

# A brief introduction to C++ and Interfacing with Excel

ANDREW L. HAZEL

School of Mathematics, The University of Manchester  
Oxford Road, Manchester, M13 9PL, UK

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	The programming work cycle . . . . .	4
1.2	The simplest C++ program . . . . .	4
1.3	The C++ keywords . . . . .	5
1.4	Syntax of the source code . . . . .	5
1.5	Practicalities: compiling and running C++ . . . . .	6
1.5.1	Simple command-line compilation . . . . .	7
1.5.2	Visual Studio Compilation . . . . .	8
<b>2</b>	<b>Getting started with C++</b>	<b>12</b>
2.1	Data types . . . . .	12
2.2	Variables . . . . .	12
2.2.1	The scope of variables . . . . .	13
2.2.2	Arrays . . . . .	13
2.2.3	Dynamic memory allocation and pointers . . . . .	14
2.3	Manipulating Data — Operators . . . . .	17
2.3.1	Arithmetic Operators . . . . .	17
2.3.2	Relational and logical operators . . . . .	18
2.3.3	Shorthand operators . . . . .	19
2.4	Talking to the world — Input and Output . . . . .	19
2.4.1	A note on namespaces . . . . .	21
2.4.2	File I/O . . . . .	21
2.5	Conditional evaluation of code . . . . .	22
2.5.1	if . . . . .	23
2.5.2	The ? command . . . . .	24
2.5.3	switch . . . . .	25
2.6	Iterative execution of commands — Loops . . . . .	26
2.6.1	for loops . . . . .	26
2.6.2	while and do-while loops . . . . .	28
2.7	Jump commands . . . . .	29
<b>3</b>	<b>Functions</b>	<b>31</b>
3.1	Where to define functions . . . . .	32
3.2	Function libraries . . . . .	34

3.3	Modifying the data in function arguments . . . . .	35
3.4	Function overloading . . . . .	36
<b>4</b>	<b>Objects and Classes</b>	<b>38</b>
4.1	Object-oriented programming . . . . .	38
4.2	The C++ class . . . . .	38
4.3	Constructors and Destructors . . . . .	43
4.4	Inheritance . . . . .	46
4.4.1	Overloading member functions . . . . .	48
4.4.2	Multiple inheritance . . . . .	51
4.5	Pointers to objects . . . . .	51
4.6	The <code>this</code> pointer . . . . .	52
<b>5</b>	<b>Interfacing C++ and Excel</b>	<b>53</b>
5.1	Writing a simple Excel Add-In in C++ . . . . .	53
5.1.1	Creating the DLL in Visual Studio . . . . .	53
5.1.2	Calling the library functions from within Excel . . . . .	56
5.2	Using Excel from within C++ . . . . .	58
5.3	The Excel Object Model . . . . .	60
5.3.1	The Excel Application . . . . .	61
5.3.2	The Excel Workbook . . . . .	61
5.3.3	The Excel Worksheet and Ranges . . . . .	61

## 1 Introduction

Computers are an essential tool in the modern workplace. They are ideally suited to the processing, analysis and simulation of data. In order to use a computer to solve a particular problem, however, we must generate a list of instructions for the computer to follow, a task known as **programming**. For common tasks, the lists of instructions will almost certainly have been created by somebody else and we can use their work to solve our problem. For example, Microsoft Excel contains the instructions to perform a huge number of standard tasks. Nonetheless, Excel cannot do everything; and for unusual, or new, tasks a new set of instructions must be written. The aim of this course is to provide you with the necessary skills to generate your own lists of instructions so that you can use computers to perform (almost) any task.

A programming language is a set of keywords and syntax rules that are used to “tell” the computer what you want it to do. A list of instructions written in a programming language is called a **program** and is most often created using text editors or the text editor component of an integrated development environment. Ultimately, these instructions must be translated from the programming language into the native language of the computer (assembly instructions). The translation is performed by specialised programs known as **compilers**. In order to use a programming language you must have a compiler for that language installed on your computer.

At the time of writing, there are hundreds, if not thousands, of different programming languages, each with different strengths and weaknesses. The choice of programming language is driven in part by the nature of the project. Excel Visual Basic for Applications (VBA) is the native language of Excel and is ideal for writing small extensions, or macros, within Excel Worksheets. In essence, Excel is the compiler for Excel VBA. VBA is not suitable for every task, however. One restriction is that the language is not very portable; every Excel VBA program must run from within Excel, which means that you need to have Excel installed on your computer. In addition, for intensive numerical calculations, VBA can be rather slow.<sup>1</sup>

C++ is perhaps best described as middle-level, computer-programming language. It is highly portable (there are compilers for almost all computers), efficient and very popular. In these notes, we shall discuss how to use C++ on its own and in conjunction with Excel, allowing the development of powerful and efficient solutions to complex problems.

---

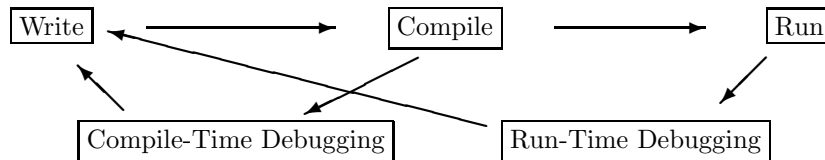
<sup>1</sup>VBA is an interpreted language; its commands are translated into assembly language “line by line” and the translation takes up time during the execution of the program.

## 1.1 The programming work cycle

The compiler is responsible for checking that what you have written is legal; *i.e.* that you have obeyed all the syntax rules and not violated any restrictions in the language. When programming, a large amount of time is spent correcting your program in response to compiler errors.

Once compiled, or built, the program must then be run, or executed, at which point the computer will carry out the tasks specified in the program. Just because a program compiles, however, does not mean that it will run. It is perfectly possible to write a syntactically correct program that tries to perform an illegal action, for example dividing by zero. These run-time errors, as opposed to compile-time errors, are another source, or rather sink, of development time.

The basic work cycle when writing computer programs is illustrated below:



For complex projects, the work cycle can be simplified by using an integrated development environment (IDE), such as Microsoft Visual Studio, or the Visual Basic Editor within Excel. The development environment contains a text editor, for writing programs; a compiler, for building the programs; and a whole range of debugging tools for tracking down problems during the compilation and/or execution of the program. For the uninitiated, however, IDEs can seem overwhelming with a huge range of options that are not required for simple projects.

## 1.2 The simplest C++ program

Every valid C++ program must “tell” the compiler where its set of instructions starts and finishes. Hence, every C++ program must contain a **function**, see §3, that contains **every** instruction to be performed by the program. The required function is called `main()` and the simplest C++ program is shown below:

```
int main() {}
```

The program is very boring because the function contains no instructions, but it will compile and it will run. The keyword `int` indicates that the function `main()` will return

an integer (a whole number) when it has completed all its instructions. In C++, sets of instructions are grouped together by braces, sometimes called curly brackets, `{}`; everything between the braces following `main()` will be executed while the program is running. The round brackets are used to specify arguments to functions, see §3. It is possible to write programs using only the main function, but to do so would fail to take advantage of the more powerful structural features of C++.

### 1.3 The C++ keywords

There are 32 keywords defined by the ANSI C standard, see Table 1. If you are programming in C, these are the only keywords that you have to remember and, in practice, the working vocabulary is about 20 words. The C++ standard defines an additional 32 keywords, shown in Table 2. C++ is a rapidly evolving language and the number of keywords may still change in the future. Case is important in C++ and keywords must be specified in lower case: e.g. `else` is a keyword, but `Else`, `ELSE` and `ELSe` are not.

<code>auto</code>	<code>double</code>	<code>int</code>	<code>struct</code>
<code>break</code>	<code>else</code>	<code>long</code>	<code>switch</code>
<code>case</code>	<code>enum</code>	<code>register</code>	<code>typedef</code>
<code>char</code>	<code>extern</code>	<code>return</code>	<code>union</code>
<code>const</code>	<code>float</code>	<code>short</code>	<code>unsigned</code>
<code>continue</code>	<code>for</code>	<code>signed</code>	<code>void</code>
<code>default</code>	<code>goto</code>	<code>sizeof</code>	<code>volatile</code>
<code>do</code>	<code>if</code>	<code>static</code>	<code>while</code>

Table 1: The 32 C keywords

### 1.4 Syntax of the source code

The key elements of C++ syntax are shown below:

- A semicolon is used to mark end of an instruction.
- Case is important, i.e. `sin` is not the same as `SIN`.
- Totally free form, lines and names can be as long as you like!
- Comments take the form `/* C style comment */` or `// C++ style comment`.

asm	export	overload	throw
bool	false	private	true
catch	friend	protected	try
class	inline	public	typeid
const_cast	mutable	reinterpret_cast	typename
delete	namespace	static_cast	using
dynamic_cast	new	template	virtual
explicit	operator	this	wchar_t

Table 2: The 32 additional C++ keywords

- Groups of instructions are surrounded by braces {}.

The most important point to remember is that all statements must end with a semicolon ;. Often, forgetting a semicolon can cause a huge number of compilation errors, particularly if the omission is near the start of a program.

## 1.5 Practicalities: compiling and running C++

Before it can be compiled, a C++ program must be saved as a file and, by convention, C++ programs are labelled by the filename extensions `.cc`, `.cpp` or `.C`. Large projects may, and probably should, be split into separate files, which can be compiled separately. The division of large projects into separate “compilation units” speeds up the re-compilation of the whole project if only one small part of the project has been changed. Keeping track of separate files and their interdependence is an area in which IDEs are extremely useful. The alternative is to use command-line compilation and keep track of all the different files yourself.

A command line is, as the name suggests, a place where you can issue (type) commands. Rather than clicking an icon to start a program, you must type the name of the program and then press return. Before windowing environments, all computation was performed using the command line and it is still easier, and quicker, to use the command line for compiling very simple programs.

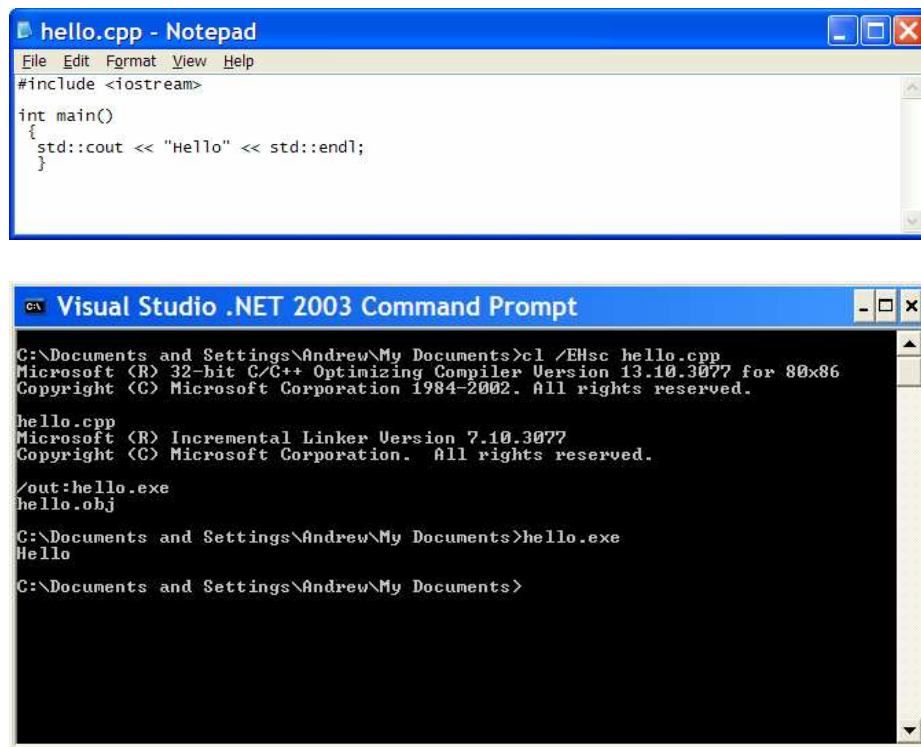
We consider both “simple” command-line compilation and compilation within an IDE (Visual Studio) for a simple program that merely prints the word `Hello` on the screen. The C++ instruction that performs this task is

```
std::cout << "Hello" << std::endl;
```

see §2.4.

