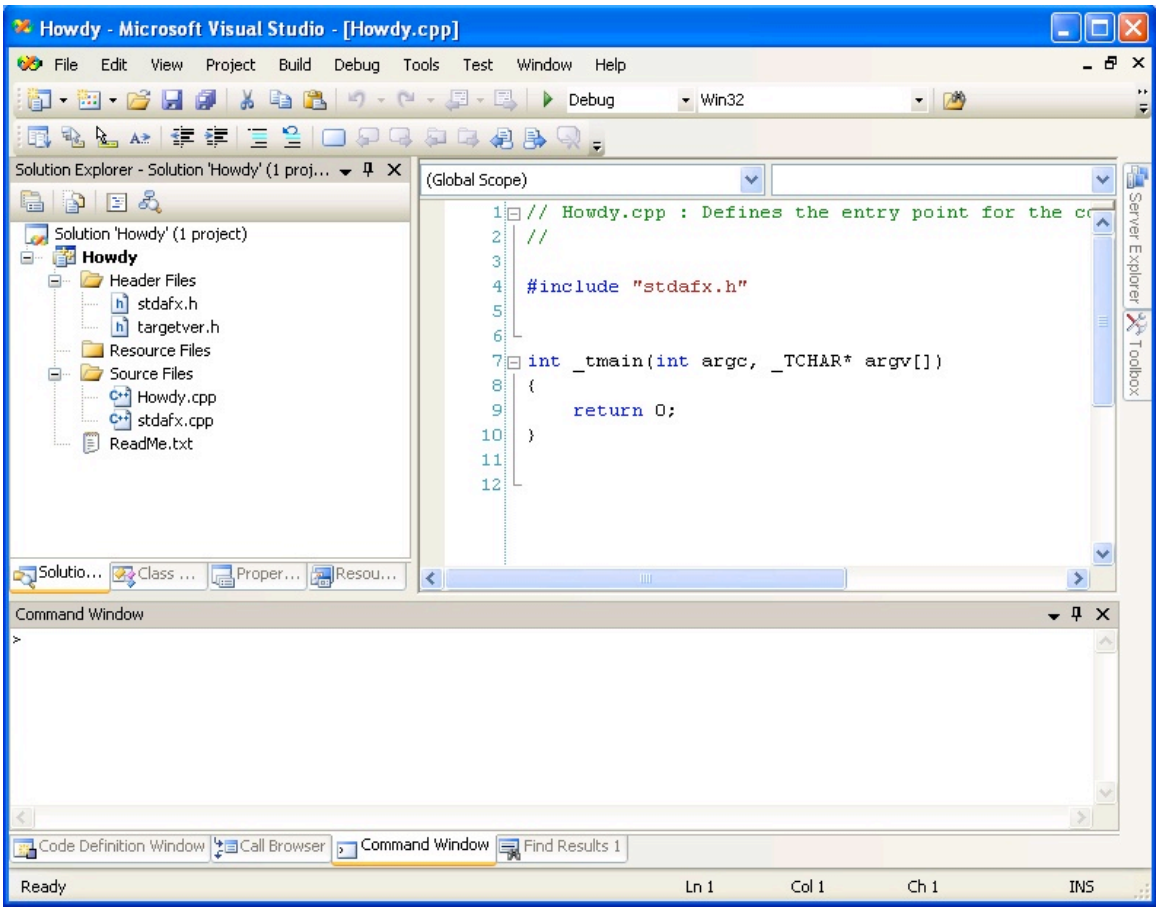
1. Log onto a Windows (XP in the SLC216 lab) computer, and start up Microsoft Visual Studio 2008. When it comes up, there are several daughter windows displaying various things including news, getting started tutorials, and stuff like that. Ignore all that. You can close the news (Developer Center) and related clutter window by clicking the small black on grey "X" in the upper right corner (not the red one.

2. On the File menu, choose "New" and then "Project". When you release, a series of windows appear which guide you through creating the new project.

3. The first window allows us to choose what type of project. We want the simplest possible thing we can do to start. Under the "Project Types" list select "Visual C++" and under that Win32. (We are going to be content to do a 32 bit address word

length project.) Now, another choice option flashes up: what kind of Win32 project? There are a couple of "Templates" we can choose from. Again going for simplicity, we choose "Win32 Console Application". That means all of our interactions with the application will take place in text in a dedicated "console" window. We also need to name the application (Type in "Howdy") and tell Visual Studio where you want it. At this point you may want to use your H drive or insert your thumb drive and use the browser to point to it. (The default is C:\Documents and Settings\(?Your user name?)\My Documents\Visual Studio 2008\Projects. That way it stays on the computer and you can't carry it with you when you leave, unless you copy the folder to your drive.) Leave checked the box to create a directory (named Howdy) for the "Solution." (A "solution" is a package of related applications. We will have only the one application "Howdy" in our solution.) Now click "OK".

4. Now you get a window "Welcome to the Win32 Application Wizard." Just hit "Finnish". Visual studio now creates an "empty" project that does nothing. When it finishes, what you see is"



a. In the largest window, a file named "Howdy.cpp" that contains the "source code", the executable statements that do what the programmer wants to have happen. More about that soon.

b. On the left, the "Solution Explorer" window that is a diagram of all the various stuff that it takes to make this code run on a Windows machine. That includes two "Source" files, Howdy.cpp and stdafx.cpp, and two "header" files, stdafx and targetver.h.

c. There are lots of other buttons and icons and a command window at the bottom, but we'll talk about those when we need to.

5. Now, we will create an "empty" project that does nothing. We want to compile and run this project as it is right now. Under the "Build" menu, select and release on "Build Solution". You will see an "Output" window appear that will describe the steps taken to "build" the application. At the end you should see " `Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped`". That means it got built.

6. Now let's run it. Find where the "Howdy" folder is (where you said to put it) and inside, look inside the "Debug" folder for "Howdy.exe". That is your application. Double click on it to run it. What you will see is a window very briefly flash open and then disappear. What has happened is the application does nothing except close itself. It is, after all, empty. We have not yet added instructions to do anything. (You could also run it in the "debugger" by clicking on the green arrow on the commkand bar.)

7. So, now go back to your Visual Studio main window. We will now modify the main application source file Howdy.cpp as follows iding the IDE editor.:

a. Add two new lines before the "return 0:" statement:

printf("Howdy!\n");

while(getchar()==0);

After doing that, the text of Howdy.cpp should look like this:

```
// Howdy.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
	printf("Howdy!\n");
	while(getchar()==0);
	return 0;
}
```

b. Now "make" your application again using the "Build" / "Build Solution" menu choice.

8. Run "Howdy.exe" by double clicking on it or clicking the green "debug" icon. Here's what you should see:

9. Quit by hitting return. The application will instantly exit.

So, what have you done? You have created an application that interacts with the user using text (a console window) that simply types out a message, then waits for the user to enter anything and hit return. Let's see how it works. There is a "main program" that can accept some incoming stuff that we will ignore. The statement that defines the beginning of our main program is:

```
int __tmain(int argc, _TCHAR* argv[])
```

The main program is actually just another function, "_tmain" (terminal version of main). Data that can be passed into it are an integer named "argc" and an array of character srtings (called T_CHAR's) named argv. We don't have to worry about any of that because we don't use any of it. Our main program, though, does have to return an integer (int) when it finishes. When we had the "empty" program, the only executable statement inside main (That is, in between the two braces "{" and "}" was "return 0;". What that does is exit and return (to whoever called main) the value 0. (Returning out of main with a value other than 0 means that an error occurred and the program is exiting abnormally.)

Our modification to main included first a statement to print a message to the screen. "printf" is a function defined in the standard input/output library for printing to the console. Inside the parentheses we put a "string" of characters to be printed. The "\n" means to do a "new line" operation. Any text that follows will be on a new line. So, as a result of this statement the message "Howdy!" appears in our console window.

The other statement we added, "while(getchar());" uses a control structure called a "while" loop. That is, we wait until whatever is inside the parentheses is "true" (which in C means nonzero). Inside the while we call the function "getchar (passing no information into it). The function getchar() obtains a character of input

from the screen. Anything that comes back other than a zero causes us to get out of the "while" and move on to the next statement, the return.

