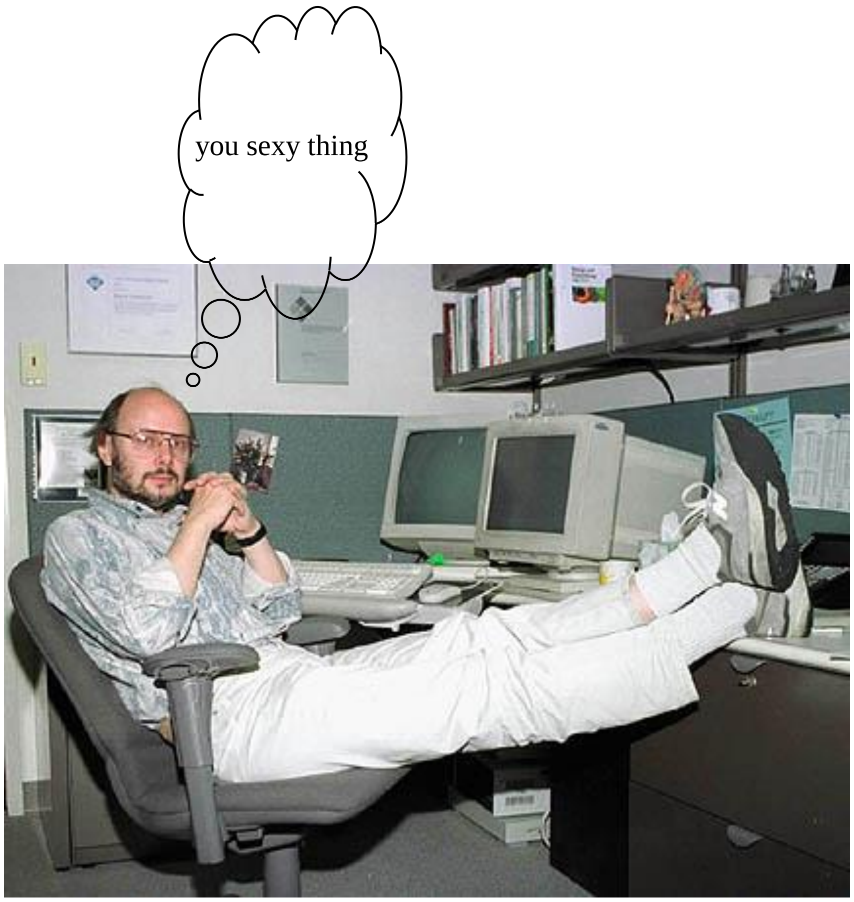
Applied Computational Finance in C++

Riaz Ahmad



DEPARTMENT OF MATHEMATICS
University College London

"If programming in Pascal is like being put in a straightjacket, then programming in C is like playing with knives, and programming in C++ is like juggling chainsaws."

Anonymous.

1. Fundamentals

1.0 Introduction

These days it is hard to escape financial news; whether we watch the news reports on TV, breeze through the financial pages in the press, or look at our hand held devices. Terms such as *Derivatives*, LIBOR, *short selling* or FTSE100 are constantly reminding us that finance is the most global of industries. The financial crisis has been the chief headline across worldwide news bulletins and remains a fierce topic for discussion.

Undeniably, the way finance has developed in recent years can be attributed to the part of mathematics (including computational power), which has played a central role and remains the chief driving force allowing the financial markets to become increasingly sophisticated. There is little doubt that maths has 'hijacked' most disciplines and its appeal and influence is noticeable in most branches of knowledge. In addition to the traditional areas of scholarship that depend on maths for its framework; less obvious academic themes are also enjoying the tangible advances being made due to the reliance on maths, such as Political Science, Medical Research and Sociology.

Mathematical modelling is the science of applying mathematics to understand the functioning of real-life complex situations arising in engineering, science, medicine and more recently in finance. Unless the problem is ideal/simple, it is unlikely that an analytical (closed form) solution can be obtained; reliance on computer programming will be necessary. Solutions of such problems help in understanding real-world systems. The availability of computing power and programming design has enabled mathematicians to study increasingly difficult problems and in turn has allowed industry to continuously evolve with greater technologically advanced developments. Whilst some argue that the collaboration between academia and industry is far from optimal, Quantitative Finance is an example of the 'perfect' partnership.