### 1.5.1 Simple command-line compilation

The easiest way to write a small C++ program is to use a text editor, say Notepad, to generate a `.cpp` file that contains the required instructions and then to compile the program in a Command Prompt Window. The command for the Visual Studio C++ compiler is `cl /EHsc file.cpp`, where `file.cpp` is the file that contains your C++ program. The process is illustrated in Figure 1. The result of the compilation process is



```
hello.cpp - Notepad
File Edit Format View Help
#include <iostream>

int main()
{
    std::cout << "Hello" << std::endl;
}

Visual Studio .NET 2003 Command Prompt
C:\Documents and Settings\Andrew\My Documents>c1 /EHsc hello.cpp
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.

hello.cpp
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.

/out:hello.exe
hello.obj

C:\Documents and Settings\Andrew\My Documents>hello.exe
Hello

C:\Documents and Settings\Andrew\My Documents>
```

Figure 1: An illustration of simple command-line compilation. A Notepad window containing a simple C++ program, `hello.cpp`, that will print the word Hello on the screen; and a Visual Studio Command Prompt showing the compilation of the program `hello.cpp` and its execution.



an executable file `hello.exe`. In order to run the program one can either double click on the `hello.exe` program from the File Manager, or type `hello.exe` (and then return) into the command prompt, see Figure 1.

This method of compilation will work for all simple projects, that is all projects in this course. The method can be made to work for more complex projects, but it becomes more and more difficult to do so.

### 1.5.2 Visual Studio Compilation

A better way of managing large projects is to use Visual Studio. Initially, this will seem more complicated than the simple method outlined above, but it does not become significantly more complex as the projects become larger. Once Visual Studio has been started, it is used to write, compile and run the program. One major advantage of Visual Studio is that it automatically includes a number of **libraries** that are used to help your program interact with the Windows operating system. A library is a collection of functions that have already been written and compiled. Precisely which libraries should be included depends on the type of program that you are writing. If the program will use graphics or interact with the mouse it requires more libraries than a simple command line application. You can specify which libraries are included by choosing your project type when creating a new project. During this course, we shall consider only (Win 32) Console Applications and Win32 Projects.

The first time that you run Visual Studio you may be asked to select a default environment, in which case you should select “Visual C++ Development Settings”. In Visual Studio 2003 (and earlier), the Start Page of Visual Studio contains a New Project button, which when pressed brings up the New Project dialog box, see Figure 2. In Visual Studio 2005, the same dialog box is accessed by clicking the Project: Create link in the Recent Projects window on the Start Page.

For simple C++ programs, the project should be a Visual C++ Console Application, also called a Win32 Console Application, located in the Win32 submenu of the Visual C++ project templates. Filling in a name, *e. g.* `Hello`, for the project in the dialog box and double-clicking on the Console Application icon from the Templates window (or clicking Open in the dialog box after selecting the Console Application icon) will bring up the Application Wizard. For the default settings, simply press Finish and Visual Studio will then create a number of files, which should appear in the Solution Explorer window on the side of the screen. For information you can double-click on the `ReadMe.txt` file. Double-clicking on `Hello.cpp` (the actual C++ program) brings up a short section of

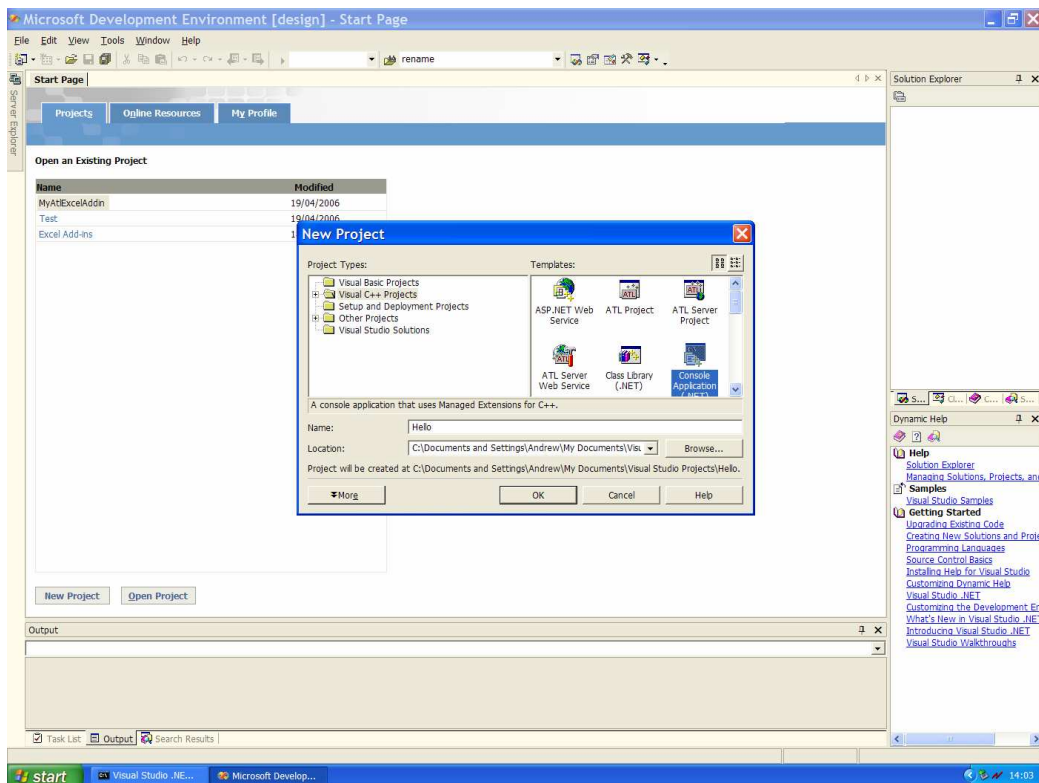


Figure 2: Visual Studio Start Page and New Project dialog. A Console Application called Hello is about to be created.

C++ code, see Figure 3. In order to write our own program, we comment out the pre-created instruction in the function `int _tmain()` and add our instruction. The program may then be compiled by going to the Build menu and choosing the Compile option. Alternatively, we can choose to run our program by selecting Start from the Debug menu. If the program has been modified without compiling it a dialog box pops up, see Figure 4, pressing yes causes the program to be compiled and then run. Thus, the entire write, compile and run process takes place within Visual Studio.

A complete description of the features of Visual Studio is well beyond the scope of this introductory course. A vast amount of information is provided in the on-line help documentation.

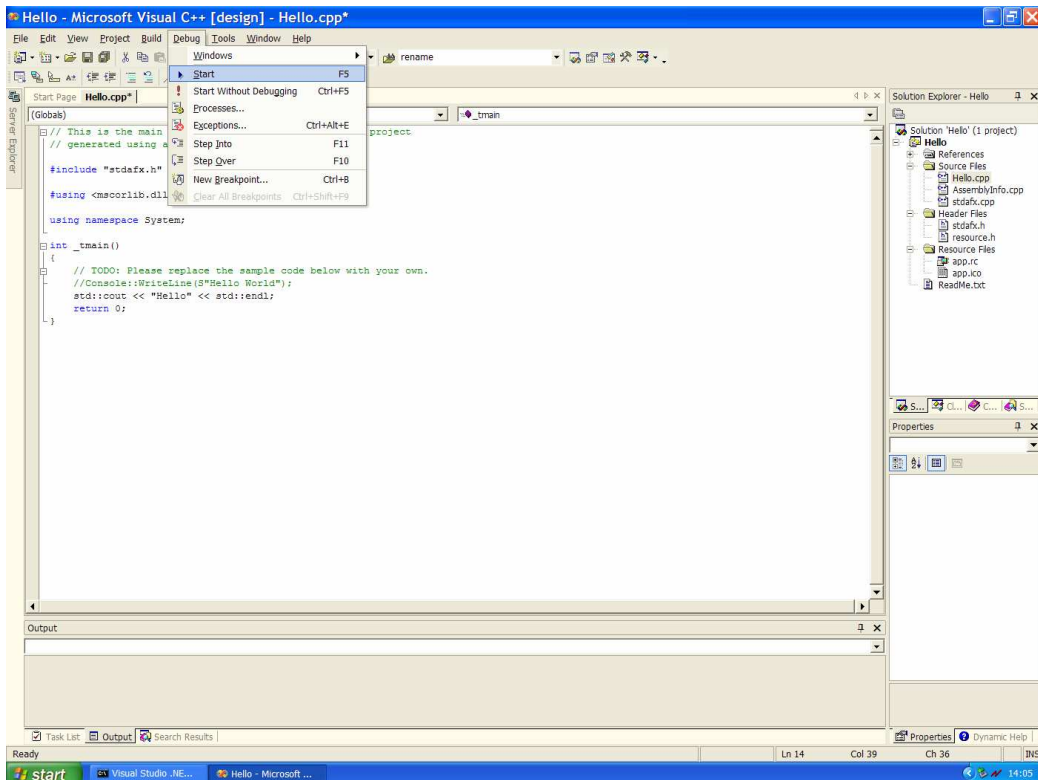


Figure 3: A Visual C++ Console Application, Hello, and the main source code Hello.cpp. The Run command is about to be issued.

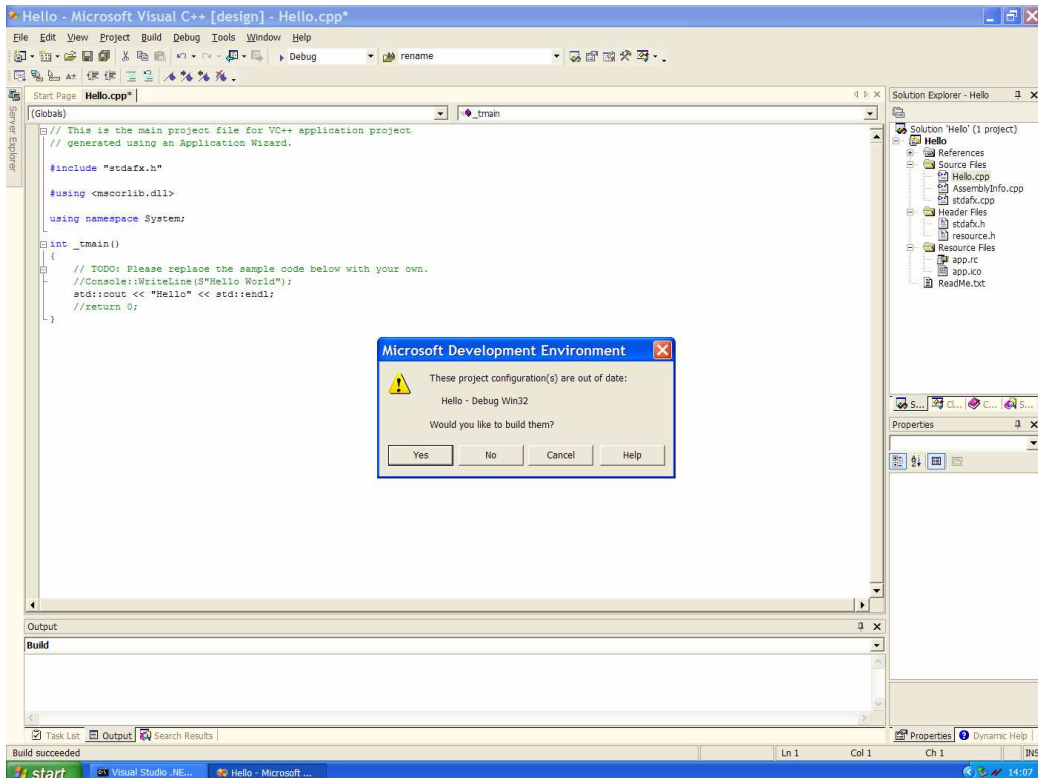


Figure 4: Visual Studio dialog box that appears when trying to run a program without re-compilation, clicking on **Yes** will cause the program to be compiled and then executed.