In summary, this application opens up a console window, it prints "Howdy!" and then waits for the user to hit return, at which time it quits.

Now, let's modify the program to do something more interesting. We'll keep the "Howdy" and the exit, but in between we'll have it calculate a dot product of two vectors. We will go back in and edit the main function as shown below:

```
// Howdy.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

float A[]={1,2,3};
float B[]={4,5,6};
int _tmain(int argc, _TCHAR* argv[])
{
	int i;
	float dot=0.;

	printf("Howdy!\n");

	for(i=0;i<3;i++){
		dot=dot+A[i]*B[i];}

    printf("The dot product is %f\n",dot);

	while(getchar()==0);
	return 0;
}
```

The statement "float A[]={1,2,3};" causes there to be a global variable named "A" that is a vector (that's what the [] means) of three floating point (32 bit representation of real numbers) values, and the vector initially has the value [1,2,3]. (Even though you don't see decimal points, the statement says they are floats, so floats they will be.) There's a similar statement for the vector B. The elements of vector A are A[0] (which has the value 1), A[1], and A[2]. Notice that the indexing starts with 0, not 1. That's a difference between C and many other languages, including MATLAB and FORTRAN, which start indexing vectors with 1, not 0.

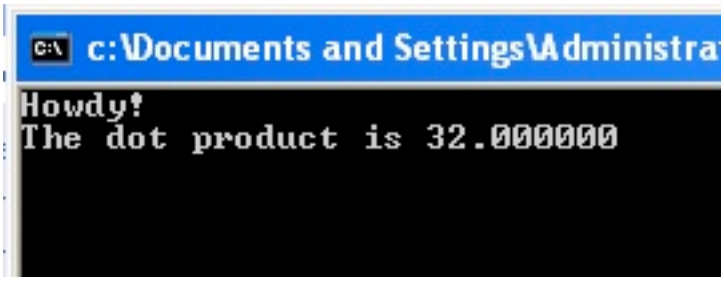
Inside main we need an integer to use to count which element of the vector we are working on. That can to be an integer variable rather than a float. (Under Windows integers are by default 32 bits; if you want a 16 bit integer ask for a "short". On some machines an intis only 16 bits. So if it matters, it's safer to ask for a "long" if you really want values up to 2^31. Use "int" if it doesn't matter, or to be compatible with library functions.) We also need a float in order to accumulate our dot product. Notice that by putting "float dot=0." inside main it is "local" and if we had other functions they could see and use A and B (which are global variables) but not "dot" which is local to main. We are not getting that fancy (yet).

The "for" statement says start by making i equal to 0. Then check to see if i is less than 3. If it isn't (that is, if it is 3 or more) then exit. Otherwise, do the stuff within the braces. Once the stuff is in the braces is done, add one to i. In C, "++" means add one. Then see if it is less than three. Keep going until you exit.

So, the stuff inside the "for loop" gets executed three times, once for i=0, then for i=1, then for i=2. The first pass A[0] is multiplied by B[0] and the product added to "dot" (which initially had the value 0). Then, A[1] is multiplied by B[1] and that product added to dot. Finally, the product of the third elements of the two vectors is added to dot. The result is the dot product. It should be 1*4+2*5+3*6 = 32.

The following print not only prints text but the "%f" means to print out a floating point number, "dot" to be specific. So, this statement should print our answer to the console.

So, now "make" the application and execute it. You should get:



Yes, 32 is the right answer.

Now, there are lots of things that could be done to improve the program, but hopefully this example gets across several important points:

1. You can take this "Howdy.exe" file and put it on any PC and run it. You don't need Visual Studio 2008 to do so. That was just a development tool.

2. A computer does one simple thing at a time. We didn't simultaneously multiply all the elements of A to those of B. We had to do them one at a time. MATLAB has to do the same thing, but it gives the user the illusion that it is all happening simultaneously and instantly, and that the computer is really operating on vectors, not just individual numbers. (Modern higher end computers do have some organic vector processing capability, but we are not going to get into all that. It gets used mostly forr graphics and such.)

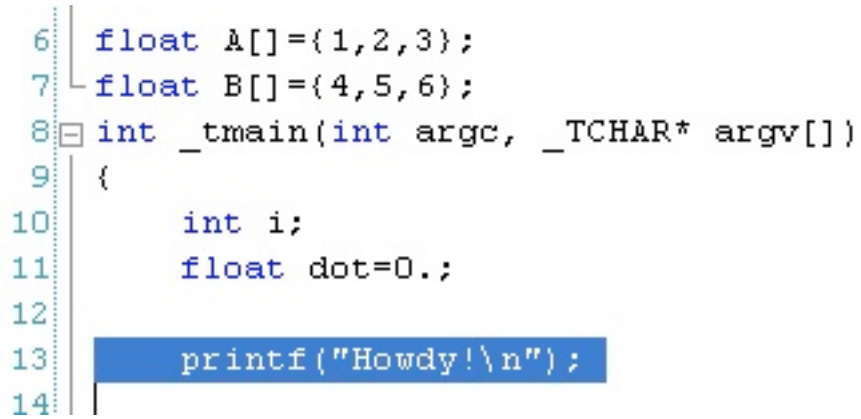
3. Even though we were limiting ourselves to "C" statements, the compiler (and IDE) are actually C++ tools. We could have used C++ language constructs if we had wanted to. To do a console application under Windows there is a lot of C++ stuff under the hood (attached through "stdafx") that we don't want to have to look at. If you want to do something more complicated than a console application, something with pointing and clicking, you have to dive into C++ under Windows. That can get involved.

OK, so it's possible to write a C program (using the C++ facilities of the Visual Studio IDE) and make it work. How does it work? There are a couple of ways this question could be answered. One answer is, "I don't care. Just so it works and gives me the answer I want." That is like being content to flip on a light switch and not caring how electricity works, or where it comes from. Or owning an automobile and not caring what the "motor" is or how it works. It is possible to go through life not understanding or caring about some of the things that you interact with routinely. Most engineers, though, are more curious. Maybe there isn't enough time and effort to understand, for example, how cellular biology works in detail or the nuclear reactions inside the sun. We take advantage of both routinely without a detailed understanding. But for an engineer, the computer is both an essential tool and a component of many systems. So, I'm going to assume interest.

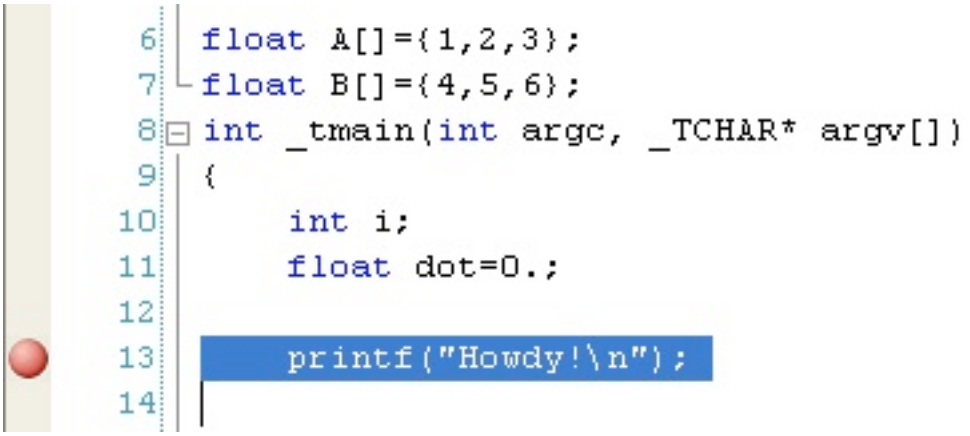
Within Visual Studio 2008, we have an application called a "debugger" that has been integrated in with the other applications in the package, to help with understanding what is going on "under the hood". Just as someone seeking to understand what is going on inside an automobile engine might use a timing light, a compression gauge, or a dynamometer, the debugger provides a window into the internals of a computer program, in our case the application "Howdy". So, what follows is a description of how we can use the debugger to see what is inside the Application and how it works.