The aim of this book is to introduce C++ programming applied to solving problems in mathematics, particularly those numerical techniques and code, applied to financial mathematics. It assumes the student has very little exposure to the language or is completely new to the subject. Within finance the widespread use and importance of this increasingly popular language has become central to the area of derivative pricing. It has become the main mode of technology within banking whilst retaining the status of being a "sexy" language in the global financial markets.

Whilst this opening section is not intended to be a marketing drive in praise of C++; we all depend on the use of programs designed in this language, whether in part or entireity. Search engines (Google); operating systems (Windows, Linux); online shopping (Amazon); computer games (Call of Duty); Adobe and MicroSoft software; are a few examples of how we rely on C++ on a daily basis. For this reason C++ has acquired the status of being a 'legacy language'.
The origins of this book date back to 2003 when it was initially written as a concise set of notes for practitioners attending short courses at FitchLearning (part of the Fitch Group). In 2008 these notes were edited for teaching on the University of Oxford MSc Mathematical and Computational Finance programme. They were further revised and used to deliver the Applied Computational Finance in 2013, as part of the new MSc Financial Mathematics degree at University College London.

There is no doubt that since the nineties Object Oriented Programming (OOP) has created much excitement. As a branch of software development OOP has been a major innovative theme, and C++ is the most widely used language, which supports the object-oriented paradigm. OOP is a design philosophy that superseded languages such as Pascal and C; defined broadly as procedural/structured programming this was the principle of reducing a code into smaller and independently functioning parts. It supported efficient programming when applied to moderately complex systems. With the requirement of larger and further complex programs, this mode of programming was not so effective. Whilst OOP has inherited the best ideas from structured programming, what makes it vastly different is that it encourages the decomposition of the problem into related subgroups or self-sustainable 'objects'. Each object contains its own related data and instructions. This functionality promotes reusability and maintainability thus reducing the overall complexity of the problem. The three fundamental concepts shared by all object-oriented languages are

- Encapsulation – 'bundling' together of data and code and hiding it. It is the ability to define a new user type and set of operations on that type. An object contains both data and code, without revealing the implementation of the object.
- Polymorphism – the word is comprised of *poly* meaning 'many' and *morph* i.e. 'shape'. It is the ability of performing many different (but related) tasks using the same name. For example a function with name **average** may have three different definitions.
- Inheritance – creating new types that inherit properties from existing ones. This is the cornerstone of reusability.

These are the basic strands of OOP; some books may list more, e.g. *Abstraction* and *Specialization*.

‘Derivatives’ as a branch of modern finance continues to be one of the fastest growing areas within the corporate world. The sophistication and complexity of modern financial product’s, continually acts as the motivation for new mathematical models and the subsequent development of computational schemes. Investment decisions for predicting risk and return are being increasingly based on principles taken from the Quantitative Finance arena, providing a challenge for both academics and practitioners.

The field of mathematical finance has become particularly prominent due to the much-celebrated Black-Scholes equation written in 1973 by Fischer Black, Myron Scholes and Robert Merton, for which the latter two were awarded the Nobel prize for economics, in 1997. Fischer Black sadly passed away in 1995; the Nobel Prize is not awarded posthumously. The origins of quantitative finance can however be traced back to the start of the twentieth century. Louis Jean-Baptiste Alphonse Bachelier (March 11, 1870 - April 28, 1946) is credited with being the first person to derive the price of an option where the share price movement was modelled by Brownian motion, as part of his PhD, entitled *Théorie de la Spéculation* (published 1900). Thus, Bachelier may be considered a pioneer in the study of financial mathematics and one of the earliest exponents of Brownian Motion.

In recent years there has been an explosion in the availability (and popularity) of advanced degrees in mathematical finance at some of the top tier universities internationally, which are aimed at leading numerate graduates towards highly lucrative careers as quantitative analysts (quants), developers and traders in investment banks and hedge funds. Whilst universities are producing graduates with excellent training in the mathematical methods applied to quantitative finance, there is clearly a lack of focus on the development of practical computational skills in C++. Currently programming expertise in C++ for a quant is of equal importance as mathematical proficiency. At the interview stages, many capable mathematicians fail to demonstrate the necessary degree of competence in programming. The UCL MSc programme has addressed this concern from the outset, and a major component of this degree is the inclusion of C++ instruction during the course. An MSc in Mathematical Finance is the basic prerequisite for a serious approach to quantitative analysis work in investment banking. The combination of mathematics and programming provides a very attractive mix of technical expertise for any prospective employer, or forming the basis of future research work.

