Programming in C++ using MS Visual Studio 2005

Introduction

This document will give a brief introduction to C++ programming using Microsoft's Visual Studio 2005 environment. The main thrust of the tutorial will be to understand the basic workings of the C++ language for mathematical calculations.

Visual Studio is Microsoft's programming environment for the different languages they support. As in most Microsoft software, it is especially geared for Windows: at the beginning we will only touch the standard C++ language, so it should be possible to compile (more on that later) the programs in other compilers, like Borland's C++ Builder and Gnu's gcc; but the later part will have Windows-specific code.

Compilers

When you use a programming language to write some code you use a grammar that, though not always easy, is readable. This code will allow you to define variables using base-10 numbers, write alphanumerical characters to files and the screen, etc. This **source code** is completely obscure to the computer, since the machine uses binary for everything, from numbers to characters, and, unless told, does not know what a screen or a file is. You need a program that will translate the source code into **binary code**, which is what the computer will run. This program will perform the translation in several steps, and though only one of them can be properly termed as compiling, the whole program is normally called a **compiler**.

C compiling will start with your source code in one or more files, and run the **pre-processor** directives it finds at the beginning of the code (lines that start with "#"). It then will compile the resulting code into **object files**, which contain machine code and other information about variables and objects in the program. The resulting object files are taken as input by the **linker**, which will link them and add several the libraries the program needs to run properly. The end result will be a binary-code file the computer can use.

The binary files are dependant on the type of computer, processor and operating system you use; so do not try using a program compiled on a Windows XP Intel Pentium machine with an UNIX Sparc system, it will not work. You will have to recompile the source code into binary-code files for the appropriate system you are using[1].

C++

The C programming language came into being between 1969 and 1973 for use in UNIX. It was later 'ported' to other operating systems and enjoyed great popularity due to its flexibility and robustness.

From 1979, the Dane Bjarne Stroustrup[2] implemented an extension of C introducing object-orientation and other changes. This language, originally known as "C with Classes" and renamed as C++ in 1983, has gained an even wider acceptance than the original C[3].

---

[1] Some languages, like Java, avoid this limitation by having the computer do the final machine-specific stages of the compilation as it runs the program.

[2] http://www.research.att.com/~bs/homepage.html.

[3] See http://www.levenez.com/lang/ for more information on programming languages.

Visual C++ is Microsoft's implementation of C++ for Win32 machines (Windows 2000, XP and Vista). It is designed for programming at a higher level than we are going to, so we will not use most of its utilities.

An example

- Once you have logged in your computer, open the Start menu, move to All Programs → Microsoft Visual Studio 2005 and click on Microsoft Visual Studio 2005. This will start the programming environment

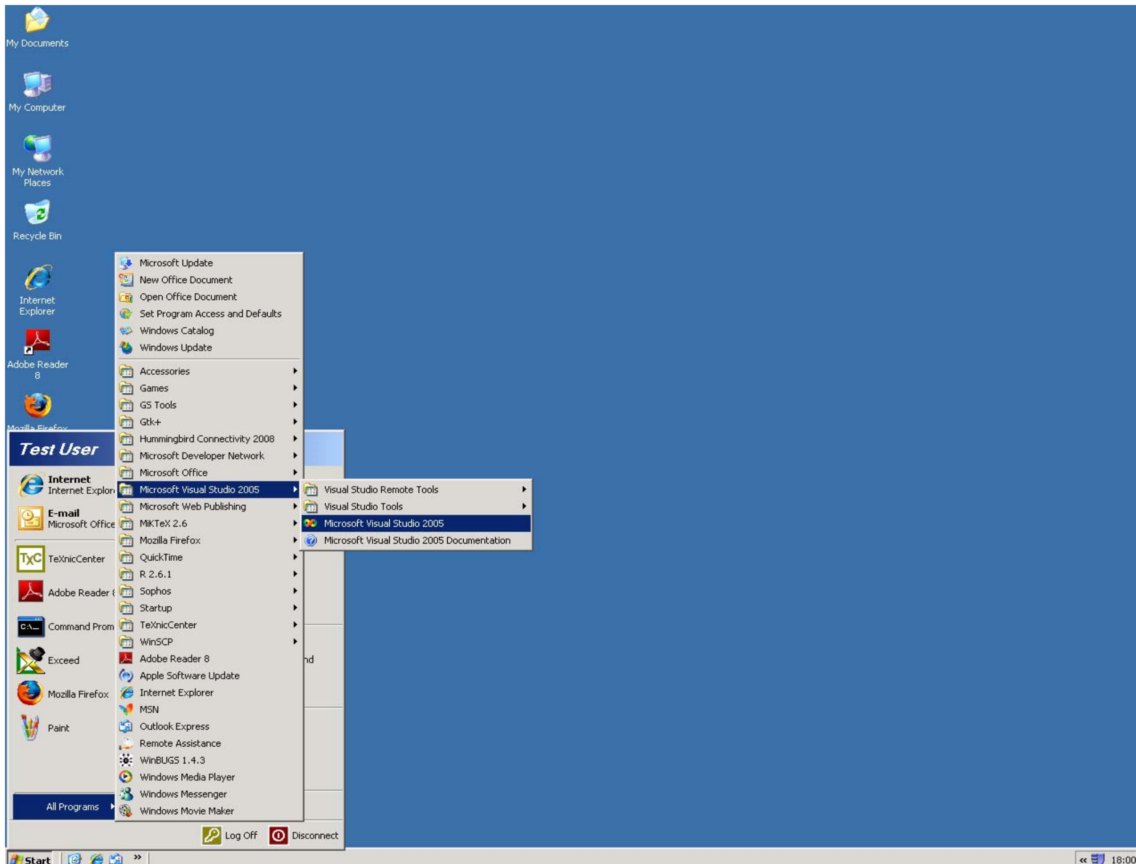

Figure 1. Opening Visual Studio

- Select the Visual C++ Development Settings option and click on Start Visual Studio
- Wait a few seconds for the environment to be initialized; you only need to do this once
- Click on the File menu and select New and Project…
- Select the Visual C++ Win32 project type
- Select the Win32 Console Application template
- Name your project[4] `tutorial1` in the Name box
- Leave the default path in the Location box
- Leave the Create directory for solution option checked (Figure 2)
- Press OK
- On the Win32 Application Wizard – tutorial1 window, click on Next >
- Uncheck the Precompiled header option

[4] Do not use spaces in the names of projects and files; it can cause problems when you run them.