## 2 Getting started with C++

### 2.1 Data types

Almost all computational tasks require the manipulation of data, e.g. finding the maximum of a set of numbers, printing an e-mail, modelling the growth of the stock market, etc. In C++, the particular type of each datum must be declared before it can be used. It is possible to create your own “custom” data types, but for simple tasks the seven built-in data types (**char**, **wchar\_t**, **bool**, **int**, **float**, **double** and **void**) are sufficient. Table 3 shows the type of data represented by the first six keywords.

<b>char</b>	a single character (letter)
<b>wchar_t</b>	a wide character (letter) (used for characters from non-english alphabets)
<b>bool</b>	a boolean datum (true or false)
<b>int</b>	an integer datum (whole number)
<b>float</b>	a floating-point datum (a real number with six digits of precision)
<b>double</b>	a double-precision, floating-point datum (a real number with ten digits of precision)

Table 3: The (non-void) C++ built-in data types.

The **void** data type is rather special and is used to declare a function that does not return a value (a subroutine), see §3.

### 2.2 Variables

A *variable* is a named location in memory, used to store something that may be modified by the program. All variables must be declared before they are used. You can call variables anything you like, provided that the name you choose does not start with a number, is not a C++ keyword and is not the same as any other variable within the same scope, see §2.2.1. It is usually a good idea to use descriptive names that indicate what the variable represents. Example declarations are shown below:

```
int i,j,k;
double a,b;
char initial;
```

Note that commas may be used to separate variables of the same type and that each line ends with a semicolon. The initial value of a variable may be assigned when the variable is declared. Uninitialised variables can be a source of mysterious errors, so it's a good idea to initialise your variables if practical:

```
int i=0,j=1,k=2;
double a=12.35648;
char ch='a';
```

Note that characters must be enclosed in single quotes ' '.

### 2.2.1 The scope of variables

C++ permits the declaration of variables anywhere within your program, but a variable can only be used between the braces {} within which it is declared. The region in which the variable can be used is known as its **scope**. An advantage of local scopes is that you can use the same name for variables within different regions of the same function. The following two example codes illustrate these ideas.

<pre>int main() { //Declare a in new scope {int a = 10;} //Declare a in new scope (ok) {int a = 15;} //a used outside scope (illegal) a = 20; }</pre>	<pre>int main() { //Declare a in current scope int a=10; //Re-declare a in scope (illegal) int a = 15; //a used in scope (ok) a = 20; }</pre>
---	---

Neither of the two codes will compile. The code on the left fails on the last line of `main` with the error `'a' : undeclared identifier` because `a` has not been declared in scope. The code on the right fails on the fourth line of `main` with the error `'a' : redefinition; multiple initialization` because the variable `a` cannot be declared twice in the same scope.

### 2.2.2 Arrays

An *array* is a just like an array in maths: a collection of variables of the same type, called by the same name, but each with a different index. The standard way to define an array is to enclose the dimension in square brackets [] after the variable name.

```
int x[100]; // 100 integer array
double y[300]; // Array of 300 doubles
```

Arrays are initialised by using braces to define the array and commas to separate the individual array entries:

```
double x[3] = {1.0, 2.5, 3.7};
```

The individual entries of an array are accessed by adding a single number in square brackets after the variable name. The array `double x[3]` has the entries `x[0]` (= 1.0), `x[1]` (= 2.5) and `x[2]` (= 3.7). **Important note:** In C++, array indices start from 0.

Multidimensional arrays are declared and initialised in an obvious way:

```
double matrix[10][10];
int 3dtensor[3][3][3] = {{1,2,3},{4,5,6},{7,8,9}};
```

The maximum number of dimensions, if any, is determined by the compiler. Note that arrays defined in this way are allocated in the stack, an area of memory designed to store variables with short lifetimes. The size of the stack is set by the operating system. An area of memory that is designed to store longer-lived variables is known as the heap, but can only be accessed via dynamic allocation.

### 2.2.3 Dynamic memory allocation and pointers

In many cases, the size of the array will not be known when writing the program. For example, if we are reading in data from a file then the total number of data will depend on the size of the file. In these situations there are two options:

- Static Allocation: Allocate a fixed, but large, amount of storage, `double A[1000]`; (possibly limited by stack size).
- Dynamic Allocation: Determine the exact amount of storage required as part of the program.

In C++ dynamic memory allocation is handled by pointers, which are declared using the `*` operator.

```
double *A_dynamic;
```

We have declared the variable `A_dynamic` as a **pointer** to a `double`, or an address in memory. Declaring a pointer does **not** actually allocate any storage for a `double` variable, it merely “points” to an area of memory that **can** be used to store `double` variables.<sup>2</sup> The use of pointers for dynamic memory allocation may seem rather convoluted, but pointers have many other uses. In particular, they are an essential part of the interface between C++ and Excel — the only way that we can know where Excel has stored a particular variable is by its memory address, *i.e.* a pointer, see §5.

Having declared a pointer, we can use it to allocate storage for as many `double` data as required. The instruction

```
A_dynamic = new double[100];
```

allocates an array of 100 `double` variables and the pointer refers to the first of these variables. We use the standard array syntax to access the variables, `A_dynamic[0]` is the first entry in the array, etc. Of course, if we knew that we needed storage for 100 data we could have defined the array statically.

The following program assigns storage for a number of `int` variables read in from a file, for details on file I/O see §2.4.2.

```
#include<iostream>
#include<fstream>
using namespace std;

int main()
{
    //Declare and null a pointer to integer data
    int *A_dynamic=0;

    cout << "Reading in data from file" << endl;
    ifstream in_file("input.dat");

    //The number of data must be the first entry in the file
    int num_data=0;
    in_file >> num_data;
```

---

<sup>2</sup>The introduction of a pointer does introduce an additional memory overhead because we have to store the pointer itself. A bad choice of data structure with multiple pointers can be very wasteful.



```
//Allocate storage
A_dynamic = new int[num_data];

//Loop over the number of data and read from file
for(unsigned i=0;i<num_data;i++)
{
    in_file >> A_dynamic[i];
}

//Close the file
in_file.close();

//Calculate the sum of all the data
double sum=0.0;
for(unsigned i=0;i<num_data;i++) {sum += A_dynamic[i];}

//Print out the average value
cout << "Average value " << sum/num_data << endl;

//Free the memory allocated
delete[] A_dynamic; A_dynamic=0;
}
```

If the file `input.dat` contains the single line

```
10 1 2 3 4 5 6 7 8 9 10
```

the output from the program is

```
Average value 5.5
```

Once we have allocated memory for variables using the `new` keyword, then we are in charge of cleaning up when we have finished with it. The keyword `delete` is used to deallocate memory. If an array has been allocated, square brackets must be added after the delete command, e.g. `delete[] array_data`.

If memory has been deleted then it can be reused by any other programs running on the computer. If it is accidentally used by your program again, without reallocation, the result is a nasty run-time error known as a **segmentation fault**. These are particularly

hard to track down because if the memory has not yet been grabbed by another program then everything will appear to be fine. For this reason, it is very good practice to “null out” unused pointers. When a pointer is declared, initialise it to null, 0, and when the allocated memory has been deleted reset the value of the pointer to zero.

## 2.3 Manipulating Data — Operators

We can now create and initialise variables, but how do we modify the data? The answer is to use operators, which, as the name suggests, operate on the data. The built-in operators in C++ may be broadly subdivided into four classes *arithmetic*, *relational*, *logical* and *bitwise*.

You have already been introduced to the assignment operator, =. The most general form of assignment is

```
variable = expression;
```

Note that C++ will convert between data types in assignments. The code shown below will compile without complaint.

```
int i=1;
char ch;
double d;

ch = i; // Conversion from int to char
d = ch; // Conversion from char to double
```

The compiler will usually do the right thing, for example converting from double to integer should give the integer part of the result. That said, it is unsafe to rely on automatic conversion, and doing so can give rise to funny errors. Be sure to check data types in expressions carefully.

### 2.3.1 Arithmetic Operators

Table 4 lists the C arithmetic operators, which are pretty obvious apart from the last three, %, -- and ++. The modulus operator, %, gives the remainder of integer division, it cannot be used on floating point data types. The increment and decrement operators may seem strange at first: ++ adds 1 to its operand and -- subtracts 1. In other words,  $x = x + 1$ ; is the same as ++x; and  $x = x - 1$ ; is the same as x--;. Increment and

-	Subtraction
+	Addition
*	Multiplication
/	Division
%	Modulus
--	Decrement
++	Increment

Table 4: The C++ arithmetic operators

decrement operators can precede or follow the operand and there is only a difference between the two when used in expressions.

```
x = 10;
y = ++x; // Increments x and then assigns y; i.e. x = y = 11
```

```
x = 10;
y = x++; // Assigns y and then increments x; i.e. x = 11, y = 10
```

Arithmetic operators are applied in the following order:

<b>First</b>	++ --
	- (unary minus)
	* / %
<b>Last</b>	+ -

Parentheses () may be used to alter the order of evaluation by forcing earlier evaluation of enclosed elements. i. e.  $2 + 3 * 5 = 17$ , but  $(2+3) * 5 = 25$ .

### 2.3.2 Relational and logical operators

Relational and logical operators rely on concepts of false and true, which are represented by integers in C++. False is zero and true is any value other than zero. Every expression involving relational and logical operators returns 0 for false and 1 for true. Table 5 shows all the relational and logical operators used in C++.

Examples of relational operations and their values are

```
10 > 5 // True (1)
```

Relational operators	
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
==	Equal
!=	Not equal
Logical operators	
&&	AND
	OR
!	NOT

Table 5: The C++ relational and logical operators

```
19 < 5 // False (0)
(10 > 5 ) && (10 < 20) // True (1)
10 < 5 && !(10 < 9) || 3 <= 4 // True (1)
```

These operators are usually used to control the flow of a program and will be further explored in section 2.5.

### 2.3.3 Shorthand operators

C++ uses a convenient (well some would, and do, say obscure) shorthand to replace statements of the form `x = x + 1`,

```
x += 10; /* is the same as */ x = x + 10;
x -= b; /* is the same as */ x = x - b;
```

This shorthand works for all operators that require two operands and is often used in professional C++ programs, so it is well worth taking the time to become familiar with it. In fact, shorthand operators are slightly more efficient because they avoid the need to create temporary variables.

## 2.4 Talking to the world — Input and Output

No matter how wonderful your program is in isolation, at some point it'll need to interact with the outside world. I/O or input and output is not part of the standard C++

keywords. The appropriate functions are present in libraries that can be incorporated into any C++ program. The necessary library is called `iostream` and is included using the compiler directive `#include<iostream>`. In fact, Visual Studio automatically includes the appropriate libraries for you when it creates the Console Application template.

An example of simple I/O is shown below

```
#include <iostream>

int main()
{

    int i; //Declare an integer variable

    std::cout << "This is output.\n"; //print a string

    std::cout << "Enter a number: ";
    std::cin >> i;                //read in a number

    // output the number and its square
    std::cout << i << " squared is " << i*i << "\n";
}
```

`std::cout` is a *stream* that corresponds to the screen and `std::cin` is a stream that corresponds to the keyboard. The `<<` and `>>` operators are output and input operators and are used to send output to the screen and take input from the keyboard. Several output operators can be strung together in the same command. The special character `'\n'` represents a newline. This program reads into a variable, `i`, that has been declared to be an integer and what you type at the keyboard is automatically converted into an integer. If the input data is not of the correct type very strange things can happen, see below.

```
%. /a.out
This is output
Enter a number 5
5 squared is 25
```

```
% ./a.out
```

```
This is output
Enter a number 1.45
1 squared is 1
```

```
% ./a.out
This is output
Enter a number a
1 squared is 1
```

### 2.4.1 A note on namespaces

You may be wondering about the meaning of the prefix `std::`. The answer is that `std` is a C++ namespace. When writing large projects it is almost impossible to think of unique names and it's easy to accidentally call two distinct things by the same name. In an attempt to prevent this, C++ allows the grouping of functions and data into named collections — namespaces. All functions and classes defined in the standard libraries are in the standard namespace, `std`. In order to tell the compiler that we want to use `cout` from the namespace `std` the syntax is `std::cout`. An alternative is to place the statement `using namespace std;` at the top of our program, see later examples, in which case the standard namespace will be searched for any unknown names and we do not need to use the `std::` prefix at all.