If we just click the green button to debug (or select "Start Debugging" on the Debug menu) we start the program running, and it completes everything prior to getting the signal to exit. We want to go more slowly. So, we start out by setting a "breakpoint". A computer runs by executing "instructions" statements one after another. Each "statement" in our source code (a line ending with ";") has been converted by the compiler (another component of the IDE) into several instructions that the machine understands. What we want to do is stop the program in the middle of execution so we can look around and see what is going on. A "breakpoint" is how we tell it where to stop.

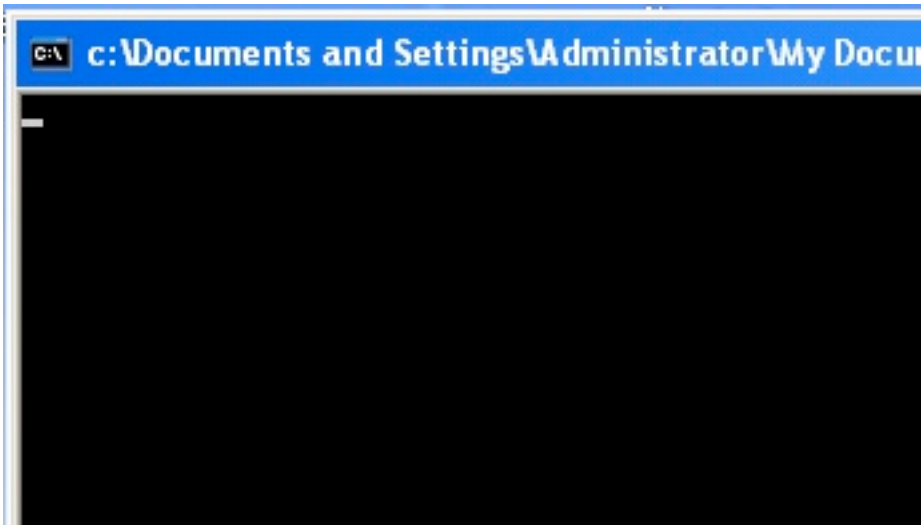
With the source file "Howdy.cpp" still open, click to the left of the statement "printf("Howdy!\n");". The statement will be highlighted (selected).



Now click in the grey left border of the window. A red spot will appear to indicate a breakpoint at this statement.



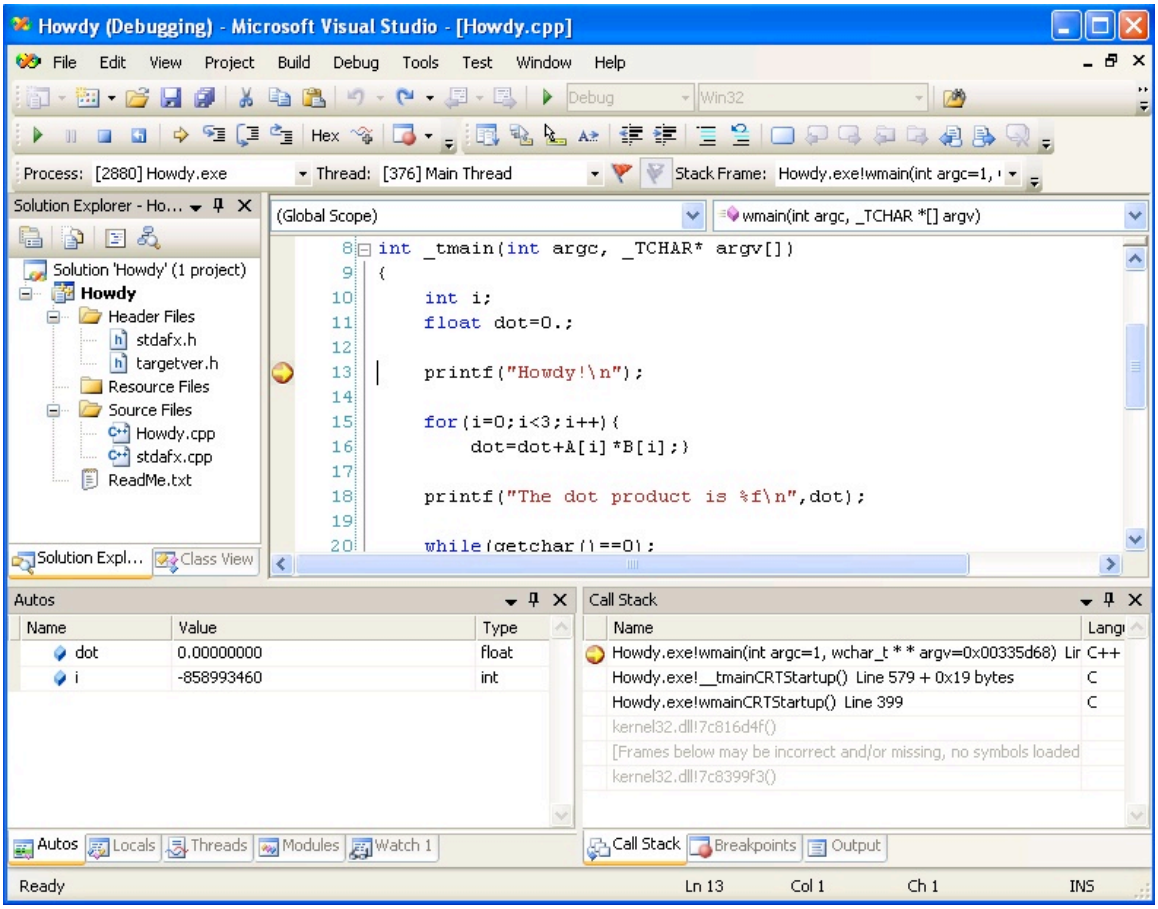
Now, start debugging by clicking the green arrow or pulling down and releasing "Start Debugging". Now the program stops just after the console window comes up. (There is a flashing cursor; that's the operating system waiting for input.):



Back in the Visual Studio window, the program is stopped at the printf statement, as seen by the yellow arrow at the breakpoint. In a couple of other windows at the bottom, we can see (on the left) the variables "dot" and "i". On the right we can see the "call stack" that tells us what function we are in. We are in Howdy.exe!wmain(.), the main program. There are a couple of functions higher up which are Windows C++ code that get our console function started. We aren't going to go there.

The variables dot and i are of interest. "dot" has the value 0.000. The debugger knows it is of type "float" so it is showing the floating point value. If you look at the source code, dot was initialized with the value zero given, which is what we are seeing here. The integer "i" is a different matter. Whan it was called into being by "int i;", no value was assigned. So the value of i is "uninitialized." There is a