- Check the Empty project option
- Click on Finish

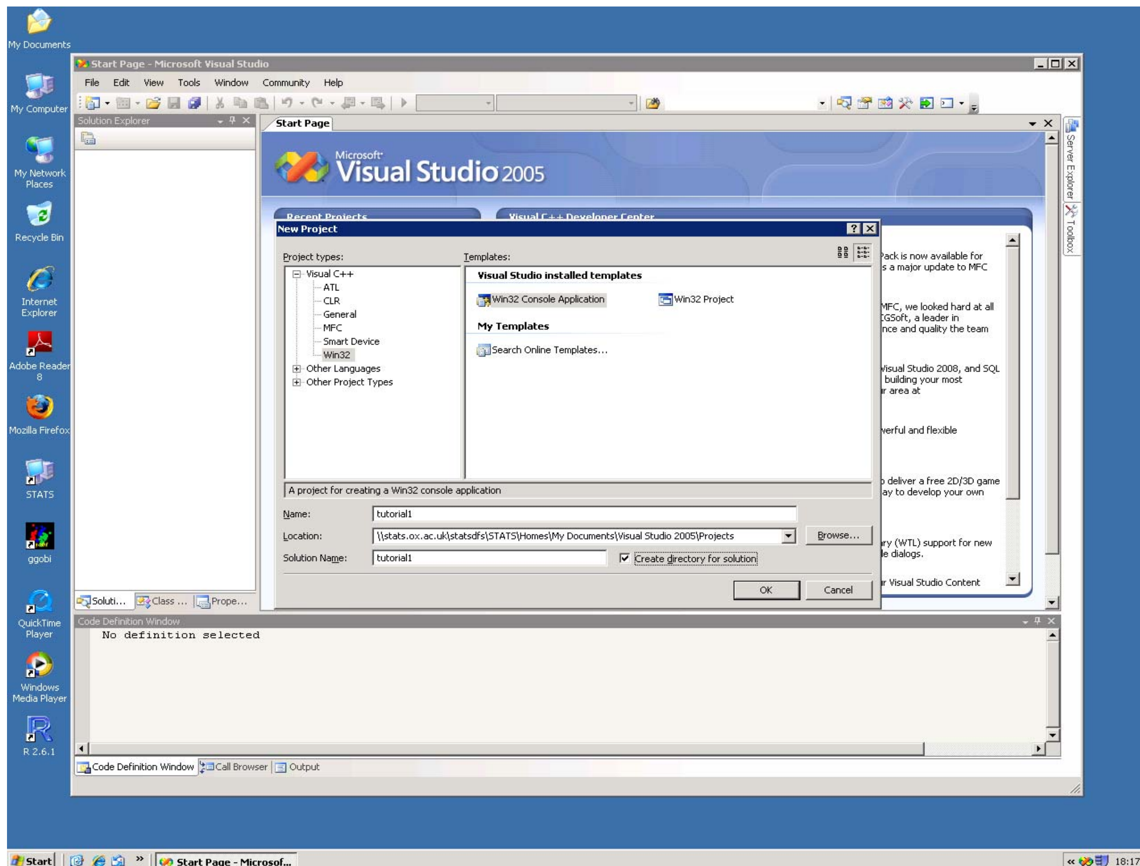

Figure 2. Naming your project

Visual Studio has created an empty-project folder in the path specified (the folder itself is not empty though). Now we are going to add a file to this project:

- Right-click on **tutorial1** in the Solution Explorer – tutorial1 panel, and select Add → New Item…
- On the Add New Item – tutorial1 window, click on Code in Categories
- Click on the C++ File (.cpp) in Templates
- Type a Name for it, like `tutorial1`[5]
- Click on Add (Figure 3)

This will create a source file (tutorial1.cpp) and will include it in the project folder. It will also open a window on the screen showing its contents (Figure 4).

[5] The name of the file needs not be the same as the project, but it looks tidier.