### 2.4.2 File I/O

File I/O is very similar to the I/O system described above and is also based upon streams. An extra header file must be included for file I/O, `#include <fstream>`. An input stream uses the `ifstream` class and an output stream uses the `ofstream` class. The following program writes a simple text file that lists the integers 1 to 10 in one column and their squares in the next column.

```
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;
    ofstream out("test.out"); // Open an output file called test.out
```

```
                                // The output stream is called out
//Error check:
//If the file has not been opened then out will be zero
if(out==0)
{
    cout << "Cannot open file test.out\n";
    throw; // Throw an exception (end program)
}

//A loop in which i takes the integer values 1 to 10
for(int i=1;i<=10;i++)
{
    //Print the value of i and i squared to the output file
    out << i << " " << i*i << endl;
}

//Close the output file
out.close();
}
```

We test whether the file has actually be opened by using an `if` statement, see §2.5. If the file has not been opened, the error is reported by using the keyword `throw`, which immediately exits the `main` function and returns control to the operating system. The object `endl` is an output stream modifier that outputs a newline and flushes the stream, i.e. writes everything to disk.

In fact, opening a file does not have to be done when the stream is declared. The command `out.open("test.out");` can be used anywhere in the body of the function.

## 2.5 Conditional evaluation of code

In many tasks, the next instruction will depend on the results of a previous calculation and this means that the actual instructions that are performed can vary when the program is executed at different times. In the previous example, the program will print an error and stop if the file cannot be opened, which could occur because the disk is full.

### 2.5.1 if

C++ supports two main conditional constructs: **if** and **switch**. The general form of the **if** command is

```
if(expression) statement;
    else statement;
```

where a *statement* may be a single command, a block of commands, enclosed in braces {} or nothing at all; the **else** command is optional. If *expression* evaluates to true, anything other than 0, the statement following **if** is executed; otherwise the statement following **else** is executed, if it exists. Let's consider a concrete example:

```
#include <iostream>
using namespace std;

int main()
{
    float a;

    cout << "Please enter a number ";
    cin >> a;

    if(a > 0) cout << a << " is positive\n";
    else cout << a << " is not positive\n";
}
```

The above program takes a user-entered number and determines whether or not it is positive. If-else commands may be nested and this language feature can be used to add a zero test to the above program:

```
#include <iostream>
using namespace std;

int main()
{
    float a;

    cout << "Please enter a number ";
```



```
cin >> a;

if(a > 0) cout << a << " is positive\n";
else
    if(a == 0) cout << a << " is zero\n";
    else cout << a << " is not positive\n";
}
```

**Important note** The test for equality is the `==` relational operator. One of the most common programming mistakes is to use the assignment operator `=` instead of the relational operator `==`.

```
if(a=0) cout << a << " is zero\n";
```

The above code will compile, but it doesn't do what you think! The expression `a=0` assigns the value 0 to the variable `a` and returns true if successful. Thus, instead of testing for zero, the statement automatically sets `a` to zero.

### 2.5.2 The ? command

The `?` operator is a shorthand for simple if-else statements. The syntax is

$$\text{expression 1 ? expression 2 : expression 3}$$

and if *expression 1* is true, then *expression 2* is evaluated; if *expression 1* is false *expression 3* is evaluated.

```
x = 10;
y = x > 9 ? 100 : 200;
```

In the above example, `y` is assigned the value 100; if `x` had been less than 9, `y` would have the value 200. In if-else terms the same code would be

```
x = 10;
if(x > 9) y = 100;
else y = 200;
```

For clarity, it is generally better to avoid the `?` operator and write out the full if-else construction. Nevertheless, you might encounter it in programs written by others.

### 2.5.3 switch

The **switch** command is used to construct multiple-branch selections and tests the value of an expression against a list of integer or character constants. Unlike **if**, **switch** can only test for equality, but it can be useful in certain situations, such a menu operations.

```
#include <iostream>
using namespace std;

main()
{
    char ch;
    double x=5.0, y=10.0;

    cout << " 1. Print value of x\n";
    cout << " 2. Print value of y\n";
    cout << " 3. Print value of xy\n";

    cin >> ch;

    switch(ch)
    {
        case '1':
            cout << x << "\n";
            break;
        case '2':
            cout << y << "\n";
            break;
        case '3':
            cout << x*y << "\n";
            break;
        default:
            cout << "Unrecognised option\n";
            break;
    } \\End of switch statement
} \\End of main function
```

The `case` keyword is used to demark individual tests and the `break` keyword breaks out of the `switch` command at the end of each test segment. The optional `default` block corresponds to the final `else`, it executes if none of the tests are true. If the `break` command is omitted then the next command(s) in the `switch` statement will execute. This can be used to produce the same behaviour for many tests. *i.e.*

```
switch(ch)
{
  case '1':
  case '2':
  case '3':
      cout << "1, 2 or 3 pressed\n";
      break;
}
```

## 2.6 Iterative execution of commands — Loops

Almost all programming tasks require the repeated execution of the same, or very similar, commands. Instead of writing the same, or almost, the same instruction several times, it would be much better to write the instruction once and tell the computer to perform it several times, such a construct is known as a loop.

### 2.6.1 for loops

The most versatile loop construct in C++ is the `for` loop, which repeats a block of code a number of times until a *predefined* condition is satisfied. The generic form of this construct is

$$\text{for}(\textit{initialisation}; \textit{condition}; \textit{increment}) \textit{statement};$$

where the *initialisation* command is executed at the start of the first loop. The *condition* is tested at the top of each loop and if it is true then the loop commands are executed. The *increment* command is executed at the end of each loop.

```
#include <iostream>
using namespace std;
```

```
int main()
{
```

```
    for(int i=1; i <= 10; i++) cout << i << i*i << "\n";  
}
```

The above example prints the numbers 1 to 10 and their squares. In this case the increment operator ++ has been used as the *increment* command. The comma operator allows multiple variables to control for loops:

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int i,j;  
  
    for(i=1, j=20; i < j; i++, j-=5)  
    {  
        cout << i << " " << j << "\n";  
    }  
}
```

produces the output

```
1 20  
2 15  
3 10  
4 5
```

The components of a for loop may be any valid C++ commands, or they may be absent, allowing powerful generalisations. For example

```
for(x=0; x!=100; ) cin >> x;
```

will run until the user enters 100.

```
for(;;) cout << "An infinite loop\n";
```

A for loop with no components is an infinite loop, it can only be terminated by using a **break** command somewhere in the loop. Time delay loops may be constructed by omitting the body of a for loop e. g.

```
for(t=0; t < 100; t++); // wait for a while
```

### 2.6.2 while and do-while loops

Another type of loop is the **while** loop, which has the functional form

```
while(condition) statement;
```

and continues to iterate until the *condition* becomes true. The **while** loop tests the *condition* at the top of the loop and so the *statement* will not execute if the *condition* is initially false. A common use of while loops is for convergence tests

```
#include <iostream>
using namespace std;

main()
{
    double num=50;

    while(num > 0.0001)
    {
        num /= 2.0;
    }

    cout << num << "\n";
}
```

This loop repeatedly halves the variable `num` until it is less than a specified tolerance.

A **do-while** loop is similar to a while loop, but performs the test at the bottom of the loop rather than the top. It has the form

```
do{statement;} while(condition);
```

The do loop above could equally well have been a do-while loop:

```
#include <iostream>
using namespace std;

main()
{
    double num=50;
```

```
do
{
    num /= 2.0;
}
while(num > 0.0001);

cout << num << "\n";
}
```

Remember, a do-while loop always forces at least one execution of the commands in the body of the loop, whereas a while loop does not.

## 2.7 Jump commands

In certain situations, you might want to jump to a new point in the program skipping over the intermediate instructions. The most common jump command is **return** which is used to return from functions to the main program, see §3. The **break** command has already been mentioned when describing the **switch** construct. It has a more general scope, however, and can be used to force the immediate termination of *any* loop. For example

```
for(int i=0; i <= 10; i++)
{
    cout << i;
    if(i==5) break;
}
```

will only display the numbers 1 to 5 on the screen because **break** overrides the conditional test `i <= 10` in the for loop.

The “opposite” of the **break** command is **continue**, which forces another iteration of the loop, skipping any intermediate code. In a for loop, the increment command is performed and then the conditional test. In while and do-while loops control passes directly to the conditional tests. This can be used to force the earlier evaluation of the condition.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    bool flag=false;
    char ch;

    while(!flag)
    {
        cin >> ch; //read in a character

        if(ch=='x')
        {
            flag = true;
            continue;
        }

        cout << "You have entered the character " << ch << "\n";
    }
}
```

The preceding program takes input from the keyboard and prints the individual characters entered. If the user enters an x the loop terminates without printing that character. This program also illustrates another common C++ programming practice, using boolean variables as test variables. If flag is false (0), then not flag (`!flag`) is 1 which is true and the while loop executes. If flag is set to true (1), not flag will be zero, which is false and the loop terminates. If you understand this program, then you should have no problem with relational expressions in C++.

### 3 Functions

Functions are the building blocks of C++ programs and, in their simplest form, are merely groups of instructions. In general, however, a function can take a number of variables (arguments), operate upon them and return a value to the part of the program from which the function was called. You should already be familiar with the most important function `main()`. The most general form of a function definition is shown below

```
return data type function_name( arguments )  
{ body of function }
```

The *arguments* to a function take the form of a comma-separated list of type definitions *i.e.* (`double a, int b, ...`). If the function does not return a value (a subroutine), the return data should be of the type `void`.

Consider now a function `square()` that returns the square of its argument:

```
int square(int a)  
{  
    return(a*a);  
}
```

The function “expects” an integer variable as its argument and returns an integer variable. The `return` keyword causes the function to return to the point in the program from which it was called and sets the return value, if there is one.

In a well-written program tasks that are performed many times should be “packaged away” in functions and it is the functions that are then called repeatedly. The advantage of this approach is that once the function has been written and tested it can be trusted not to be the source of any errors in the program, which facilitates debugging. In addition, if a more efficient method of achieving the same result is found, the function can be changed in one place and the rest of the program will not have to be altered.

The following example shows how to use functions in a C++ program.

```
#include <iostream>  
using namespace std;  
  
//Declare a function that multiplies two integers  
int product(int a, int b)
```



```

    {
        return(a*b);
    }

//main body of program
int main()
{
    int i,j; //Declare two integers

    //Loop i varies from 1 to 10, and j varies from 10 to 1
    for(i=1, j=10; i<=10; i++, j--)
    {
        cout << i << " " << j << " " << product(i,j) << "\n";
    }
}

```

Note that the choice of variable names used within the `product()` function is totally independent of any other variables in the program. It is a common misconception that if the arguments of the function are `a` and `b` then the function must be called using the variable names `a` and `b`.