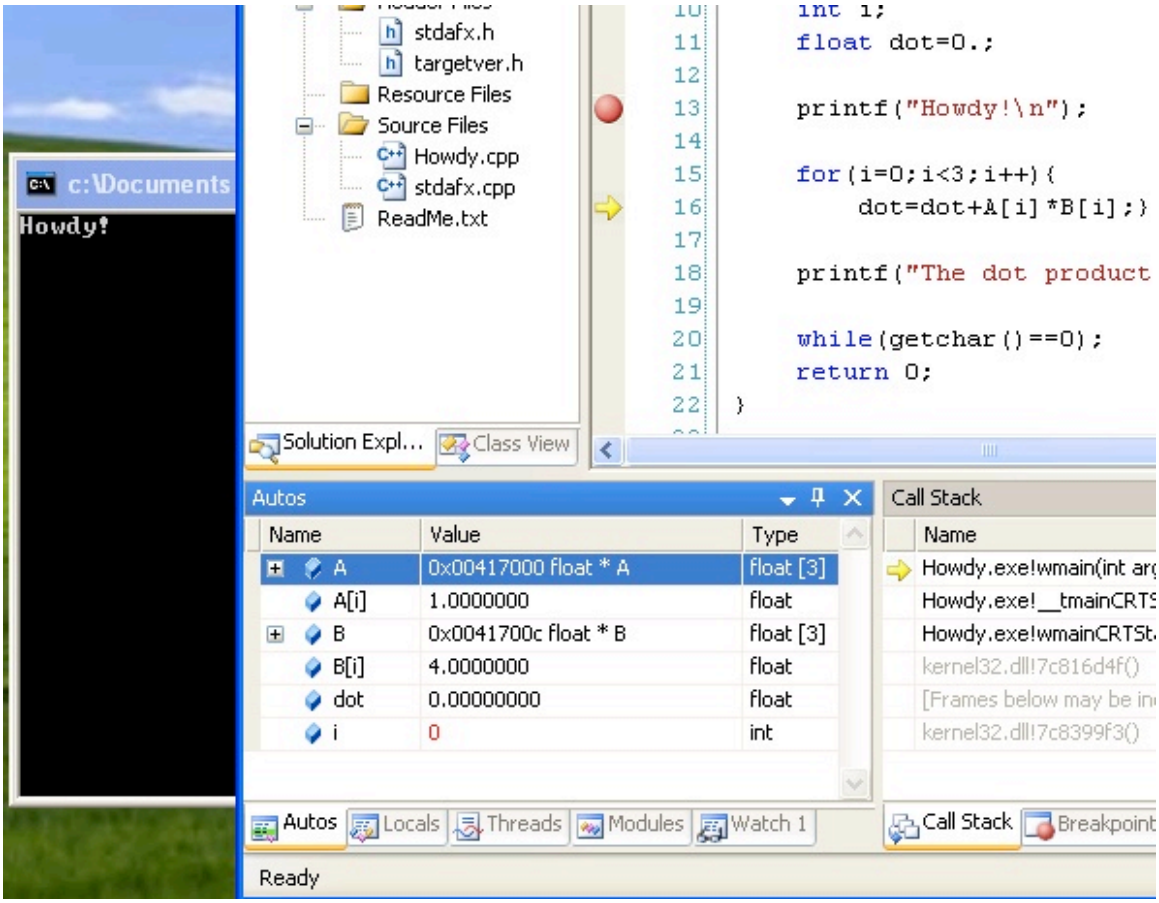
particular spot in the memory of the computer where i exists, but it now contains random garbage, the bit pattern that happened to be there before i was created. In my run the value is -858993460. But on any given occasion it might be different. What we can't see is exactly where in memory dot and i are stored (their adresses); the debugger doesn't tell us that (and apparently won't; some other debuggers will).



Now, what we want to do is watch our program execute one little step at a time. Got to the "Debug" nenu and pull down the menu, releasing on "step over". What will happen is that the statement we are stopped on will be executed. If we had chosen "Step into" instead, we would have plunged into the dark abyss of how "printf" works. We don't want that. If we had chosen "Step out" we would have completed the function we are in (the main program) before stopping again, and that's more than we want to do right now. After doing this step we are stopped on the "for" statement, and "Howdy!" has appeared on the console window. Our program has done that one step. But it has not started the "for" statement, so i remains ininitialized.

Step over once more. (Notice there are three buttons on the tool bar for "step into", "step over", and "step out" that you can use instead of the menu.). Notice the changes in the list of variables! Now "i" has the value 0. Notice that the zero is red, to indicate that the value just changed. Notice also that the variables A, A[i], B,

and B[i] have been added, since now we are at a statement where these matter. Furthermore, we can see their values.



Since i is zero, then A[i] is A[0] and has the value 1.0. Similarly B[i] is 4.0. But what is A? In the particular execution I'm debugging as I write this, A has the value 0x00417000. It is a float *. That means it is a pointer to a float (a floating point variable). So, in C the variable A is the address in memory where the elements of the vector A are stored. We are seeing a detail normally hidden from view. In MATLAB we never can see where anything is actually stored in the computer. Here we can. We can also see the address of the elements of B. They are at 0x0041700c.

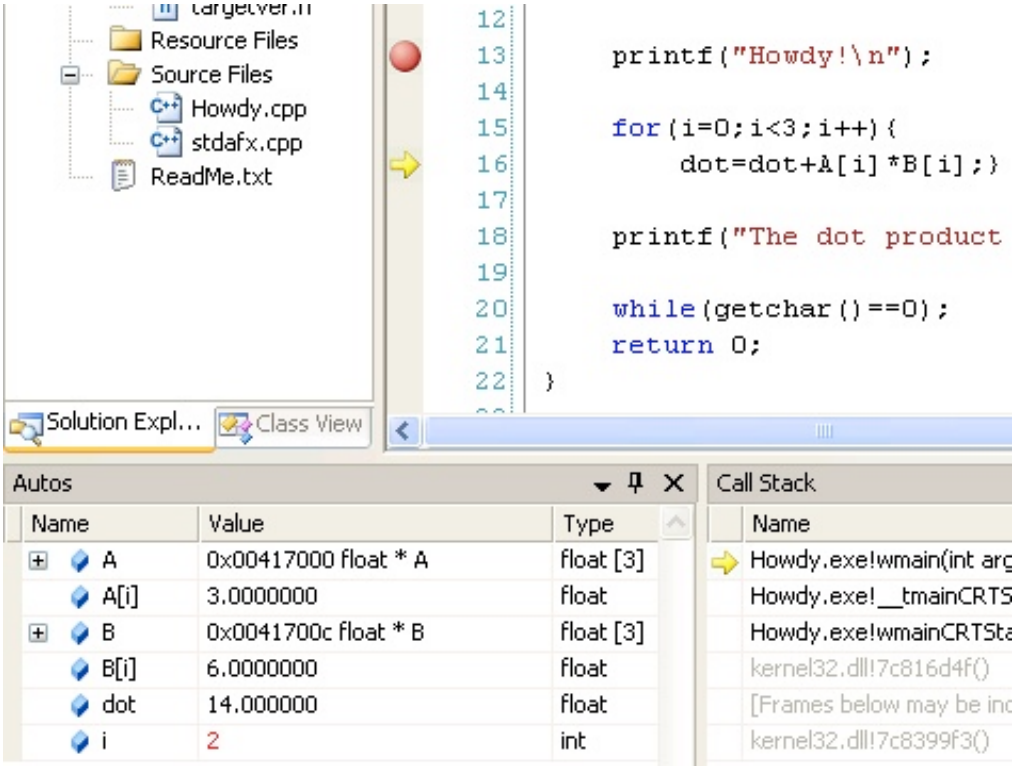
Here, we need to take a break and explain this number, this address in memory, that contains the letter "c". The prefix "0x" means that the value is givenb in hexadecimal, that is, base 16, instead of your familiar base 10. That allows us to represent very big binary numbers compactly. For each group of 4 bits in a binary number, we have 16 combinations, 0,1,2, ... 9, a, b, c, d, e, f. We use the characters a to f for vlaues 10 to 15 for which decimal does not have any digits. So, the address of the vector B is actually (in binary):

0x0041700c = 0000 0000 0100 0001 0111 0000 0000 1100 (binary)

Notice that even though we are executing a "Win32" program, the addresses in the macine are 64 bits. In old computers from the 1970's and earlier that had "front panels" we could actually peek inside the memory at any given address and see what is there. Those hardware debugging techniques have been repl;aced by softweare debuggers like this one, which is why computers no longer have a "front panel" with lots of little lights and switches. The front panel was expensive and software is cheap to produce in quantity.

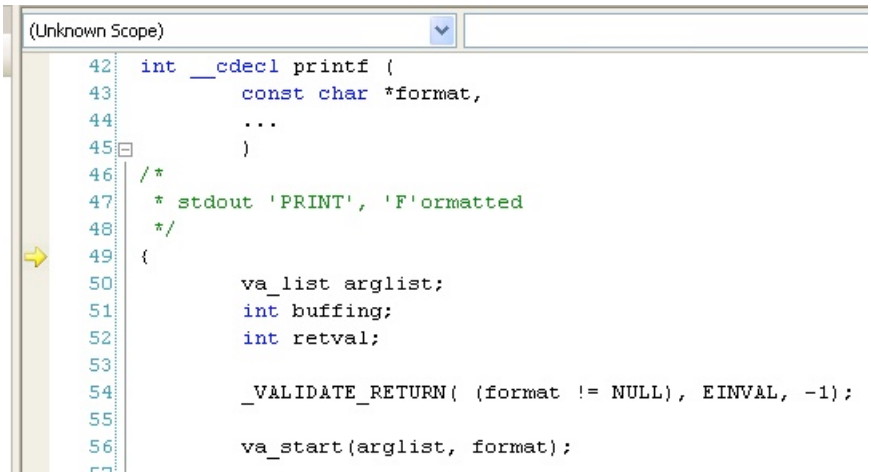
If you take the difference between the address of B and the address of A, the difference is c, or 12 decimal. That's because each "float" takes up 4 bytes of 8 bits each, or 32 bits. So, the difference isd the 12 bytes needed to store the three values of the vector A. The first element of A (pointed to by the variable A), A[0], is at 0x00417000, the second A[1] is at 0x00417004, and A[2] is at 0x00417008. The three elements of B would be at 0x0041700c, 0x00417010, and 0x00417014. We don't really need to know all that to follow the action, but there are times when we need to follow pointers to understand more complicated programs.