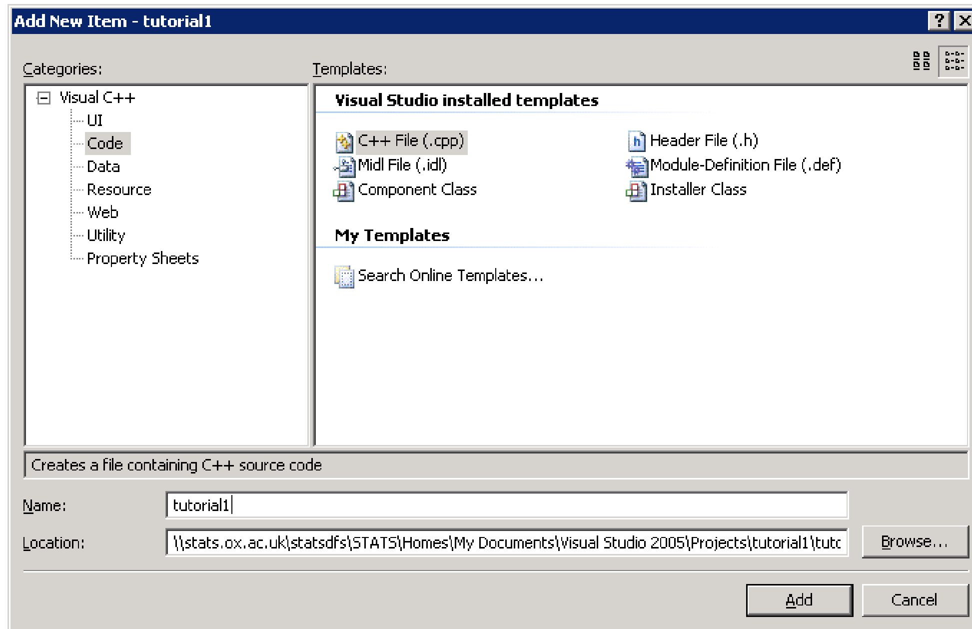

Figure 3. Creating the first file

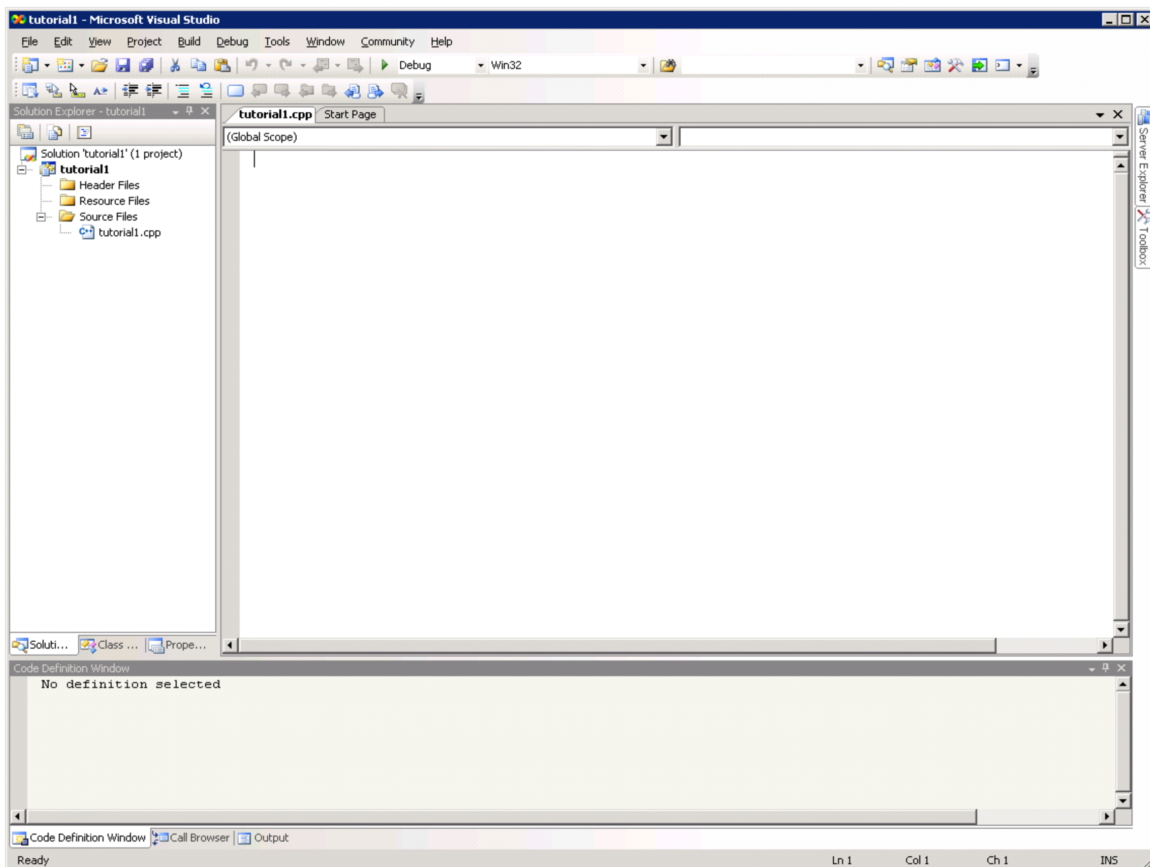

Figure 4. Our first file awaits some code

- Type the following code in the file window, a line-by-line explanation will follow in the next section:

```
#include <iostream>        // This is to include the console IO libraries
using namespace std;       // This is to use the standard namespace

int factorial(int n);      // Declaration of the function factorial

int main()                 // The program itself
      {
      int a;               // Declaration of the integer variable a
      cout << "Please insert a number: ";
      cin >> a;            // Getting the value of a from the keyboard
      cout << '\n';
      cout << "The factorial of " << a << " is " << factorial(a) << '\n';
      return 0;            // main returning 0 to the OS
      }

int factorial(int n)       // Definition of the function factorial
      {
      if(n>0)
            return (n * factorial(n-1));
      else
            return 1;
      }
```

- Go to the Build menu and select Build Solution (Figure 5)

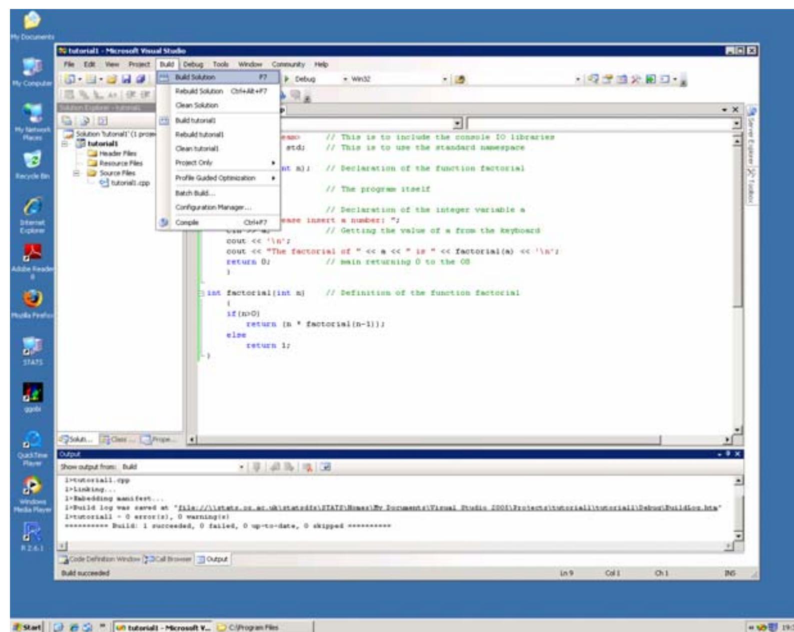

Figure 5. Building the project

This command will compile the program and create an executable file named as your project in the debug subfolder of your project folder.

You should get a `Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped` message in the bottom window signifying a successful compilation. If you get any error messages (things that will not compile) or warnings (things that will compile, but that the compiler thinks they could be a programming error), use the line numbers specified in the messages to check your code for typing errors.

There are three main ways to run the file you just created: browse to and double-click on it[6]; open a command prompt, change the directory to where it is and type its name; or click on the Debug menu and select Start Without Debugging.

The program will ask you for a number (Figure 6) and print out its factorial. If you ask for the factorial of big integers be prepared for strange answers; all variables in C++ have a maximum value that can be held in the memory provided, and if you overflow that value you will get nonsensical, even negative, values. The program works well up to 13.

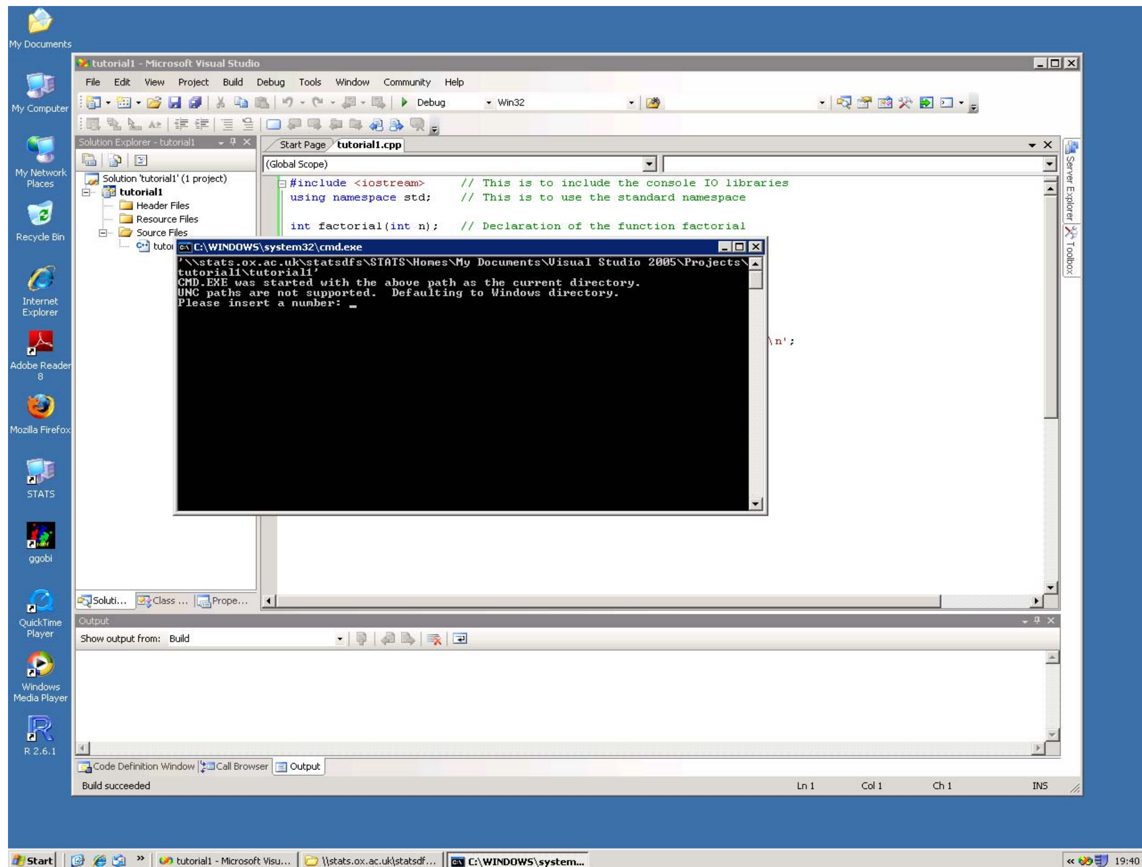

Figure 6. Running the program

What is it about?