### 3.1 Where to define functions

In C++, nested functions are not allowed, which means that functions cannot be defined within other functions. In particular, all functions must be defined outside the `main()` function. For example, the following is illegal code.

```

int main()
{
    //Nested function --- illegal
    double square(double x){return x*x;}

    square(2.0);
}

```

A legal version of the above code would be

```

//Function definition outside main -- legal

```

```
double square(double x) {return x*x;}

int main()
{
    square(2.0);
}
```

In addition, functions must be defined before they can be used. The following is again illegal code:

```
int main()
{
    //Function called before definition
    square(2.0);
}

double square(double x) {return x*x;}
```

Actually, we do not have to define the entire function before it is called, we can specify the function prototype — its name, arguments and return type — and delay specification of the function body. The function prototype is defined in the same way as a function but with a semi-colon placed after the closing parenthesis, see below.

```
//Function prototype (establish the interface)
double square(double x); //Note the semi-colon here

int main()
{
    //Call function (interface defined so OK)
    square(2.0);
}

//Now define body of square function, as usual
double square(double x) {return x*x;}
```

### 3.2 Function libraries

The core C++ language is very compact, but much of the power of the language rests in the vast number of existing libraries, which contain many thousands of specialised functions. We have already used the I/O libraries, but there are many, many others. Another commonly used library is the math library, `cmath`. The standard trigonometric, hyperbolic and exponential functions are all present: e. g. `sin()`, `exp()`, `acos()`, `log()`, `tanh()`. Also of use are the function `pow(x,y)` which raises the number `x` to the power `y` and `sqrt()`, the square-root function.

```
#include <iostream>
#include <cmath>
using namespace std;

//Define our own cube root function
double cbrt(double arg)
{
    double result;
    result = pow(arg,(1.0/3.0));
    return(result);
}

int main()
{
    int i;
    double x;

    // Loop over the numbers 1 to 10
    for(i=1;i<=10;i++)
    {
        x = i*i*i; // This will transform the integer result to a double
        cout << x << " has the cube root " << cbrt(x) << "\n";
    }
}
```

This example also illustrates the use of local variables in functions. The variable `result` is only defined within the function `cbrt()`, it cannot be accessed from `main()`

and is a form of *data encapsulation*.

### 3.3 Modifying the data in function arguments

The functions defined thus far **cannot** change the value of the arguments passed to them. Consider the following example

```
#include<iostream>
using namespace std;

void square(int x) { x = x*x; }

int main()
{
    int a=5;
    square(a);
    cout << a << endl;
}
```

The result of running this program is that the number 5 will be printed on the screen. The variable `a`, defined in `main`, has not been changed by the function `square`. This is because the local variable `x` is a **copy** of the value stored in the variable `a`. The value of the copy is changed within `square`, but the original remains unchanged. The behaviour of the function is known as **call by copy** and is the “default” for C++ functions.

If we want the function to be able to modify the value of its arguments we must change this behaviour as shown below

```
#include<iostream>
using namespace std;

void new_square(int &x){ x = x*x; }

int main()
{
    int a=5;
    new_square(a);
    cout << a << endl;
}
```

```
}
```

Now, the number 25 will be printed on the screen. The **only** difference between the two functions `square` and `new_square` is the introduction of an ampersand, `&`, in the definition of the arguments to `new_square`. This is an instruction to the C++ to change the default behaviour and to use **call by reference** rather than **call by copy**. Now, the original variable, rather than a copy of its value, is directly accessed by the function.

### 3.4 Function overloading

The function `square` was written assuming that the data was `int`, but what happens when we call it with `double` data.

```
#include<iostream>
using namespace std;

int square(int x) {return x*x;}

int main()
{
    double a=1.5;
    cout << square(a) << endl;
}
```

The answer is that the `double` passed to the function will be converted into an `integer` and the program displays the number 1. Most compilers will issue a warning when compiling the above code.

We must write a new function that calculates the square of a `double` datum.

```
double square(double arg) {return (arg * arg);}
```

In older languages, such as C and FORTRAN, it's impossible to have two functions with the same name. In C++, you can write many functions with the same name, a property known as *function overloading*. To avoid confusion, functions with the same name should serve the same purpose, usually acting on different types of variables or classes. The only limitation is that C++ uses the arguments to the function to decide which version of the

function to call. Thus, it is impossible to have two functions with the same name and arguments, but different return types.

```
int solve_pde(double a);
float solve_pde(double a); // Illegal
int solve_pde(double a, int b) // OK
```

The compiler will determine which version of the function to use when it is called. The following code is illegal in C, but will compile and run in C++.

```
#include <iostream>
using namespace std;

int square(int arg)
{
    cout << "Integer square() called " << endl;
    return (arg * arg);
}

double square(double arg)
{
    cout << "Double square() called " << endl;
    return (arg * arg);
}

main()
{
    int i=10;
    double a=20;

    cout << i << " squared is " << square(i) << "\n";
    cout << a << " squared is " << square(a) << "\n";
}
```

## 4 Objects and Classes

### 4.1 Object-oriented programming

The programming that we have done so far has been procedural — everything is done by functions that are ultimately called from within the `main` function. This use of functions hides details of the information and instructions needed to perform specific tasks from the rest of the program. Such compartmentalisation of code and data allows easy upgrades to the program and also makes it easier for many programmers to work on a large project.

Object-oriented programming develops these ideas further and imposes a higher level of structure than procedural languages. In general, a problem is broken down into sub-problems organised in a hierarchical structure. Each subproblem can be translated into self-contained units called objects, which can contain variables **and** functions (also called methods) that operate upon those variables.

In their simplest form, objects can be regarded as “custom” data types. For example, a complex number is an object that has a real and an imaginary part. If we want to operate on the new complex number object then we must write new functions to do so.

Object-oriented programming introduces three main ideas:

*Encapsulation* binds code and data together into an object that cannot be influenced from outside sources except in very tightly controlled ways.

*Polymorphism* represents the concept of “one interface, multiple methods”. The same interface can be used to do different things for different objects: i.e. define `+` to add real numbers, and “custom made” complex numbers, matrices, etc.

*Inheritance* allows one object to acquire the properties of another. The new object has all the properties of the old and may add more of its own. An example would be to define a generic `Option` object and the derived objects `EuropeanOption`, `AmericanOption`, `AsianOption`, etc, could all inherit from `Option`.

### 4.2 The C++ class

The C++ `class` keyword is used to define objects. A very simple example would be to create a bank account class that stores the name and balance of a customer.

```
#include <string>
using namespace std;
class Account
{
```

```

public:
    string Name;
    double Balance;
};

```

The syntax is straightforward, the `class` keyword followed by a name indicates that we are defining a new class. Inside the braces we define all the variables (member data) that make up our new class. A `string` is a C++ object that represents a collection of letters, *i.e.* words. The `public` keyword indicates that these data can be accessed (and potentially modified) from outside the class. Finally, we **must** finish the declaration by placing a semicolon after the closing brace `};`.

Once defined, the `Account` class can be used in the same way as any built-in data type. We define objects of the `Account` class as follows:

```

int main() "
{
    Account ac_001, ac_002;

    ac_001.Name = "John Smith"; ac_001.Balance = 0.0;
    ac_002.Name = "Jane Jones"; ac_002.Balance = 10.0;
}

```

The two objects (variables) `ac_001`, `ac_002` are both instantiations of the `Account` class. Note that the variables within the class are accessed by using the dot operator, but that we must use the name of the specific object, not the class name, `Account.Name` is an error. If the `public` keyword had not been included in our class definition, the above code would not compile; it fails with the error `'Account::Name' cannot access private member declared in class 'Account'`.

The most general class definition is:

```

class class_name {
    private data and functions
    access-specifier:
    data and functions
    access-specifier:
    data and functions
}

```



```

        .
        .
        .
        access-specifier :
        data and functions
        } object list ;

```

The *object list* is optional, but if present it defines objects of the class. Once the class is declared, objects may also be defined anywhere in the code. The *access-specifier* is one of the three keywords **public**, **private** or **protected**. The default access type of classes is **private**, which means that the data and functions can only be used by objects of the same class. If the items are **public**, they are accessible by other parts of the program, as shown above. Finally, the **protected** keyword is essentially the same as **private**, but protected members may be used by any derived classes, §4.4. Note that access specification can be changed many times in a class declaration.

Member functions are functions that are defined within the class and can operate on all the class's member data. An example would be a `print_balance` function in our `Account` class.

```

#include <iostream>
#include <string>
using namespace std;

class Account
{
public:
    string Name;
    double Balance;

    //Print the balance to the screen (cout)
    void print_balance()
    {
        cout << Name << "'s account  has a balance of "
             << Balance << endl;
    }
};

```

```
int main()
{
    //Declare and initialise the account
    Account ac_001;
    ac_001.Name = "John Smith"; ac_001.Balance = 100.0;

    ac_001.print_balance();
}
```

Note that the member function is accessed in the same way as member data by using the dot operator.

Another obvious member function for an account class is a `deposit()` function.

```
#include <iostream>
#include <string>
using namespace std;

class Account
{
public:
    string Name;
    double Balance;

    //Print the balance to the screen (cout)
    void print_balance()
    {
        cout << Name << "'s account has a balance of "
             << Balance << endl;
    }

    //Deposit amount into the account
    //Function prototype, 'guts' must be defined later
    void deposit(const double &);
};

//Definition of the deposit function
```

```
void Account::deposit(const double &amount)
{
    Balance += amount;
}

int main()
{
    //Declare and initialise the account
    Account ac_001;
    ac_001.Name = "John Smith"; ac_001.Balance = 100.0;
    ac_001.deposit(50.0);

    ac_001.print_balance();
}
```

We can define the “guts” of member functions outside the class provided that we provide a function prototype within the class. A function prototype must specify the name of function, its return type and its arguments, but instead of the definition within braces, we finish the statement with a semicolon. The function body can be defined anywhere else in the program, but must be defined somewhere or the program will not compile. If defined outside a class declaration, a member function must include the class namespace, `ClassName::member_function()`, to distinguish it from member functions of other classes with the same name. If the arguments of the function are not the same as those of the prototype the program will not compile. The `const` keyword in the function argument indicates that the function is not allowed to change the argument `amount` even though it has been passed by reference.

Having written our `deposit` function, we can make `Balance` a private variable so that it can only be modified by member functions of the `Account` class. The following program should produce the same output as the program above, but it does not. Why?

```
#include <iostream>
#include <string>
using namespace std;

class Account
{
```

```
private:
double Balance;
public:
    string Name;

//Print the balance to the screen (cout)
void print_balance()
{
    cout << Name << "'s account has a balance of " << Balance << endl;
}

//Add amount to the balance (now declared in the class)
void deposit(const double &amount)
{
    Balance += amount;
}
};

int main()
{
//Declare and initialise the account
Account ac_001;
ac_001.Name = "John Smith"; ac_001.deposit(100.0);

ac_001.print_balance();
}
```

The answer is that because we have made `Balance` private, we cannot access it directly in the main program and it has not been initialised, which means that the result could be anything! The initialisation of private data can only be performed by a special member function — the constructor.

### 4.3 Constructors and Destructors

**Constructor** functions are called whenever an object is created and are used to initialise variables within the object. Similarly, **destructor** functions are called when an object

is destroyed and are used to clean up memory or close files that may have been opened by the object.

For any object, the constructor has the same name as the class and the destructor has the same name as the class, prepended by a tilde `~`. We now add a constructor and a destructor to our `Account` class.

```
#include <iostream>
#include <string>
using namespace std;

class Account
{
private:
double Balance;
public:
string Name;

//Constructor, initialise the Balance to zero
Account() {Balance=0.0;}
//Destructor, print closing Balance
~Account()
{
cout << "Closing account :";
print_balance();
}

//Print the balance to the screen (cout)
void print_balance()
{
cout << Name << "'s account has a balance of " << Balance << endl;
}

//Add amount to the balance
void Account::deposit(const double &amount)
{
Balance += amount;
}
```

```
    }  
};  
  
int main()  
{  
    //Declare and initialise the account  
    Account ac_001;  
    ac_001.Name = "John Smith"; ac_001.deposit(100.0);  
  
    ac_001.print_balance();  
}
```

The balance of a new account is initially set to zero and the balance will be printed when the `Account` object is destroyed (goes out of scope). It is possible to pass variables to a constructor and so we could also set an initial balance for a new account. It is **never** possible to pass arguments to a destructor.

```
Account::Account(const double &initial_balance)  
{Balance = initial_balance;}  
  
int main()  
{  
    //Set the opening balance of the account to 100  
    Account ac_001(100.0);  
  
    //Set the opening balance of the account to 50  
    Account ac_002 = 50.0;  
}
```

The second form of initialisation, normal assignment using `=`, only works if the constructor takes a single argument. Note that if the class constructor takes arguments, it is impossible to create an object without passing those arguments. We can use function overloading, if required, to provide a number of different constructors.