So, now let's take one more step. We go back up to the for. That's because we are in a loop. The statement within the braces is executed for each value of i. We have not quite finished the previous loop because i is still zero. Take another step. Now we are back down to the dot calculation again. Notice dot has the value 4.0 as expected after executing this statement the last time. Now i has been changed to 1. We can see the values of A[1] and B[1]. Click two more steps and we see that dot is 14 and we are about to add A[2]*B[2} to it.

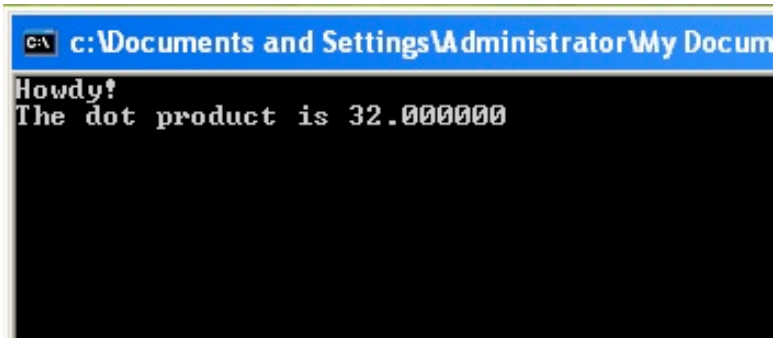


One more click to step and we are up at the "for" again, then another click and we fall out of the loop. The variab;le i has been incremented to 3 and so it fails the test "i<3". We are at the printf that will send our answer to the screen.

OK, let's be bold. This time we will "step int" the function. printf is a library function that has been "linked" into our application when it was built. That was done by the "linker", another application of the IDE. So, when we step inside printf, we find ourselves at the beginning of some unfamiliar code that we didn't write.



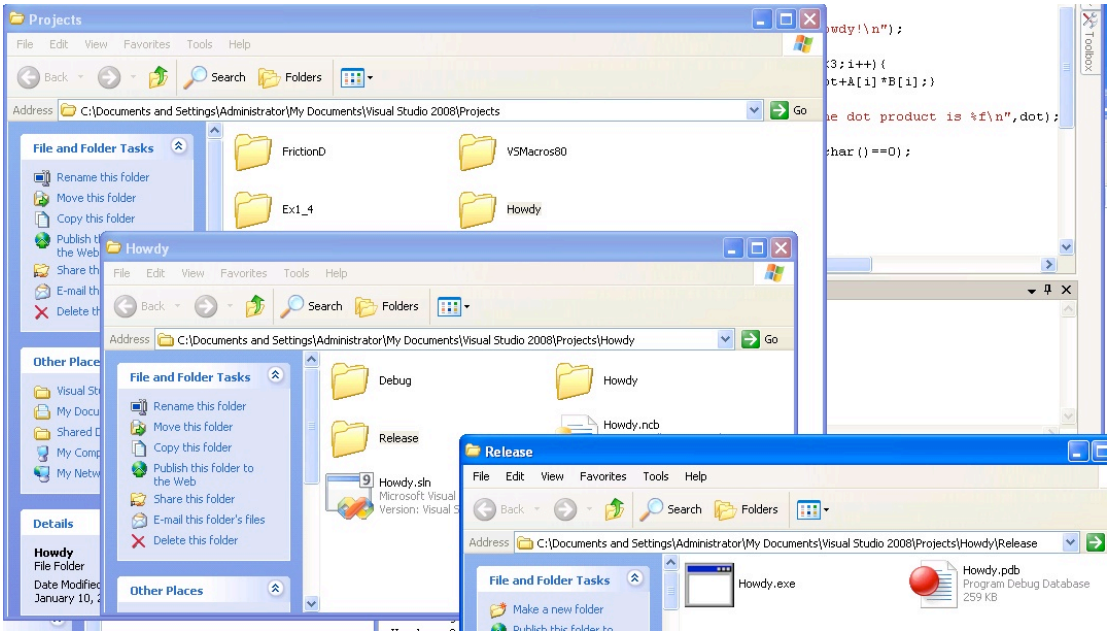
Ugh! Let's make an escape. We will "step out." Normally we won't want to do any debugging inside library functions. (But, if we ever wanted to know how they worked in detail, that's how we could find out.) We will "step out." The printf has been executed.



At this point we are done with what we wanted to see, and will just let the program "go". Either click the green arrow or relares on the Debug / Continue menu item. Since there are no more breakpoints that we will encounter, execution continues normally until we exit.

Now, having satisfied ourselves that it works OK, we can tewll the IDE, Visual Studio 2008, that we are done with debugging. The next time we create an executable application (Howdy.exe) leave out support for debugging. If you look at the size of the debug version of the executable file, it's 28.0 KB. That's big for a program that does so little.

Back in Visual Studio, look for a text block just to the right of the green debug (go) arrow that says "debug". This is where you designate the nature of the target application. Change it to "Release". This is the version that you will emailt to all your friends for their entertainment. Now, build the solution again. Within the Howdy folder there is now a "release" subfolder, and in it is your program, now iun a more compact release of only 7.5 KB size. This is what you copy and send out. (Just to experience it, double click on the Howdy.exe in the release folder. Yep, it works. That's what all your friends will see when they run this delightful creation of yours on a PC. (Don't bother to send it to Mac users; it won't run on a Mac unless you boot the thing in Windows or run Windows as a virtual machine, like I'm doing right now.)



So, there it is. With a typical programming language, you can create applications that can do some analytical work. Furthermore, you can create copies of the program that can run on any PC. Within some development tools (like XCode for the Mac) you can create versions for different computers, one release version for unix systems, another for Macs, and another for PC's. All of these run without having to have Visual Studio (or another development IDE) open. (Debugging, on the other hand, requires you to be in the IDE.)

Hopefully this gives you a bit of a sense of what computer software is and how it runs. We have not yet gotten into files and reading input and outputting stuff that we could look at with, say a text editor or plot with another program like Excel. We'll get to that later. What will have to remain a mystery for now is how the statements get converted into machine instructions, and how those execute on digital hardware. We just can't get further intop that here, but in EGR222 Mechatronics we will dive deeper to the machine language level. How the computer hardware works is a matter of Digital Design and Computer Organization, farther than we can go without dedicated courses.

Taking the next step: Let's write a C program that will actually perform some useful analysis. We will keep the data small just to make it easier to handle for this exercise. Here's the scenario: There is a Tribometer (friction measuring machine) that records force and torque at regular intervals while a sample disk of plastic material is being rotated against a piece of brass metal. We are interested in how the average force and torque vary over time. We are also interested in how they may correlate over time: does one lag or lead the other? What we want to do is take 11 point moving averages for Force, Torque, and the correlation coeficient. We will get our data from a text file, and put our output to a text file. If we do this right, we can import the output file into Excel and use it to graph the results.

The input file is shown below (shown in two columns). There are 60 samples. On each line (after the "header" information) are data for sample number, force, and torque. (The force and torque numbers are actually Voltages for the instruments recorded; conversion to appropriate units is a later step in the analysis.)