Let's see what the program does:

```
#include<iostream>
```

is a pre-processor command that tells the compiler to include the iostream library. This library will be found by the compiler (it knows how to) and added at the beginning of the code before it is properly compiled. This is done to allow us to use the cin and cout functions in the body of the code; it essentially warns the compiler we are going to use functions from the iostream library (Input/Output Stream: standard library of functions to read from and write to the keyboard, screen and other devices) that comes with most C++ compilers. This way we do not have to create those libraries ourselves and are able to use ones created by someone else previously;

[6] As this program outputs the result to the console and this disappears quite fast, this is not very useful now, but with programs that output to a file it should not be a problem.

```
using namespace std;
```

specifies that all subsequent iostream functions called in the program use the standard namespace, std. It is the same as changing all instances of cin and cout for std::cin and std::cout;

```
int factorial(int n);
```

is the declaration of the factorial function; it tells the compiler we have written a function called factorial, that it takes an integer value as argument, and returns another integer value as result;

```
int main() {…}
```

is the program itself: all C++ programs that compile to an executable file need to have a main() function; here is where the execution will start. All the code between the braces is the body of the program and will be executed sequentially;

```
int a;
```

is the declaration of a variable of type integer and name a. The variable's name and type will be stored in the memory, and space will be made in it to house an integer value; Warning: C++ will not give this variable any value until you initialise it, it will only make space in memory for it;

```
cout << "Please insert a number: ";
```

will print "Please insert a number: " in the standard output, in our case to the screen[7];

```
cin >> a;
```

will read the value you type on the keyboard (you have to press `Enter` at the end) and store it in the variable a;

```
cout << '\n';
```

will start a new line in the screen, so that what we print next is separated from the previous input by an empty line;

```
cout << "The factorial of " << a << " is " << factorial(a) << '\n';
```

will print the text "The factorial of ", then the value of a, then the text " is ", then it will calculate the result of calling the factorial function with argument a (see below) and print it, and then it will go to the next line. This technique of having code that calculates the result and generates the output with it is very common;

```
return 0;
```

will terminate the program returning 0 as the result of the main function. This is not strictly required, but some operating systems like programs to return 0 if they have run successfully;

```
int factorial(int n) {…}
```

is the definition of the function we declared earlier: we are going to use a function called factorial that will take an integer argument called n inside its body, and will return another integer value as a result. The code inside the braces (the body of the function) will be executed whenever the function is called;

[7] The standard output for your PCs is the screen and the standard input the keyboard. Other types of machines can have different standard input/output devices depending on their architecture.

```
if(n>0) return (n * factorial(n-1));
```

makes the function check to see if the variable called n is greater than 0 and, if it is, will return the result of multiplying the variable by the factorial of the variable decreased by one (this in turn will call the factorial function again, and so on successively until the n of the latest called function is 0; this technique is called recursion);

```
else return 1;
```

will terminate the execution of the function and pass control back to the `main` program; it returns 1 as the result of `factorial` when the previous if fails (0! = 1).

One point to notice: the compiler does not care about "whitespace", that is, any spaces or carriage returns you put in between the commands; as far as the compiler is concerned, the whole file is a single long line of text; the way you organize that text is irrelevant as long as it is in the right order and the command words are not broken or joined.

Visual C++ runs a "spelling" checker as you type, and changes the colour of the text to highlight what it is: command words are coloured blue and comments green.

You do not need to follow the way I organize the lines, just decide on a way that is clear for you and be consistent with it!

A bagful of tricks

The most useful trick you can use in Visual C++ is the help files that come with it: in your source code file window, move the text cursor (the vertical blinking line) to one of the commands (in blue) and press `F1`. The first time (and only the first time) you do this you will be asked where to get your help from: online or local. In our case, the local help is the Microsoft Developers' Network Library (MSDN) for Visual Studio 2005 installed on the computers, which we will use. On the Online Help Settings window, select the Use local help as primary source option and click on OK (Figure 7).

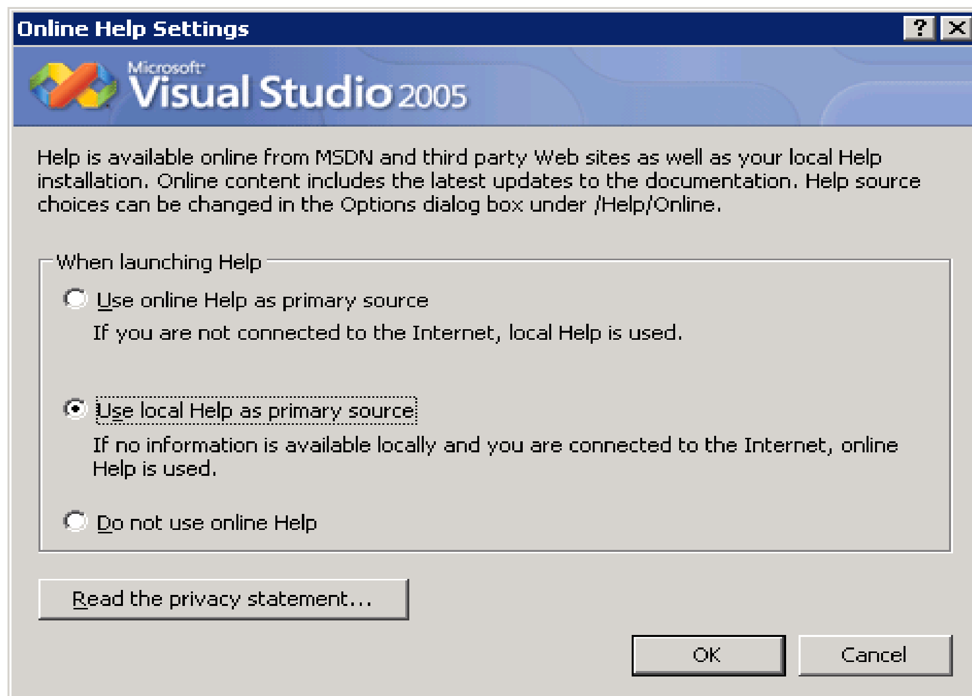

Figure 7. Selecting help primary source

A new window will appear and show you the information it has on the command you selected (in our case, the `return` command; see Figure 8).