## 4.4 Inheritance

C++ allows the creation of new, sometimes called *derived*, classes from existing, or *base* classes. The idea is that the derived classes should be related to the base classes, perhaps they are more specialised versions of an abstract concept. Let us create a new type of `Account` called a `SavingsAccount` that pays interest on the balance.

```
class SavingsAccount : public Account
{
    private:
        double Interest_rate;
    public:

    //Default Constuctor, call Account's default constructor
    SavingsAccount() : Account()
    {
        //Set the interest rate
        Interest_rate = 0.05;
    }

    //Constructor with initial balance, call equivalent constructor
    //of Account
    SavingsAccount(const double &initial_balance) :
    Account(initial_balance)
    {
        //Set the interest rate
        Interest_rate = 0.05;
    }

    //Add interest to the account
    void add_interest()
    {
        Balance += Balance*Interest_rate;
    }
};
```

The new `SavingsAccount` has all the member features of the `Account` class, but adds new functionality of its own. The following simple program

```
int main()
{
    //Create a new savings account with initial balance of 100
    SavingsAccount sav_ac_1(100.0);
    sav_ac_1.Name = "John Smith";

    //Add interest to the account
    sav_ac_1.add_interest();
}
```

should produce the output

```
Closing account :John Smith's account  has a balance of 105
```

but instead it fails to compile and produces the error

```
'Account::Balance' cannot access private member declared
in class 'Account'
```

This is because we declared `Balance` to be `private` which means that it can only be used by member functions of the `Account` class. We could make `Balance` public, but then any function anywhere in the program could modify it. The solution is to change the `Account` class so that `Balance` is `protected`, meaning that it can be used by `Account` and any of its derived classes.

```
class Account:
{
    protected:
        double Balance;
};
```

The general syntax for creating derived classes is as follows

```
class class_name : access-specifier base_class_name {};
```



The *access-specifier* can be **public**, **private** or **protected**, just as inside class definitions. The principle of encapsulation cannot be violated, and so the access-specifier only applies to the public and protected members of the base class. Private members of the base class always remain private to that class. Protected members of the base class can be used by derived classes, but cannot be used outside them. Finally, public members of the base class remain public in the derived class, unless overruled by the access-specifier. This can all get quite confusing, but the essence is that data can only be made “more” private by using an access-specifier in front of a class — private data can never be made public.

#### 4.4.1 Overloading member functions

In the `SavingsAccount` class, we added extra member functions and data to the base `Account` class. We can also use function overloading to modify the functions defined in the base class. For example, let us create a new `account_type` function that returns a string describing the type of account and modify the `print_balance` function to use of this function.

```
#include <iostream>
using namespace std;

//Base Account class
class Account
{
    protected:
        double Balance;

    public:
        string Name;

        //Constructor that sets initial balance
        Account(const double &initial_balance)
        {Balance = initial_balance;}

        //Return the type of account as a string
        string account_type() {return "Basic Account";}

        //Print the balance to the screen (cout)
```

```
void print_balance()
{
    cout << Name << "'s " << account_type()
    << " has a balance of " << Balance << endl;
}
};

//A savings account inherits from the Account class
class SavingsAccount : public Account
{
private:
    double Interest_rate;

public:
    //Constructor calls Account's constructor
    SavingsAccount(const double &initial_balance) :
    Account(initial_balance)
    {
        //Set the interest rate
        Interest_rate = 0.05;
    }

    //Overload the account type
    string account_type() {return "Savings Account";}
};

int main()
{
    //Create a savings account
    SavingsAccount sav_01(100.0); sav_01.Name = "Fred Philips";
    //Create a standard account
    Account ac_01(50.0); ac_01.Name = "Fred Philips";
```

```
//Print the account names
cout << ac_01.account_type() << endl;
cout << sav_01.account_type() << endl;

//Print the balance of both accounts
ac_01.print_balance();
sav_01.print_balance();
}
```

The result of the program is the output

```
Basic Account
Savings Account
Fred Philips's Basic Account has a balance of 50
Fred Philips's Basic Account has a balance of 100
```

What has happened? The function `account_type` has been correctly overloaded when called directly, but not when it is called indirectly from within the member function `print_balance`. The problem is that `print_balance` is a member function of the base `Account` class so that when it is compiled it “does not know” that the function `account_type` will be overloaded in `SavingsAccount`. We can indicate to the compiler that the function might be overloaded by using the `virtual` keyword in the initial definition of the function.

```
virtual string account_type() {return "Basic Account";}
```

After this simple modification, the program produces the result

```
Basic Account
Savings Account
Fred Philips's Basic Account has a balance of 50
Fred Philips's Savings Account has a balance of 100
```

If you are writing member functions that are going to be overloaded you should nearly always make them `virtual`. It is also possible to define a “pure virtual” function; that is an interface for a function that **must** be implemented for every derived class. The syntax for a “pure virtual” function is

```
virtual string account_type()=0;
```

If the “pure virtual” `account_type()` function is not overloaded in the `SavingsAccount` class, the program will not compile.

#### 4.4.2 Multiple inheritance

A class may be derived from more than one base class, in which case the parent classes are separated by commas in the definition:

```
class derived: public base1, public base2 {};
```

One use of multiple inheritance is to “bolt together” functionality from different objects and it is possible to create very complex hierarchies of objects. One of the hardest parts of object-oriented programming is creating a simple, but complete, object model.

### 4.5 Pointers to objects

Pointers to objects can be declared in exactly the same way as any other data type, `Account* ac_pt` creates a pointer to an account object. Note that member functions and data must be accessed from an object pointer by using the arrow operator, `->`. An important feature of pointers to objects is that an object of **any** derived class can be assigned to a pointer to a base class. Provided that all overloaded functions have been defined as `virtual` functions the correct version of the function will be called.

```
int main()
{
    //An array of three pointers to accounts
    Account* ac_pt[3]={0,0,0};

    //Allocate a standard account to the first pointer
    ac_pt[0] = new Account(100.0); ac_pt[0]->Name = "John";

    //Allocate a savings account to the second pointer
    ac_pt[1] = new SavingsAccount(500.0); ac_pt[1]->Name = "Mary";

    //Loop over all accounts and print the balance
    for(unsigned i=0;i<3;i++)
    {
        //Check that an object has been allocated
        if(ac_pt[i] != 0)
        {
            ac_pt[i]->print_balance();
        }
    }
}
```

```

    }
}

//We should delete the objects here to be safe, but because it is
//the end of the program, they will go out of scope and be cleaned
//up anyway
}

```

The result of the program is

```

John's Basic Account has a balance of 100
Mary's Savings Account has a balance of 500

```

The Excel object model relies on the extensive use of pointers to objects. For example, if we have a pointer to a particular Excel application called `XL`, then we obtain a pointer to the Worksheet “Sheet1” in the active Workbook as follows:

```
XL->ActiveWorkbook->Worksheets->Item["Sheet1"];
```

Note that a pointer to a worksheet is itself an object defined in the `Excel` namespace.

## 4.6 The `this` pointer

Every C++ object contains a pointer to its own location in memory. This pointer is accessed using the `this` keyword in member functions of the class, e. g.

```

Account::print balance()
{
    cout << this->Name << "'s account has a balance of "
         << this->Balance << endl;
}

```

In most member functions, the `this` pointer is used implicitly by the compiler and it is not necessary to use the `this` keyword explicitly. An important exception is when working in classes derived from a templated base class. Nonetheless, the `this` pointer is essential in certain applications, for example, when a member function returns a pointer to the object itself.

## 5 Interfacing C++ and Excel

Communication between C++ and Excel is a complex topic and we shall cover only the very basics in these notes. There are two different (philosophical) choices:

- Call a C++ Add-in from Excel,
- Call Excel functions from within a C++ program.

The first option means that our C++ “backend” can be used from within Excel without the user having to know anything about C++, making it an attractive choice for applications developers. A complicating factor when writing such Add-Ins is that there are several different types, each with slightly different functionality.

The second option is more useful for programmers who would like to use the plotting and data analysis capabilities of Excel from within their stand-alone program; for example, displaying real-time graphs of simulation results.

### 5.1 Writing a simple Excel Add-In in C++

We shall consider the most simple type of Add-In, a dynamic link library (DLL). Furthermore, we restrict attention to functions that return a single `double` or `int` and take single (non-array) arguments. In order to use the functions contained in DLLs, we must write a little VBA wrapper for each function in our Excel worksheet. It is possible to avoid this step by converting our DLL into an XLL (an Excel-specific library), but this is beyond the scope of this course.

We now demonstrate how to write and call a function that takes a single `double` argument and returns a `double` value. For the purposes of illustration, we implement a function that simply returns the square of its argument.

#### 5.1.1 Creating the DLL in Visual Studio

The procedure for creating a DLL is similar to that for creating a standard C++ file. We create a Visual C++ Win32 Project (not a Console Application) called `MyXLLib`, although the name could be anything, of course. Instead of clicking `Finish` in the Application Wizard, however, we select the Application Settings option from the sidebar of the Wizard. We must select `DLL` as the Application Type and it is helpful to check the `Empty Project` button, see Figure 5. We add the (minimum) required two files to the Source Files directory, one C++ (`.cpp`) file and one module definitions (`.def`) file,

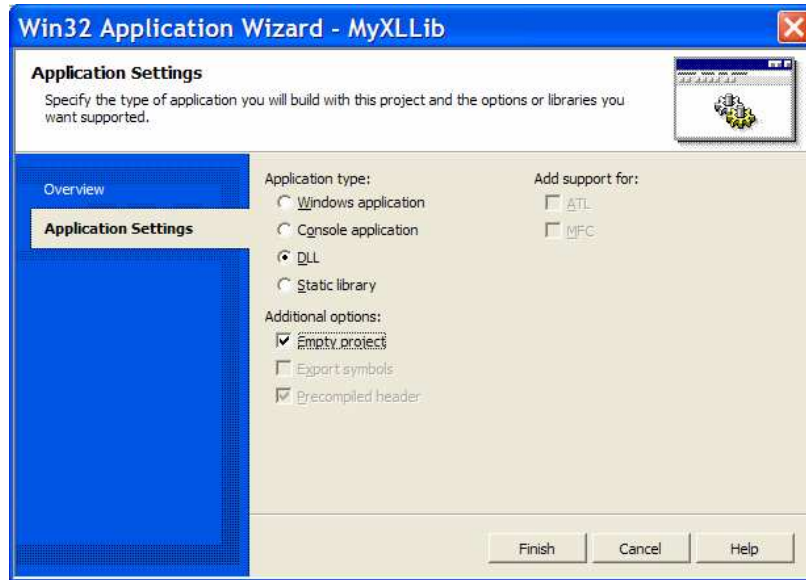


Figure 5: The Application Settings options in the Win32 Application Wizard. Note that the DLL and Empty Project options are selected.

by “right clicking” the Source File folder in the Solution Explorer, selecting Add and then Add New Item, see Figure 6. The Add New Item Window appears and we select a C++ file template, choose a name (Square in this case) and then click open (or add), see Figure 7. Repeating the procedure, but with the appropriate (.def) template, will create a Module Definition File (also called Square).