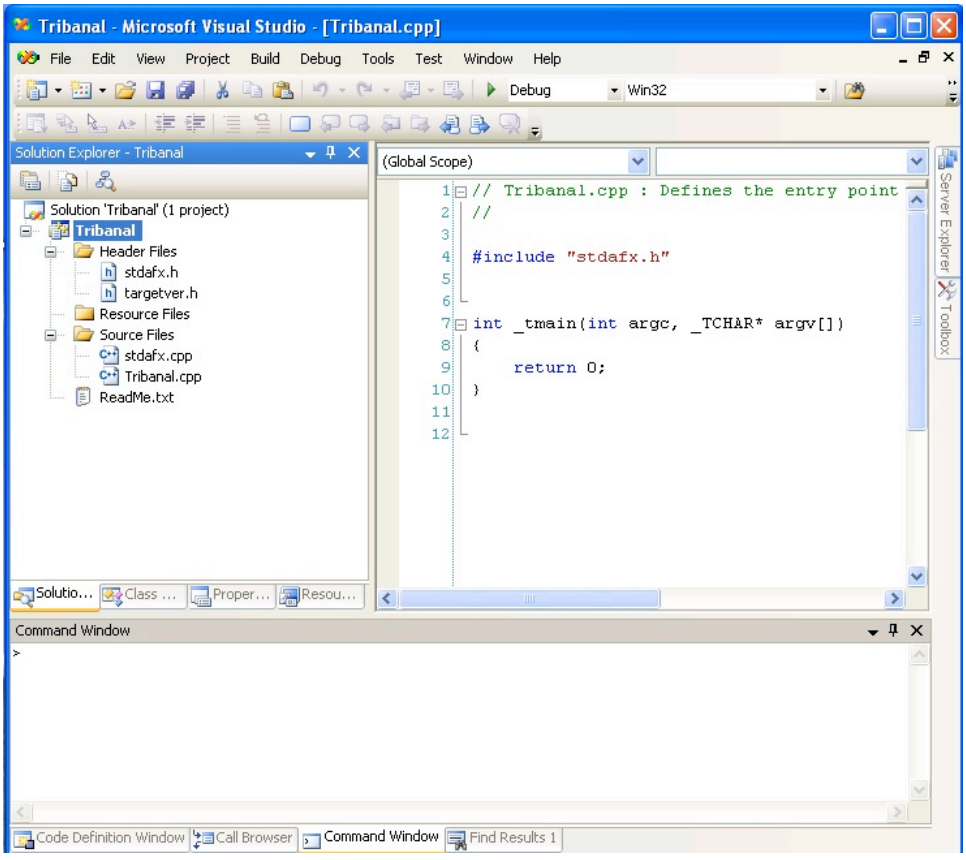
```
Tribometer data  60 Samples
Data out: Samplenumber  Force
       Torque
1     3.364 1.19
2     3.369 1.199
3     3.374 1.191
4     3.373 1.198
5     3.369 1.19
6     3.381 1.2
7     3.376 1.201
8     3.364 1.189
9     3.364 1.19
10    3.357 1.19
11    3.367 1.189
12    3.362 1.19
13    3.364 1.194
14    3.369 1.194
15    3.364 1.19
16    3.364 1.194
17    3.358 1.187
18    3.372 1.19
19    3.364 1.187
20    3.357 1.187
21    3.364 1.19
22    3.352 1.182
23    3.376 1.196
24    3.367 1.196
25    3.353 1.177
26    3.359 1.182
27    3.354 1.185
28    3.359 1.189
29    3.357 1.185
30    3.362 1.189
31    3.364 1.19
32    3.364 1.184
33    3.362 1.184
34    3.352 1.179
35    3.364 1.19
36    3.367 1.187
37    3.364 1.185
38    3.35  1.176
39    3.359 1.185
40    3.35  1.195
41    3.352 1.185
42    3.347 1.188
43    3.333 1.174
44    3.337 1.188
45    3.339 1.189
46    3.337 1.184
47    3.334 1.187
48    3.342 1.185
49    3.345 1.19
50    3.33  1.182
51    3.34  1.199
52    3.335 1.19
53    3.333 1.199
54    3.333 1.19
55    3.318 1.177
56    3.335 1.194
57    3.318 1.189
58    3.345 1.213
59    3.328 1.213
60    3.328 1.228
```

What we want from our program is the average, at sample 6, of samples 1 to 11 for both force and torque. Then for sample 7 we will want the average of samples 2 to 12 for force and torque. And, so on, until we get to sample 55. (We don't do this for samples 1-5 or 56-60 becuase we do not have a full 11 sample set from 5 before to 5 after.) We will also want correlation coeficients, but we'll do that as a second step.

So, we know what our input looks like, and what we want to do with it. Next we need to know what our output will look like. We will want to make it suitable for loading into Excel, so we want a "tab delimited" or a "comma delimited" file. I'm going to choose comma delimited. You can see a comma; you can't see a tab. That will make things easier to manage. So, let's keep it simple. Let the file look like this, where xxx, yyy, and zzz are our outputs. But we will initially skip the Correlation values zzz.

```
Tribometer analysis
Sample, Force, Torque, Correlation
6, xxx, yyy, zzz
7, xxx, yyy, zzz
........
```

We are now ready to create a C program to do this. No sense in getting fancy; we'll make it a console application. Proceeding as we did earlier for Howdy, we create an empty application named "Tribanal". It looks just like Howdy did at first.



We are going to need to read and write "files". That's already covered by the standard input and output libraries. So, what we want to do first in the code is provide the variables for those files by adding (before main) global FILE variables:

FILE *inFile, *outFile;

The two variables are pointers to the files. Whenever we do something with a file, we need to use this pointer to indicate which file we are using. The first thing we will do in main is open these files and make sure we are successful doing so. (If either file fails to open, we will quit.) Here's how we do it:

```
inFile=fopen("Tribometer sample data.txt", "r");
outFile=fopen("Tribanalout.txt", "w");
if(inFile==0 || outFile==0) {
        printf("File failed to open\n");
        while(getchar==0);
        return 1;}
```

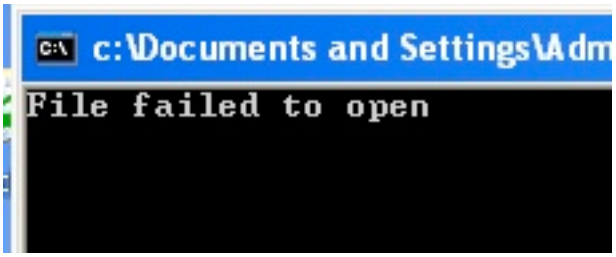
The fopen function opens the file named, either for read purposes "r" or write purposes "w". The overall Tribanal.cpp file now looks like this:

```
// Tribanal.cpp : Defines the entry point for the console
application.
//

#include "stdafx.h"

FILE *inFile, *outFile;

int _tmain(int argc, _TCHAR* argv[])
{
    inFile=fopen("Tribometer sample data.txt","r");
      outFile=fopen("Tribanalout.txt","w");
      if(inFile==0 || outFile==0){
            printf("File failed to open\n");
            while (getchar()==0);
            return 1;}
      while (getchar()==0);
      return 0;
}
```

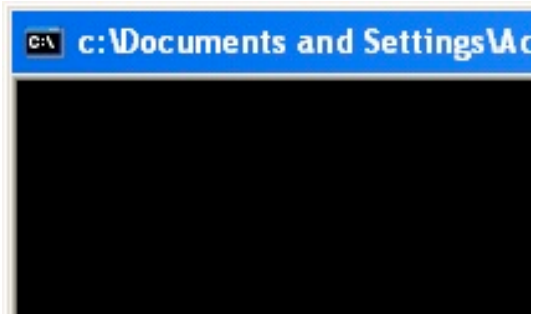
Notice the use of "==" to test whether something is zero. A zero pointer means it's invalid. We use "==" for "is equal to" in logical expressions used to test whether something is true or not, such as an "if" statement in this case. If what we test is true, in this case if either pointer is zero, we do the stuff inside the braces. The symbol "||" means a logical "or" operation. If either "==" test returns true, then we do the exit. Otherwise, we skip the stuff controlled by the "if" statement. Notice that we use "==" inside the while loop too. If you use "=" instead of "==" the program will try to take the value on the right and put it into whatever is on the left. In this case, it would set both

pointers to zero, regardless of what they were before. This is one of the most common editing errors in C and C++.