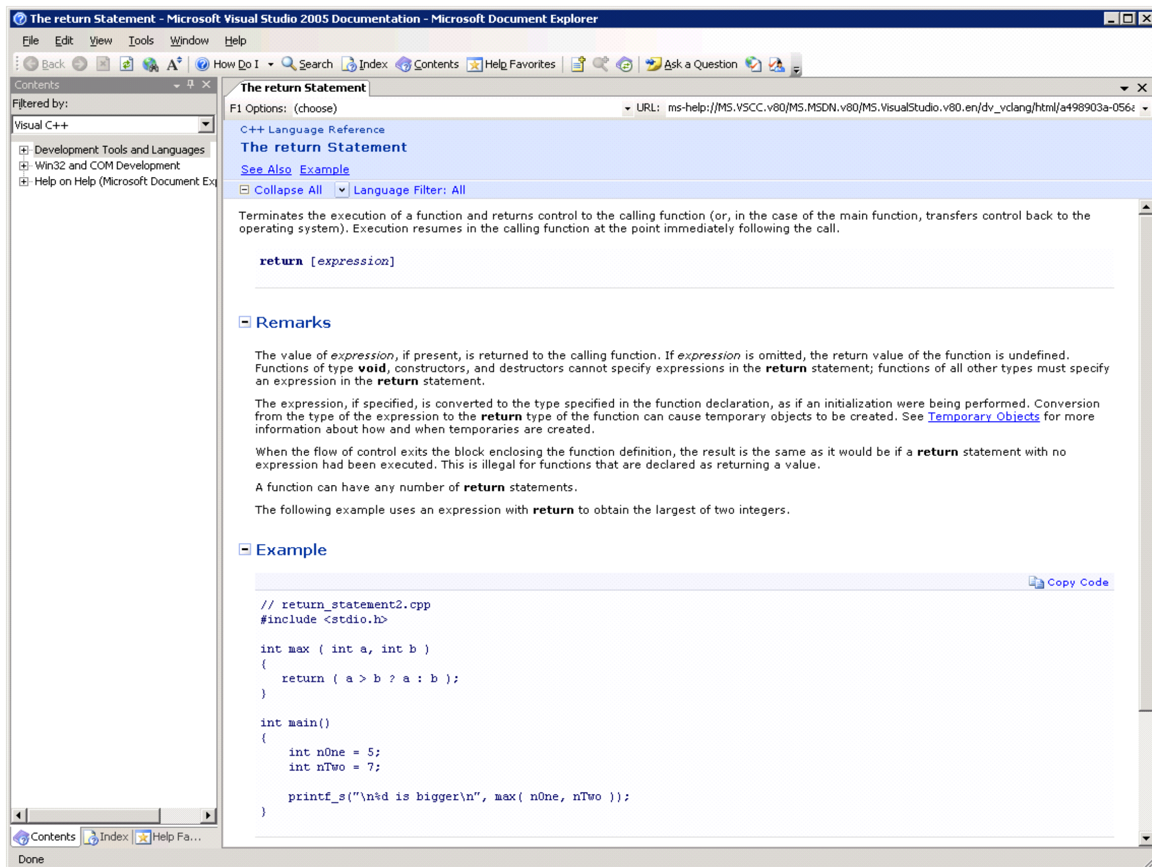

Figure 8. Microsoft Developer's Network Library

The MSDN Library contains almost anything you need to know about the compiler and the language it compiles, and can be of much help if you are writing C++ for the first time. It contains specifications for all the libraries and grammar, though it usually gives you more information than you need. You are encouraged to experiment with it, but we will not cover it here as it could fill many pages just to use its basics.

Let's be mean

We will need a new project for the next example:

- If you are still inside Visual Studio, go to File → Close Solution
- If the system asks about saving changes, click Yes
- Create a new Win32 Console Application project named `tutorial2`
- Type the following code in a C++ File (.cpp) named tutorial2.cpp:

```
#include <iostream>
#include <stdio.h>
using namespace std;

int main(int argc, char** argv)
	{
	FILE *fd;
```

```
int n=0;
float y, sum=0.0;

if(argc < 2)
      {
      char filename[100];
      cout << "File: ";
      cin >> filename;
      cout << '\n';
      fd = fopen(filename, "r");
      }
else
      fd = fopen(argv[1], "r");

if (fd == NULL)
      {
      cout << "File open failed\n";
      return 1;
      }

while(fscanf(fd, "%f", &y) ==1)
      {
      n++;
      sum += y;
      }

if (n > 0)
      cout << "The mean is " << sum/n << '\n';

return 0;
}
```

- Build the solution

  For this program we will need a file of random numbers:

- Open R (Start → All Programs → R 2.6.1 → R 2.6.1) and run the following commands in the R Console:

```
a <- rnorm(250)
cat(a, file="P:/random.dat", sep="\n")
```

A file called random.dat will be created in your P:\ folder.

- Calculate the mean of those numbers by running `mean(a)` in the R Console

To use this file with the new program, you must copy it to the same folder as the executable file (`P:\My Documents\Visual Studio 2005\Projects\tutorial2\debug`) and run it from the command prompt, typing the filename of the random-number file after the name of the program (`tutorial2 random.dat`).

If you are executing the program within Visual Studio, you must specify the path and name of the file when the program asks for it (`P:\random.dat`).

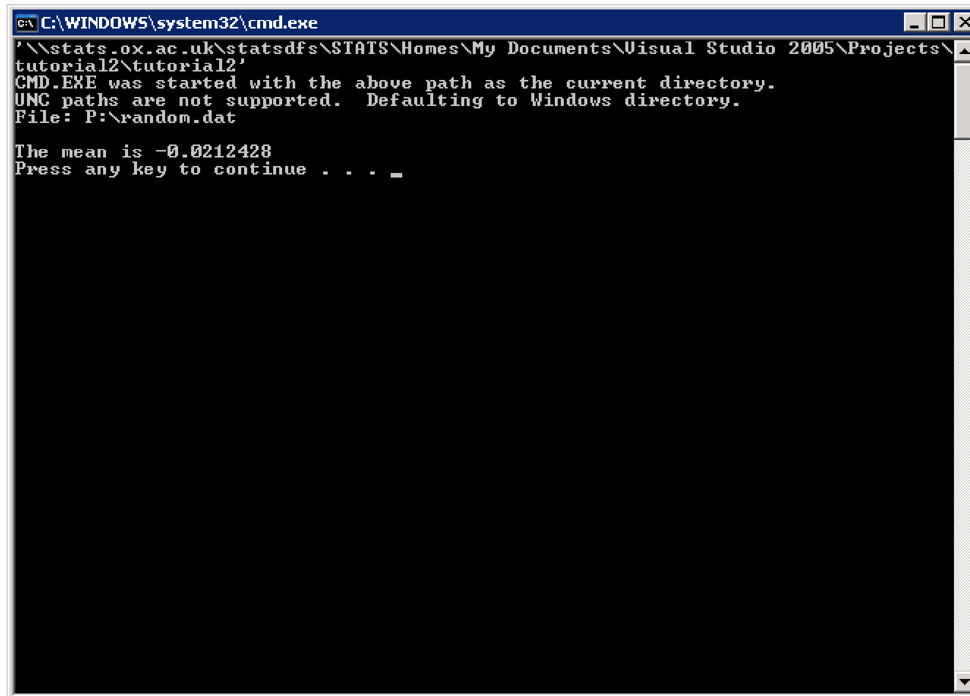

Figure 9. Running tutorial2

What's new

`#include <stdio.h>`

is another pre-processor command that includes a header file; stdio.h allows us to open and read files using the functions fopen and fscanf, and defines FILE objects;