We can now write our C++ functions in the usual way in the .cpp file, but, because we are writing a library, we **do not** require a `main()` function; in fact, we **must not** include one. In our simple example, a function that returns the square of its argument, the contents of the C++ file is just

```
double __stdcall square_in_C(double &arg) {return (arg*arg);}
```

The argument to the function is passed *by reference*, the default method in VBA. Note also the additional instruction `__stdcall`, a Microsoft specific instruction that is required to make the compiled function compatible with Excel.

The Module Definition File should contain a list of the functions, defined in the .cpp file, that we wish to export — to be usable by Excel. In our example, the contents of the .def file is

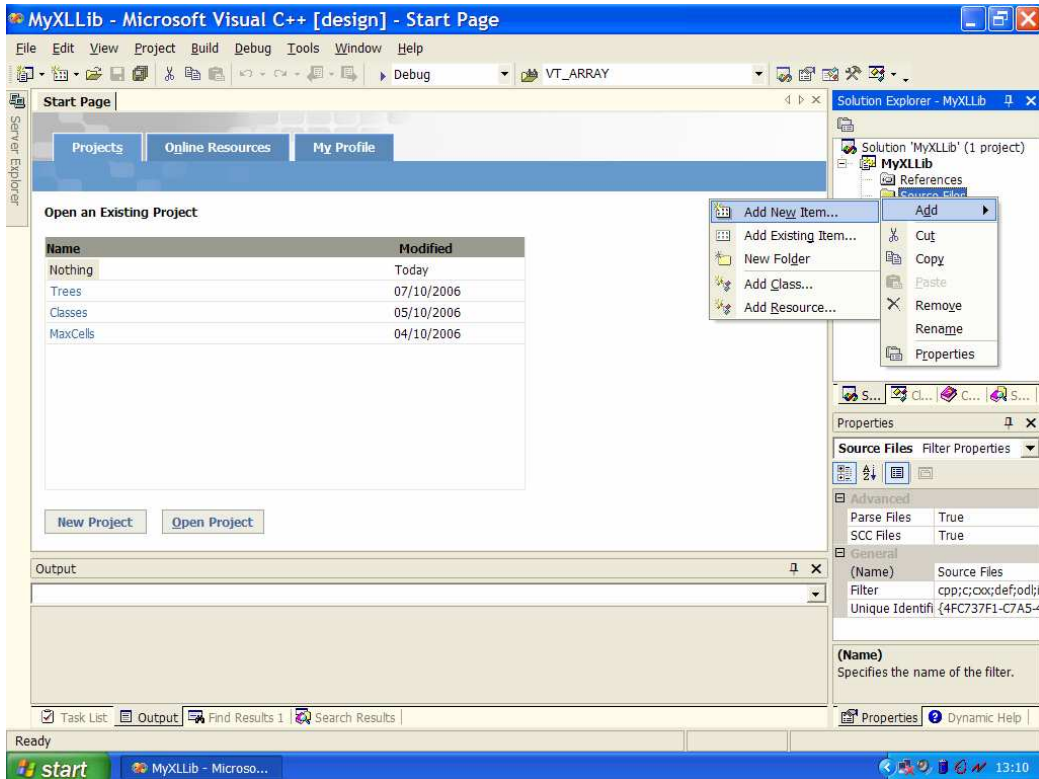


Figure 6: “Right-clicking” the Source File folder in the Solution Explorer of Visual Studio brings up a menu from which Add New Item may be selected.

```

LIBRARY MyXLLib
EXPORTS
    square_in_C

```

where `MyXLLib` is the name of the library (project). Any number of functions can be exported in this way by adding each function name on a new line below the `EXPORTS` command.

Finally, we build the library by calling the `Build Solution` option from the `Build` menu (`Ctrl+Shift+B`), which creates the file `MyXLLib.dll` in the `Debug` folder of the project. We have now created the library, but how do we make use of it from Excel?



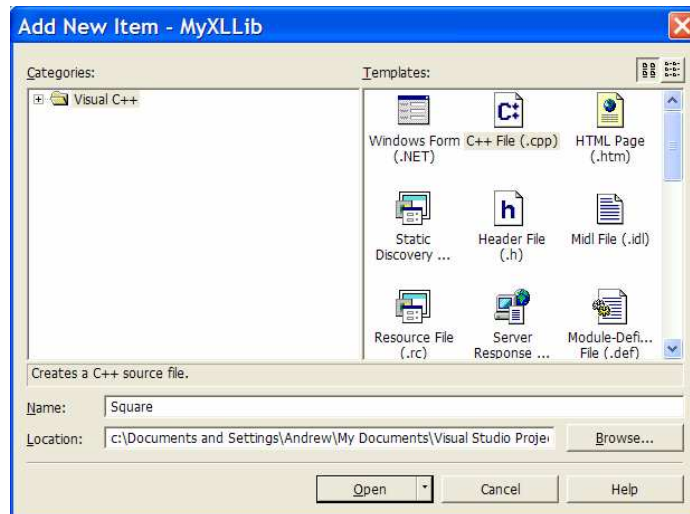


Figure 7: The Add New Item window in Visual Studio. In this case, a C++ file template has been selected that will be called `Square.cpp`.

### 5.1.2 Calling the library functions from within Excel

In order to use our newly written library, we must start Excel and use the Visual Basic Editor to declare our library functions. Press `Alt + F11` from within Excel to start the VBA Editor and then insert a Module (`Alt + I + M`), which creates an empty window. Enter the following in the newly created window

```
'Prototype the interface of the function from the library
Declare Function square_in_C _
Lib "C:/Path_to_Project/Debug/MyXLLib.dll" (arg As Double) as Double
```

Any line that starts with a single quote `'` is a comment in VBA. The `Declare Function` command states that we are defining a new function `square_in_C` that is contained in the library specified after the `Lib` keyword. Note that the path to the DLL must use the forward slash (`/`) and not the backslash (`\`) to separate folders. The underscore `_` is a VBA continuation character. Unlike C++, in VBA the end of a line represents the end of an instruction. If we have a long instruction that we wish to split over many lines, the character `_` is used to tell VBA that the next newline is not the end of the instruction, *i.e.* the instruction continues on the next line. The final part of the instruction specifies the argument and return types of the function.

We can now use the function `square_in_C` just as any other worksheet command, see Figure 8. The steps just described can be easily extended to C++ functions that take

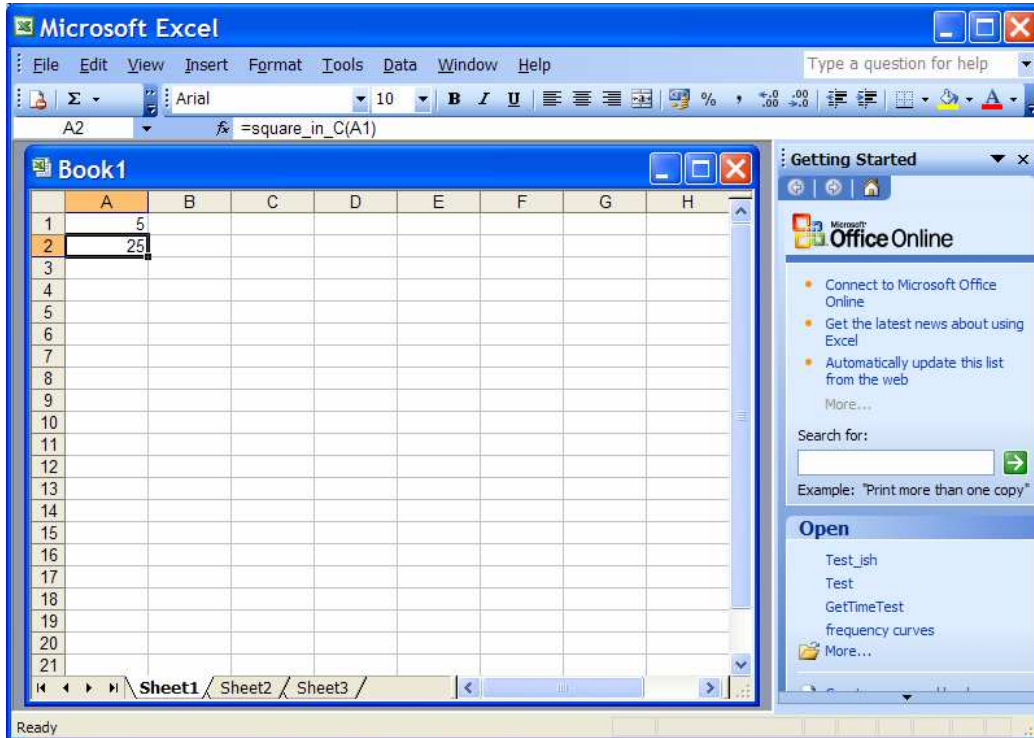


Figure 8: The `square_in_C()` function can be used just as any built-in function in an Excel spreadsheet.

many single arguments, e. g.

```
double __stdcall integrate(double &a, double &b){//Some commands}
```

would be prototyped in VBA as

```
Declare Function integrate _
Lib "C:/Path_to_lib/Lib.dll" (arg1 as Double, arg2 as Double) as Double
```

If you wish to write more complex C++ functions using arrays of cells as arguments, you must use a data type known as a `Variant`; many more details are provided in “Excel add-in development in C/C++” (2005) or “Financial Applications Using Excel Add-in Development in C/C++ (2nd Edition)” (2007) both by Steve Dalton and published by Wiley.

## 5.2 Using Excel from within C++

In order to use Excel functionality from within a C++ program, we shall use the Microsoft Component Object Model (COM). It is also possible to use COM to write Excel Add-Ins, but these are significantly more complex than the basic DLLs described above.

In order to use the COM interface, we must include a number of Microsoft libraries into our C++ source code. The exact location of these libraries will depend on the details of your installation. The required import commands are shown below and must be included at the top of every C++ program that communicates with Excel. The code in this section has been tested with Visual Studio.NET 2003 and 2005 and Excel 2003, but will not necessarily work with other versions of the software<sup>3</sup>.

```
//MicroSoft Office Objects
#import \
"C:\Program Files\Common Files\Microsoft Shared\OFFICE11\mso.dll" \
rename("DocumentProperties", "DocumentPropertiesXL") \
rename("RGB", "RBGXL")
```

---

<sup>3</sup>Thomas Seeley (Ohio State University) has kindly informed me that the following additional steps are necessary to make the code work with the latest (as of February 2009) version of Visual C++ Express Edition:

1. The library, “Microsoft Platform SDK” must be downloaded from the Internet.
2. The project must reference the Additional Include Directory:  
C:\Microsoft Platform SDK\Include
3. The project must reference the Additional Libraries Directory:  
C:\Microsoft Platform SDK\Lib
4. A few lines of code need to be added at the top of the main file:

```
#include <comutil.h>
#include <stdio.h>
#pragma comment(lib, "comsuppw.lib")
#pragma comment(lib, "kernel32.lib")
```

```
//Microsoft VBA Objects
#import \
"C:\Program Files\Common Files\Microsoft Shared\VBA\VBA6\vbe6ext.olb"

//Excel Application Objects
#import "C:\Program Files\Microsoft Office\OFFICE11\EXCEL.EXE" \
rename("DialogBox", "DialogBoxXL") rename("RGB", "RBGXL") \
rename("DocumentProperties", "DocumentPropertiesXL") \
rename("ReplaceText", "ReplaceTextXL") \
rename("CopyFile", "CopyFileXL") \
exclude("IFont", "IPicture") no_dual_interfaces
```

Note the use of the continuation character `\` so that compiler directives (commands beginning with `#`) can be split over multiple lines. The `rename("A","B")` directive changes the name of any strings A that occur in the imported library to B; and is required to avoid clashes with other libraries in which exactly the same variable, function or class names have been defined. The `exclude` directive prevents the import of the specified items from the library. The final directive `no_dual_interfaces` changes the manner in which overloaded functions are called. It must be included so that the `_ApplicationPtr`, which has a dual interface, is called in the correct manner.