Let's see if it works. We hit the debug button and get:



This is what we expect! It did not find our input file. Put the file "Tribometer sample data.txt" file into the Tribanal folder. (Notice that the file "Tribanalout.txt" exists. We can double click on it to open it in Notepad and see the contents. But it is empty because we have not written anything to it yet.) We now try "debug" again. This time no message. Oh, we didn't have it write anything out if we were successful opening the files. So, that's what it is supposed to do. Boith files opened OK.



Next we will add a print statement to say that it's working OK. We also want to write the header information out to the output file. To do that we can't use "printf"; that function writes to the console window. Instead, we use "fprintf". It does the same thing, but writes out to a file. We have to use the pointer to the output file to tell it where to write to. While we are at it, we will "comment" our source code by adding a note to describe what is going on. Comments are ignored by the compiler. Use "/*" to start a comment and "*/" to end the comment. That's the C form of comments. For C++ you can alternatively add "\\" at the point in a line beyond which everything is a comment. Notice that the "empty" project included a comment on the first line. Since we are using the C++ compiler, we can do either. As far as Visual Studio 2008 knows, we are writing C++. We just happen to be limiting ourselves to the C subset of C++ for now. After doing this, try it out. If you look in the Tribanalout.txt file, you should now see the headers.

```
/* Print out header */
printf("Tribanal started.  Files open\n");
fprintf(outFile,"Tribometer Analysis\n");
fprintf(outFile,"Sample, Force, Torque, Correlation\n");
```

```
	/* Exit */
	printf("Tribometer end; hit return to exit\n");
	while (getchar()==0);
	return 0;
}
```

Next, we want to provide storage for the data we will get in. There are several different ways to do this, but here we will do what ever is simplest. First, we will assume that we will never have more than 100 samples. (If we get more, we will need to warn the user and exit.) So, we need to provide storage space for all those samples. If we index the same way we number tha samples, we will need room for 101 of them since there is no sample zero. We will makle the samples global floats. So, we will need to ad the declarations before main:

```
#define MAXSAMPLES 100
float forceArray[MAXSAMPLES+1];
float torqueArray[MAXSAMPLES+1];
int nSamples=0;
```

Notice that the "#define" statement has no semicolon. It is not an executyable statement. It is a "MACRO". A very simple one. It simply says that whenever the compiler sees "MAXSAMPLES" it is to plug in "100". So, both arrays will have 101 floats each. Note also the use of a capital letter (not the initial one) in the global variables. That is a convention to remind us that these are global; we can see them from anywhere in our program. (Globals are defined outside all functions.) Local variables can only bee seen and used within the function where they are defined. We will use all lower case for local variable names. Conventions like this help remind us of what kind of things these are.

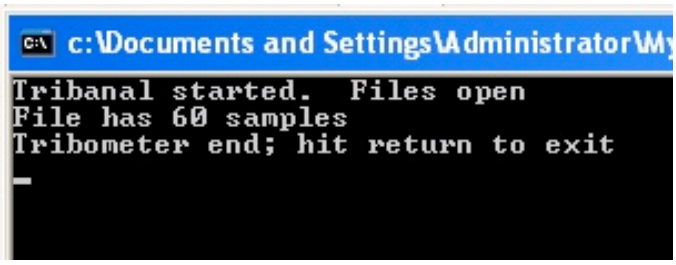
Now, within main, we need code to load up these arrays with the values from the file. The first aspect of this is to read in the number of samples so we can knopw how many samples to read. To so this, we use a "fscanf" function. The fscanf function returns an integer that tells us how many variab;les we have successfully read. So, we need to define an integer in main; call it i. That needs to be done before other executable statements (the fopen calls). We also need a place to put the expected number of samples. We will call that local variable "nsamples". Notice that this is a different variable than nSamples; capitalization matters.

Also, because in C arguments are passed into functions by value, what we need to do for nsamples is pass not the current value of nsamples, but a pointer to wher eit is in memory. That's what the "&" operator does in front of a variable name. It says, a pointer to the variable, not the variable itself. Right now nsamples isn't even initialized. But, the location of nsamples is known. Within the text string we want to read in, "%d" stands for some integer we will read, and put into the variable at the address &nsamples. So, after executing the fscanf function here, we should know how many samples are in the file.

With the added code, we now have:

```
// Tribanal.cpp : Defines the entry point for the console
application.
//

#include "stdafx.h"

#define MAXSAMPLES 100

float forceArray[MAXSAMPLES+1];
float torqueArray[MAXSAMPLES+1];
int nSamples=0;
FILE *inFile, *outFile;

int _tmain(int argc, _TCHAR* argv[])
{
      int i,nsamples;

      /* Open files */
    inFile=fopen("Tribometer sample data.txt","r");
      outFile=fopen("Tribanalout.txt","w");
      if(inFile==0 || outFile==0){
            printf("File failed to open\n");
            while (getchar()==0);
            return 1;}

      /* Print out header */
      printf("Tribanal started.  Files open\n");
      fprintf(outFile,"Tribometer Analysis\n");
      fprintf(outFile,"Sample, Force, Torque, Correlation\n");

      /* Read in samples*/
      i=fscanf(inFile,"Tribometer data  %d Samples",&nsamples);
      if(i!=1){
            printf("Couldn't read number of samples\n");
            while(getchar()==0);
            return 1;}
      i=fscanf(inFile,"Data out: Samplenumber  Force      Torque");
      printf("File has %d samples\n",nsamples);

      /* Exit */
      printf("Tribometer end; hit return to exit\n");
      while (getchar()==0);
      return 0;
}
```

We then test it to make sure it works:



What we need to do is add a loop to read in all the data. Here's how we will do that. We need to add another integer variable j for a place to put the sample number. at the end we have to subtract one from nSamples to cancel the last nSamples++.

```
int i,j,nsamples;    (modified existing statement)

...... (other stuff here)

for(nSamples=1;nSamples<=nsamples;nSamples++){
      i=fscanf(inFile,"%d %f %f",&j,&forceArray[nSamples],
                  &torqueArray[nSamples]);
      if(i<3){
            printf("Got sample discrepancy at %d %d %d\n",
                        nSamples, i, j);
            while(getchar()==0);
            return 1;
      }
}
nSamples--;
printf("%d Samples read.\n",nSamples);
```

Reading in files can be a tricky business. If they were created in an odd format, they may not read properly. For example, text files from a Mac sometimes give trouble. If the file originated in Excel, write it out as a windows text file. Some people will tell you that "fscanf" skips "white space", characters you can't see. That's generally true for spaces. But don't count on it for other things. Without the "\n" in the scanf formatting string, these won't work. Notice the diagnostic printf statement that gets written if a problem develops. That's a useful technique to help in debugging.

Something else to notice is that we can break up a long statement into multiple lines. The ";" ends a statement. we can also put multiple statements on one line. Generally one statement per line is preferred, with blank lines (and comments) separating pieces of code that work together to do something, like read in the data in this case. Also notice the use of indentation to keep track of blocks of code that are grouped together inside if and loop structures. If you do not use some form of indentation, it can be very difficult to keep track of what is going on in lengthy functions.

OK, so the next thing we want to do is the code that does the averaging. To do this we need "nested loops" with an outer loop to move down from sample 6 to the 6th from the end, and a loop inside to count off 11 samples to be averaged. We will also need some new variables to hold the averages. We won't try to hold the results in the program; we'll write them out as we go, into the Tribanalout.txt file.

Here's what we will need to add:

```
float forcesum, torquesum;   (up where we declare variables)
```

```
/* get averages */
for(i=6;i<nSamples-5;i++){
      forcesum=0.0;
      torquesum=0.0;
      for(j=i-5;j<i+5;j++){
            forcesum=forcesum+forceArray[j];
            torquesum=torquesum+torqueArray[j];}
      forcesum=forcesum/11;
      torquesum=torquesum/11;
      fprintf(outFile,"%d, %f, %f\n",i,forcesum,torquesum);}
```

When we run it, and exit, here is what we find in Tribanalout.txt (only the fiorst several lines):

```
Tribometer Analysis
Sample, Force, Torque, Correlation
6, 3.062818, 1.085273
7, 3.063091, 1.085182
8, 3.062454, 1.084364
9, 3.061545, 1.084636
10, 3.061182, 1.084273
.....
```

Now we can read this file with Excel and plot the results.