`int main(int argc, char** argv) {…}`

is another way of defining the main function: this time passing the arguments we type at the command line as argv[0], argv[1], argv[2], etc., and the number of such arguments as argc. argv[0] is always the name of the executable we are running;

`FILE *fd;`

declares fd as an object of type FILE; these objects are defined in the stdio.h file;

`float y, sum=0.0;`

declares two floating-point numbers, y and sum, and initialises sum to 0;

`char filename[100];`

declares a string of characters (alphanumeric) of length 100 and name filename. When inputting data into this variable you must make sure you do not exceed the maximum length of 100 characters or unexpected things may happen;

`fd = fopen(filename, "r");`

initialises fd with the file we supply to it, stating the file can only be read. If the file cannot be opened for some reason, fd will be given the NULL value;

`fscanf(fd, "%f", &y)`

reads the next floating-point number from the fd file and stores its value in the y variable. The function as a whole returns a 1 result if the reading operation was successful;

`fd == NULL`

is a condition that compares the expressions before and after the `==` and is true if they are equal;

`while(…) {…}`

runs whatever is between the braces while the condition inside the parenthesis remains true;

`n++;`

increases n by one. It is equivalent to n = n + 1;

`sum += y;`

is equivalent to sum = sum + y.

The code is written so that the name of the data file can be supplied either at the command prompt or when asked by the program itself. This is common and good practice.

A couple of tricks

- Once you have run the previous program a couple of times, click on the grey vertical bar besides the `n++;` line in the tutorial2.cpp window; a red dot will appear marking a breakpoint

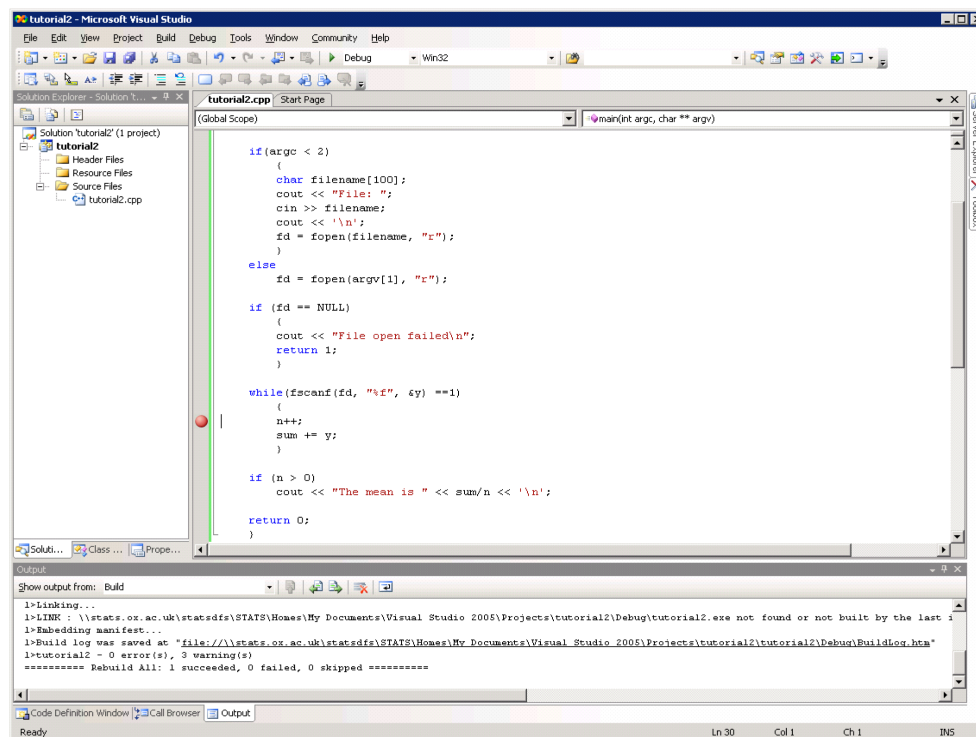

Figure 10. Breakpoint

- Press the `F5` key on your keyboard

The program will run again, but this time, it will stop when the execution reaches the breakpoint. Here Visual Studio will allow us to peek inside the running of the program to discover any mistakes we could have made; this is what is called **debugging**. To be able to

see the programming environment you have to change the window focus by clicking once on its window, otherwise the uppermost window will remain the Command Prompt window where the program is running.

You will notice two panels at the bottom of the screen (Figure 11). The first shows the objects being used by the program and is by default on the Autos tab; the second shows any messages from the environment and is by default on the Call stack. If you click on the Watch 1 tab of the first panel, you will see it has two columns with Name and Value at their heads. This one is the one we will use.