We can now open an Excel Application from our C++ code, if we declare a pointer to an Excel Application and then create a single instance of Excel.

```
int main()
{
    Excel::_ApplicationPtr XL;

    //A try block is used to trap any errors in communication
    try
    {
        //Initialise COM interface
        CoInitialize(NULL);

        //Start the Excel Application
        XL.CreateInstance(L"Excel.Application");

        //Make the Excel Application visible, so that we can see it!
```

```
        XL->Visible = true;
    }
    //If a communication error is thrown, catch it and complain
    catch(_com_error &error)
    {
        cout << "COM error " << endl;
    }
}
```

The initialisation and creation of the Excel Application is surrounded by a `try` block, which is part of the C++ error-handling system. If any part of the code does something unexpected it should **throw** an exception, returning control to the calling function; and objects that indicate the type of error can be thrown, `throw object_constructor();`. The `try` block acts as a “safety net” that surrounds a section of code that you suspect might throw an exception (fail). If an exception is thrown within the `try` block it can be caught by the `catch` keyword and any appropriate action can be taken. In the above example, if there is a communications error a `_com_error` object will be thrown, which is caught and an error message is printed on the screen.

The result of the program is that an Excel window is created and displayed on our screen. The important feature is that we have a pointer to the Excel application that can be used to communicate between Excel and our C++ program. In fact, the program is merely a fancy “wrapper” that starts Excel. The inclusion of additional commands would allow us to start Excel with some custom defaults.

A pointer to an Excel Application Object is the only thing that is required to communicate with Excel from a C++, or any other, program. If we want to use any of the Excel objects and functions, however, we must know what they are called and the arguments that they take. The information is known as the Excel Object Model and is documented in the on-line help provided with the VBA editor, although it is not easy to decipher.

### 5.3 The Excel Object Model

The Excel Object Model contains over 200 objects and a vast number of associated functions. It is impossible to provide a detailed description of the complete model. Instead, we shall concentrate on a few key objects: `Application`, `Chart`, `Workbook` and `Worksheet`.

### 5.3.1 The Excel Application

The member functions (methods) and data of the Excel Application Object are largely responsible for “global” events and setting within the running instance of Excel. The only ones that we shall use are:

---

<code>Visible</code>	Boolean flag: if true the Excel Application Window is displayed
<code>ActiveWorkbook</code>	Return a pointer to the active workbook
<code>ActiveSheet</code>	Return a pointer to the active worksheet or chart
<code>Quit</code>	Close the Excel Application
<code>Workbooks</code>	Returns a pointer to the collection of workbooks

---

### 5.3.2 The Excel Workbook

Excel Workbooks are all stored in the Workbooks object of the Excel Application. The most useful member functions of the Workbooks object are shown in the table below:

---

<code>XL-&gt;Workbooks-&gt;Add(Excel::xlWorksheet);</code>	Add a Workbook containing a single Worksheet
<code>XL-&gt;Workbooks-&gt;Open(L"test.xls");</code>	Opens the Workbook <code>test.xls</code>
<code>XL-&gt;Workbooks-&gt;Close();</code>	Close all open Workbooks
<code>XL-&gt;Workbooks-&gt;Item["test.xls"];</code>	Return a pointer to <code>test.xls</code>

Each Workbook object contains a collection of Worksheets, a collection of Charts, and a collection of Sheets, which contains the Worksheets and the Charts. New Charts and Sheets can be created by calling the `Add()` member function of the Charts and Sheets collections, respectively.

### 5.3.3 The Excel Worksheet and Ranges

For most tasks, the interaction between C++ and Excel should take place at the “Worksheet” level. In essence, we want to be able to pass data between Worksheet cells and our C++ program. The worksheet member function `Cells` returns a pointer to all the cells in the worksheet.<sup>4</sup>

```
Excel::_WorksheetPtr pSheet = XL->ActiveSheet;
Excel::RangePtr pRange = pSheet->Cells;
```

---

<sup>4</sup>The Excel object model uses a custom data type called a **Range** to represent a collection of cells in a worksheet.

The two lines of code above return a pointer to a collection (a Range object) containing all the cells of the active worksheet<sup>5</sup>. Once the pointer to the Range has been defined we can access the cells by using the `Item` method with a standard C++ array syntax. The only catch is that the Excel arrays start from the index 1.

```
pRange->Item[1][1] = 1.0;
pRange->Item[10][1] = "=SIN(A1)";
```

The above code fragment will set the value in cell A1 to 1.0 and the value in cell A10 will be  $\sin(1) \approx 0.8$ . The numerical value of the cell A10 can be read into our program in the obvious way

```
double cell_value = pRange->Item[10][1];
```

We can now read and write to cells in an Excel worksheet, and can call any of the built-in Excel functions. The “interfacing” is done, but the difficult part, exactly what to write, is only just beginning. Unfortunately, there is not sufficient time to cover all possible cases of interest, we finish with a simple(?) example: how to generate a graph in Excel from data computed in C++.

```
// Include standard header files
#include <iostream>
#include <cmath>

// Office XP Objects (2002)
#import \
"C:\Program Files\Common Files\Microsoft Shared\OFFICE11\mso.dll" \
rename("DocumentProperties", "DocumentPropertiesXL") \
rename("RGB", "RBGXL")

//Microsoft VBA objects
#import \
"C:\Program Files\Common Files\Microsoft Shared\VBA\VBA6\vbe6ext.olb"

//Excel Application objects
```

---

<sup>5</sup>You may find that you need to use `XL->Cells` in the second line.

```

#import "C:\Program Files\Microsoft Office\OFFICE11\EXCEL.EXE" \
rename("DialogBox", "DialogBoxXL") rename("RGB", "RGBXL") \
rename("DocumentProperties", "DocumentPropertiesXL") \
rename("ReplaceText", "ReplaceTextXL") \
rename("CopyFile", "CopyFileXL") \
exclude("IFont", "IPicture") no_dual_interfaces

//Use the standard namespace
using namespace std;

//Define our own function, e^{-x} sin(x)
//In general this could be the result of a simulation, etc.
double f(const double &x) {return (sin(x)*exp(-x));}

//Main driver program
int main()
{
    //Surround the entire interfacing code with a try block
    try
    {
        //Initialise the COM interface
        CoInitialize(NULL);
        //Define a pointer to the Excel application
        Excel::_ApplicationPtr xl;
        //Start one instance of Excel
        xl.CreateInstance(L"Excel.Application");
        //Make the Excel application visible
        xl->Visible = true;
        //Add a (new) workbook
        xl->Workbooks->Add(Excel::xlWorksheet);
        //Get a pointer to the active worksheet
        Excel::_WorksheetPtr pSheet = xl->ActiveSheet;
        //Set the name of the sheet
        pSheet->Name = "Chart Data";
        //Get a pointer to the cells on the active worksheet

```



```

Excel::RangePtr pRange = pSheet->Cells;

//Define the number of plot points
unsigned Nplot = 100;
//Set the lower and upper limits for x
double x_low = 0.0, x_high = 20.0;
//Calculate the size of the (uniform) x interval
//Note a cast to an double here
double h = (x_high - x_low)/(double)Nplot;
//Create two columns of data in the worksheet
//We put labels at the top of each column to say what it contains
pRange->Item[1][1] = "x"; pRange->Item[1][2] = "f(x)";
//Now we fill in the rest of the actual data by
//using a single for loop
for(unsigned i=0;i<Nplot;i++)
{
    //Calculate the value of x (equally-spaced over the range)
    double x = x_low + i*h;
    //The first column is our equally-spaced x values
    pRange->Item[i+2][1] = x;
    //The second column is f(x)
    pRange->Item[i+2][2] = f(x);
}

//The sheet "Chart Data" now contains all the data
//required to generate the chart
//In order to use the Excel Chart Wizard,
//we must convert the data into Range Objects
//Set a pointer to the first cell containing our data
Excel::RangePtr pBeginRange = pRange->Item[1][1];
//Set a pointer to the last cell containing our data
Excel::RangePtr pEndRange = pRange->Item[Nplot+1][2];
//Make a "composite" range of the pointers to the start
//and end of our data
//Note the casts to pointers to Excel Ranges

```

```

Excel::RangePtr pTotalRange =
    pSheet->Range[(Excel::Range*)pBeginRange][(Excel::Range*)pEndRange];

// Create the chart as a separate chart item in the workbook
Excel::_ChartPtr pChart=xl->ActiveWorkbook->Charts->Add();
//Use the ChartWizard to draw the chart.
//The arguments to the chart wizard are
//Source: the data range,
//Gallery: the chart type,
//Format: a chart format (number 1-10),
//PlotBy: whether the data is stored in columns or rows,
//CategoryLabels: an index for the number of columns
//                containing category (x) labels
//                (because our first column of data represents
//                the x values, we must set this value to 1)
//SeriesLabels: an index for the number of rows containing
//                series (y) labels
//                (our first row contains y labels,
//                so we set this to 1)
//HasLegend: boolean set to true to include a legend
//Title: the title of the chart
//CategoryTitle: the x-axis title
//ValueTitle: the y-axis title
pChart->ChartWizard((Excel::Range*)pTotalRange,
                  (long)Excel::xlXYScatter,
                  6L,(long)Excel::xlColumns, 1L, 1L, true,
                  "My Graph", "x", "f(x)");

//Give the chart sheet a name
pChart->Name = "My Data Plot";
}
//If there has been an error, say so
catch(_com_error & error)
{
    cout << "COM ERROR" << endl;
}

```

```
//Finally Uninitialise the COM interface
CoUninitialize();
//Finish the C++ program
return 0;
}
```

The result of the above program is that an Excel worksheet will be created that contains a line graph of the function  $f(x) = e^{-x} \sin x$  for  $x \in [0, 20]$ , see Figure 9. An important point is that even though the C++ program has finished, its “child” Excel program will continue to run until closed as usual.

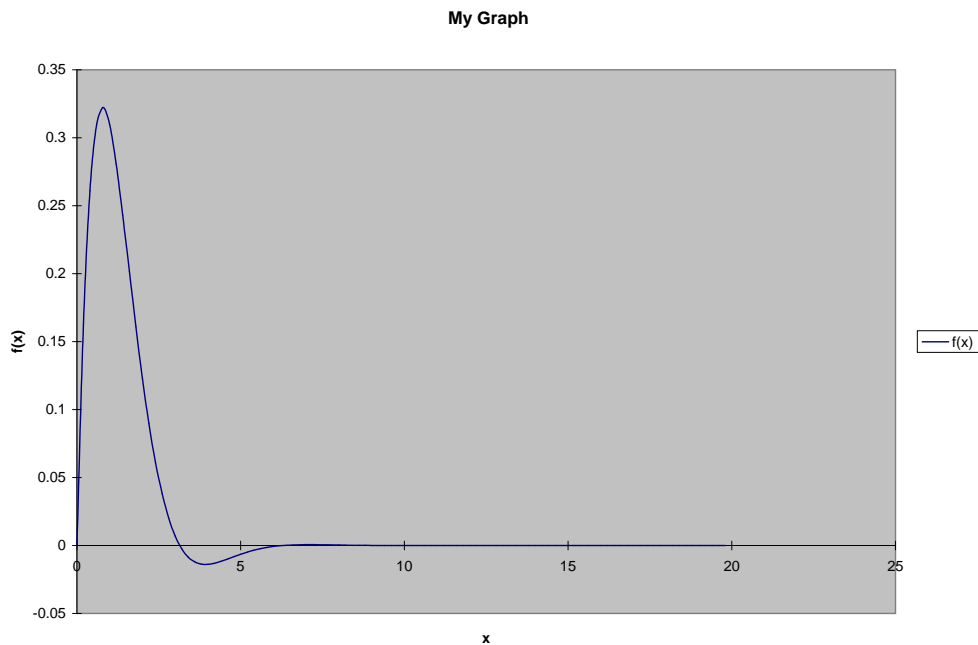


Figure 9: The graph of  $f(x) = e^{-x} \sin x$  produced by Excel driven from the C++ program listed above.