But, we wanted to do more; we wanted the correlation coeficients as a function of time too. That's more complicated. We need to get the variance and standard deviation for each variable as well as the sum of products. To make a long story short, here 's the code we have to add:

```
#include <math.h>
```
(up after the first #include)

We need the math library to be able to use the square root function. Notice the symbols "<" and ">" used. This indicates that the file is not local to our project folder. It's an "include file" (with file sextension .h) that is kept in a centralized location by Visual Studio 2008 for use by all projects. You should never modify one of these original include files.

```
float forcesigma, torquesigma, sumproducts;
```
(with other declarations)

We need variables to accumulate the variance for both force and torque, and the products of these two variables. Finally, below, we need to use the same structures to accumulate the sums. Notice that we have to zero them all out when we start another set of avarages. (The overall program file is appended to the end of this document.)

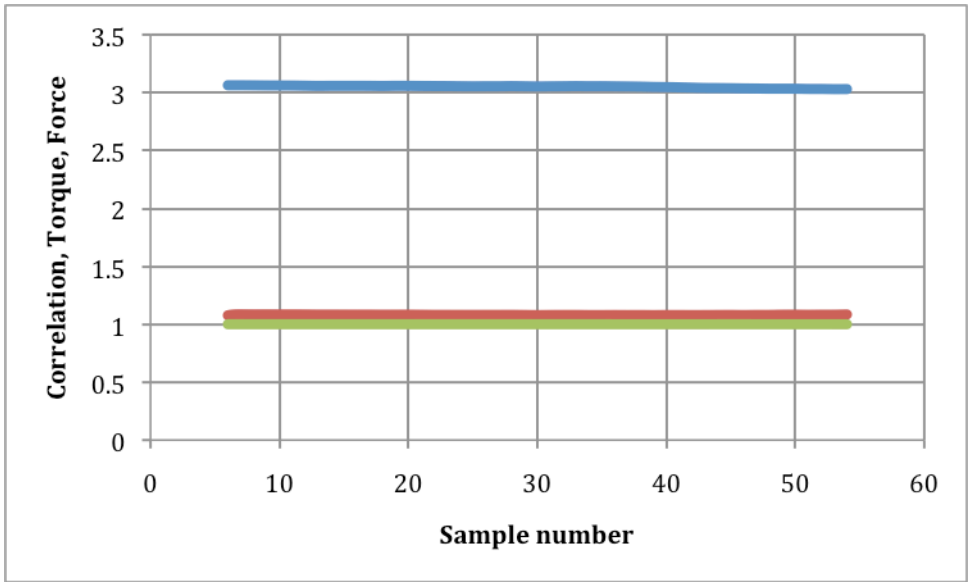
```
/* get averages */
for(i=6;i<nSamples-5;i++){
      forcesum=0.0;
      torquesum=0.0;
      /*actually variance up until - mean squared and do sqrt*/
```

```
            forcesigma=0.;
            torquesigma=0.;
            sumproducts=0.;
            for(j=i-5;j<i+5;j++){
                  forcesum=forcesum+forceArray[j];
                  torquesum=torquesum+torqueArray[j];
                  forcesigma=forcesigma+forceArray[j]*forceArray[j];

      torquesigma=torquesigma+torqueArray[j]*torqueArray[j];

      sumproducts=sumproducts+forceArray[j]*torqueArray[j];
            }
            forcesum=forcesum/11;
            torquesum=torquesum/11;
            forcesigma=sqrt(forcesigma/11-forcesum*forcesum);
            torquesigma=sqrt(torquesigma/11-torquesum*torquesum);
            sumproducts=(sumproducts/11-forcesum*torquesum);
            sumproducts=sumproducts/(forcesigma*torquesigma);
            fprintf(outFile,"%d, %f, %f, %f\n",i,forcesum,
                              torquesum, sumproducts);
      }
```

We now get an output file that looks like this:

```
Tribometer Analysis
Sample, Force, Torque, Correlation
6, 3.062818, 1.085273, 0.999955
7, 3.063091, 1.085182, 0.999950
8, 3.062454, 1.084364, 0.999964
9, 3.061545, 1.084636, 0.999968
10, 3.061182, 1.084273, 0.999971
.....
```

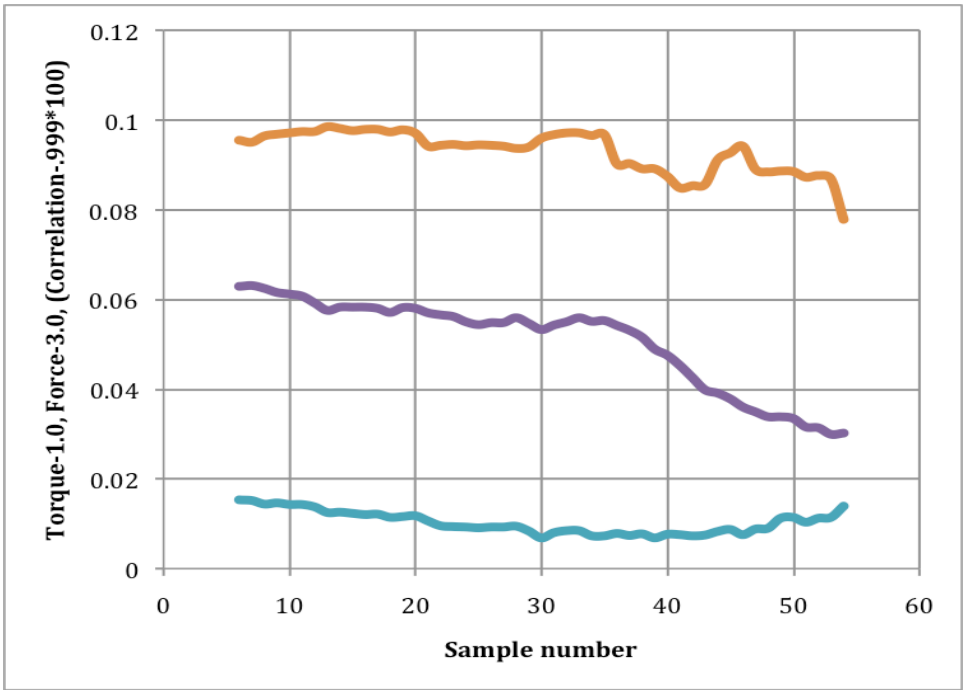
We can read it into Excel and plot it:



Unfortunately, the data varies so little within these few samples that we can't see much. That can be mitigated by subtracting a constant from each value (and in the case of correlation, multiplacation by 100 to amplify the effect). Columns were added in Excel to do that, and now the chart that plots the modified data shows the small variations that may be of interests, and may tell the user something about the friction process and how it changes over time. (Typically there would be many thousands of samples.) Is thgere a trend here? Or is it all just noise? Hard to tell yet, but the trend lines for the averages are changing. The correlation is extremely high, but perhaps that's just because nothing much has happened yet. Or, is there an error in the program? We should go back and check it with known data, for which the outcome is known, to verify the correctness of the program before we do anything serious with these results. [No time to check it yet. I'm suspicious.]

Tribometer Analysis

Sample	Force	Torque	Correlation	Force-3.0	Torque-1.07	(Correlation-.999*100
6	3.062818	1.085273	0.999955	0.062818	0.015273	0.0955
7	3.063091	1.085182	0.99995	0.063091	0.015182	0.095
8	3.062454	1.084364	0.999964	0.062454	0.014364	0.0964
9	3.061545	1.084636	0.999968	0.061545	0.014636	0.0968
10	3.061182	1.084273	0.999971	0.061182	0.014273	0.0971
11	3.060727	1.084273	0.999974	0.060727	0.014273	0.0974



So, here we have an example of C used to do analysis, supplemented by Excel to do the plotting, as well as an example of how to create a useful application.