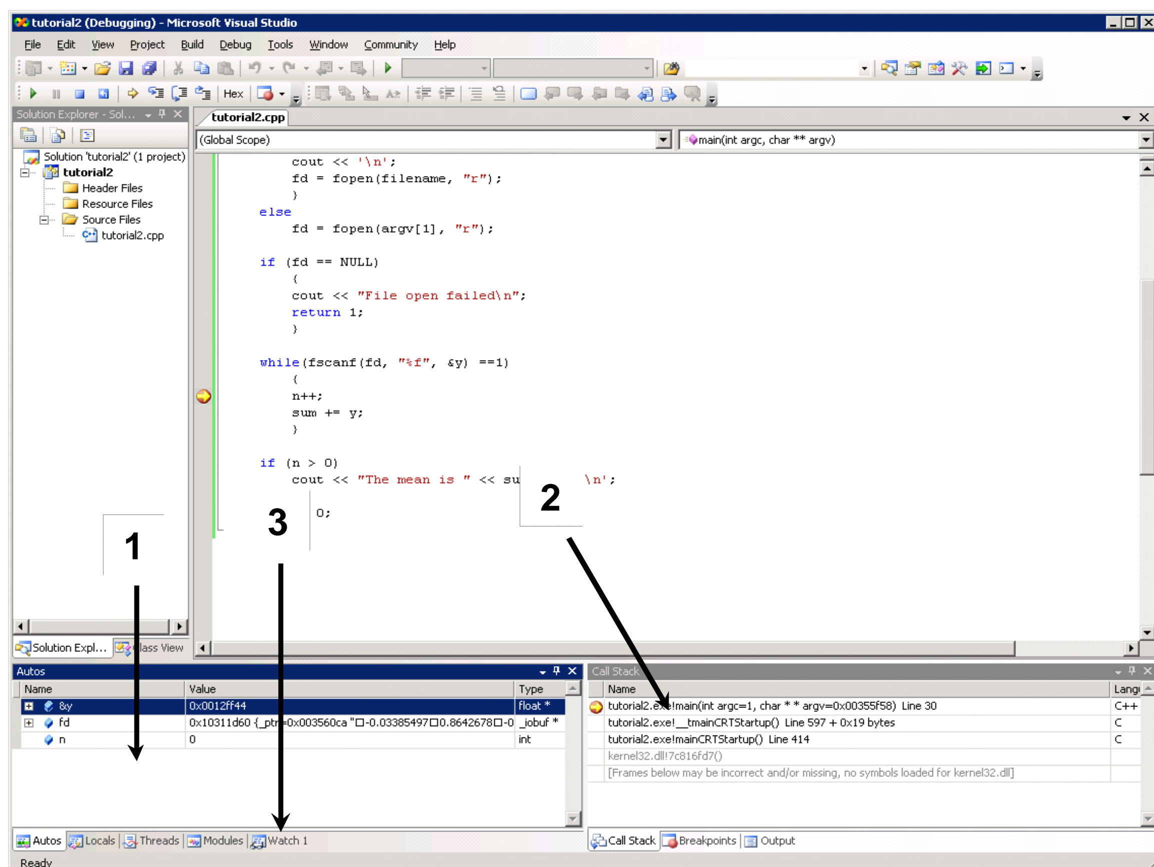

Figure 11. Debugging panels

- Click on the first space in the Name column
- Type `sum` and press `Enter`

The value of the variable sum at this stage in the execution of the program will appear on the corresponding space in the Value column (Figure 12). So far this value is 0 because we have not added any of the numbers in the file yet.

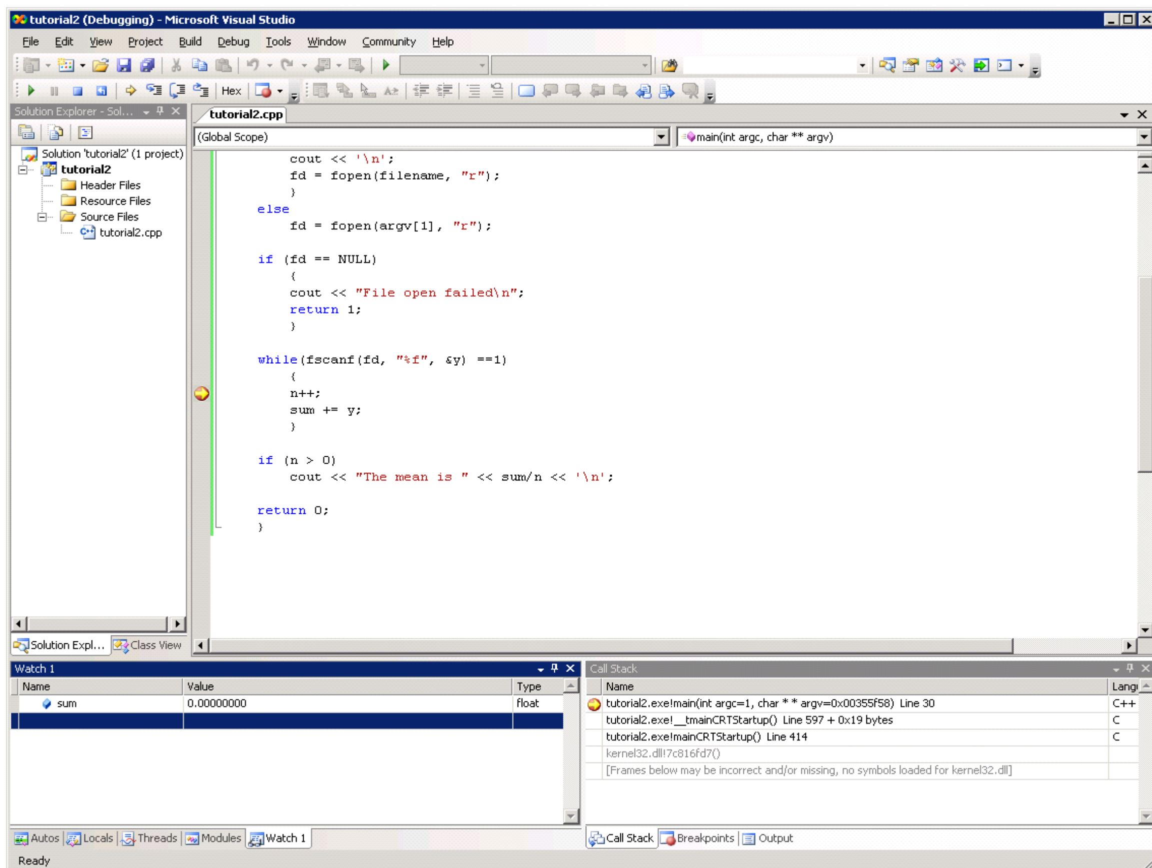

Figure 12. Looking inside a running program

- Press `F5` again with the window focus on the programming environment

After a blink, the execution will stop again at the breakpoint (it is inside the while loop) and now sum has a different value: we have now added the first number in the random-number file to it. Subsequent presses of `F5` will show us how sum is increased with the numbers taken from the file.

To get rid of the breakpoint:

- Right-click on the red dot that marks it
- Select Delete Breakpoint on the menu.

The yellow arrow will remain until you press `F5` and the program runs to its end without interruption.

You can use more complicated expressions in the Name column like 2*sum or sum/n if you need to evaluate those expressions during the run.

To delete a variable from the watch list:

- Click on it so it is highlighted
- Press the `Delete` button in your keyboard

Using these facilities of Visual Studio you can examine the running of a program to find out what is wrong with it if it does not do what you expect.

The breakpoints can be set in any line you want, but will have no effect if set on declaration lines, those lines are not really "executed", they are just warnings to the compiler of things we are going to use.

A breakpoint inside an if (or any other conditional) statement will not stop the program unless its condition is true because the debugger will not process these lines.

- Close the solution

It's good to talk

This final section will show how to create C++ code that can be used by R (and other programs). To do it we will create a **dynamic-link library** (dll) exporting one function for R to use.

A dynamic-link library is a binary code file that allows any program that loads it to use the functions it exports. These functions will be written in C++ and compiled with another file (the definition file) that tells the compiler which of the functions it is exporting and what name to give those functions.

Warning: Some of the following code is Microsoft Windows specific.

- Create a new project called tutdll, but select Win32 Project as the template (Figure 13)
- Click OK
- Next >
- With DLL and Empty project selected, click on Finish

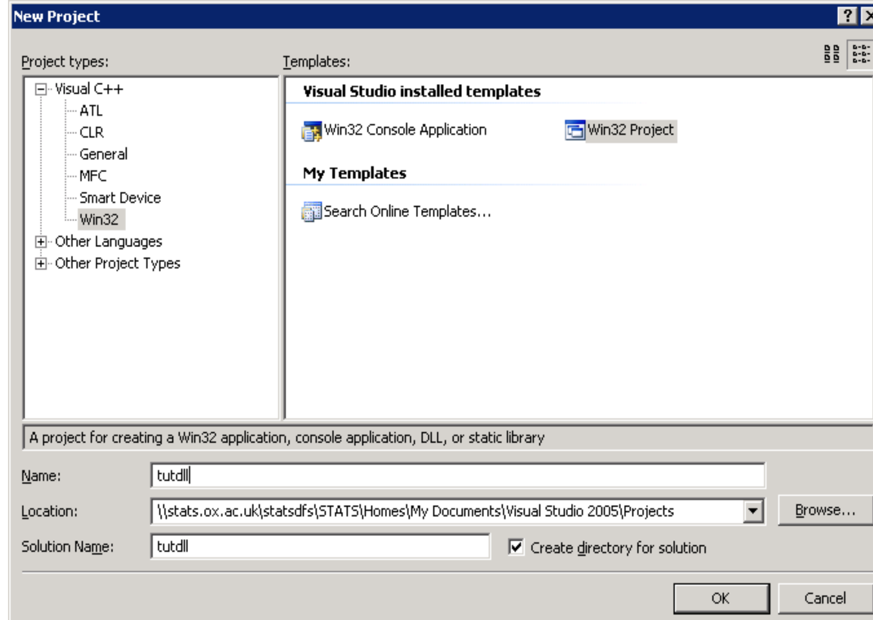

Figure 13. Creating a Win32 DLL project

- Add a C++ File (.cpp) named tutdll.cpp with the following code in it:

```
#include <math.h>

void rsum(double* x, int* pn, double* mean, double* stdv)
     {
```

```
int n = *pn;
double sum = 0;

for(int i=0; i<n; i++)
       {
       sum += x[i];
       }

*mean = sum/n;
}
```

- Add a Module-Definition File (.def) called `tutdll.def` to the project (Figure 14)
- Change the contents of the new file to:

```
LIBRARY “tutdll”
EXPORTS
       rsum
```

- Once you have the two files ready, click on the Build → Build Solution

This will create a tutdll.dll file in the debug subfolder in your project folder.

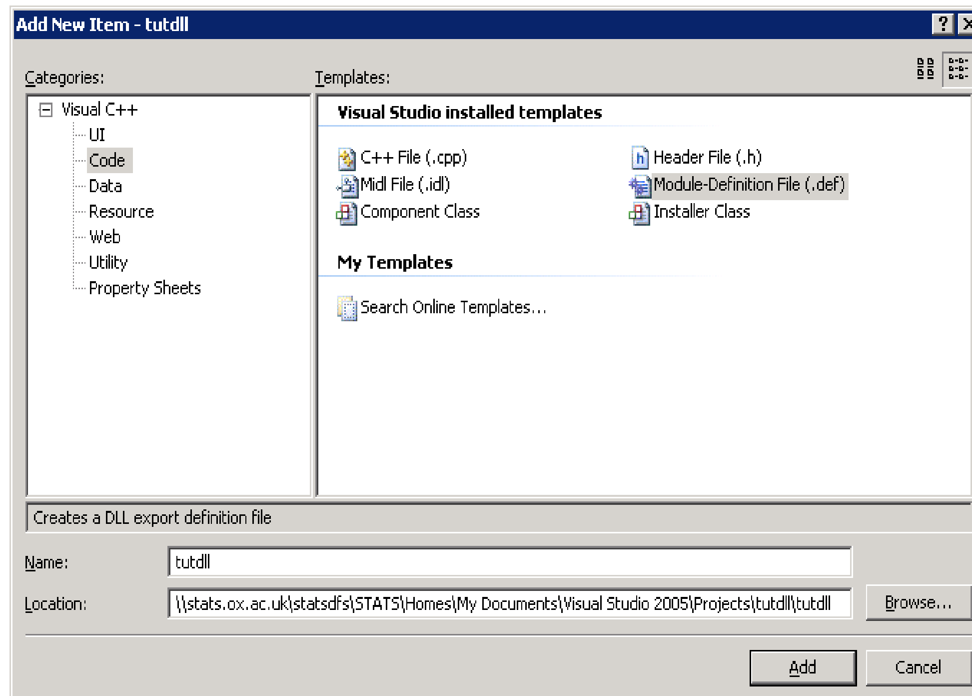

Figure 14. Adding the definition file

Let’s get dirty

Some new concepts coming as we explain the code for the `tutdll` project:

```
#include <math.h>
```

includes another header file you will be using quite extensively, math.h. It contains the most used mathematical functions for C++;

```
void rsum(double* x, int* pn, double* mean, double* stdv)
```

defines a function that does not return any values, hence the void before its name. It takes as arguments four **pointers**, three which point to double values (double*) and one which points to an int value (int*);

```
int n = *pn;
```

declares an integer variable called n and initialises it with the value in the pointer pn. A **pointer** is a name for the address in memory of a variable, and by writing *pn in the body of the function we are referring to the value stored in said address pn. We use them because this is the way R works with dlls;

```
double
```

is a variable similar to a float, but with double the space in memory, so it can store more precise numbers. The corresponding double-length variable type for integers is long;

```
x[i]
```

refers to the i-nth element of the x array. x is passed to rsum as a pointer, together with its dimension, pn. This is a standard way for C++ to work with vectors, but it forces us to pass the length of the vector separately, because the program has no way of knowing we are passing several values through the pointer instead of just one. Be very careful not to overstep the limits of the array; its index goes from 0 to its length minus one, but the compiler will not stop you from reading past the last element, effectively reading areas of the memory full of garbage. This is the most common source of bugs in C++.

R

Now we have our DLL, it is time to use it:

- Open R and type the following command in the R Console (be careful to enter the path to the file correctly):

```
dyn.load("P:/My Documents/Visual Studio
      2005/Projects/tutdll/debug/tutdll.dll")
```

This command will load the DLL in memory.

- Once loaded, the function can be used by R:

```
y <- rnorm(250)
z <- 0
x <- 0
x <- .C("rsum", as.double(y), as.integer(250), as.double(x),
      as.double(z))[[3]]
```

which will store the result of the mean of 250 random values in x. You can compare the results by typing `mean(y)` and x. The number 3 between double square brackets in the last command tells R to use the third value in the list of arguments of the rsum function.

To unload the dll from R memory, run

```
dyn.unload("P:/My Documents/Visual Studio
      2005/Projects/tutdll/debug/tutdll.dll")
```

The functions defined in the DLLs can be as complicated as you like, but they all have to follow these guidelines set by R:

- they must not return any values in themselves (void);
- the arguments they have must be pointers;
- any results must be returned through the arguments.

F. David del Campo Hill
Department of Statistics
delcampo@stats.ox.ac.uk
8th May 2008

Exercise

As you have probably noticed I have written an extra pointer, stdv, in the rsum function of the tutdll example, but this pointer is not used anywhere. This one is for you!

Write some extra code in the rsum function so that stdv returns the standard deviation for R to read.

Note: The math.h library has a function, sqrt(), which takes a double as argument and returns its square root (also a double) as result. Also, when using R, do not forget to change the `[[3]]` for a `[[4]]` in the command that runs the function if you are referring to stdv.

Solution

One possible solution to the exercise is the following code in the tutdll.cpp file:

```
#include <math.h>

void rsum(double* x, int* pn, double* mean, double* stdv)
      {
      int n = *pn;
      double sum = 0, sqrsum = 0;

      for(int i=0; i<n; i++)
            {
            sum += x[i];
            }
      *mean = sum/n;

      for(int j=0; j<n; j++)
            {
            sqrsum += ((x[j] - *mean)*(x[j] - *mean));
            }
      *stdv = sqrt(sqrsum/(n-1));
      }
```