Generally, in applied mathematics, solving a real-life problem computationally consists of three general stages:

1. Setting up the mathematical framework consisting of a Partial Differential Equation (PDE) or Stochastic Differential Equation (SDE) together with suitable boundary and/or initial conditions, based on the real world industrial principles, of the problem we wish to model.
2. Applying appropriate numerical methods (depending on computational requirements in terms of economy and efficiency) to reduce the mathematics to an algorithmic form. This leads on to expressing the algorithm in language-independent format (or pseudo code). In mathematical modelling, this is usually the most complicated stage.
3. Writing a program to solve the problem in a computer language (in our case C++).

What makes C++ the enormously powerful language that it is, is that it has inherited the linguistic constructions and concepts of the past four decades of language design. The downside of evolving from its predecessors is the creation of a technology with many complex, intricate and potentially dangerous features.

C++, was created and developed by the Danish computer scientist Bjarne Stroustrup of AT&T Bell Laboratories in 1982, and is based on the C language; although the history dates back to his PhD research in 1979 which centred on the use of a simulations language called *Simula*. The name C++ is actually a pun – (as will become apparent later) "++" is a syntactic construct used in C (to increment a variable). The C language was developed ten years earlier by Dennis Ritchie at Bell Labs. C++ is intended as an incremental improvement of C, and C is considered a subset of C++, so that most C programs can be compiled using a C++ compiler (but not conversely).

C is in many ways hard to categorise. Compared to (say) assembly language it is high-level, but it nevertheless includes many low-level facilities (e.g. pointers) to directly manipulate the computer's memory. It is therefore an excellent language for writing efficient "systems" programs. But for other types of programs, C code can be hard to understand, and C programs can therefore be particularly prone to certain types of error. The extra object-oriented facilities in C++ are partly included to overcome these shortcomings. C++ is a self contained language and hence requires no knowledge of C.

The major problem with large computer programs is complexity. This makes programs significantly more prone to error, and software bugs can be expensive and in some cases even life threatening (e.g. air traffic control systems, medical equipment). Object oriented programming has successfully addressed these problems by offering a powerful way to cope with this complexity, by providing clearer, more reliable and easy-to-maintain programs.

Fortran (*FOR*mula *TRAN*slation) which originally stood for *IBM Mathematical FORmula TRANslation System* was specifically designed for scientific programming and was particularly simple enough for self instruction. It is the oldest of the established “high-level” languages, having been designed by a group in IBM during the late 1950s. Many research students only encountered the need for writing code when embarking upon a PhD. The many in-built features of this language made it particularly attractive when using numerical schemes for solving problems in applied mathematics.

C++ is well known for being a technology that is difficult to teach yourself, and it is widely accepted that it is the most complex of all the programming languages with a general view that “if you can program in C++, you can program in any language.” Unlike many other languages, it is not simply a collection of syntactical statements, but a programming ideology. It is important to start developing C++ code immediately, hence the theme will be application based, with all examples being of a mathematical nature. The notes are presented in a way, which makes them self-contained and complete, hence reducing the need to consult C++ books to a minimum. The emphasis on developing good quality, robust and computationally efficient code will be used through out. Like mathematics, programming is not a spectator sport – and hence the key to gaining confidence and developing skills here is to work through the notes and exercises and write as much code as possible.

All programs presented are written in a Microsoft PC environment using the MS Visual Studio platform. A further advantage of using MS Windows is that we can make use of software such as excel or word. A C++ program is simply a text file with a certain form of syntax, which is specific to this programming language. In a windows operating system, files of this kind have an extension “.cpp” which enable to distinguish it from other types e.g. a MS Word file or a VBA program.

Such text files are also called *source* files, as they provide the source code for the actual program. To run the program, a C ++ compiler is then required to translate the source files into an executable file format, that can be executed by the operating system (in our case Windows). This is called a binary file (or machine code) consisting of zeroes and ones and is understood by the computer. In this compilation process of translating language syntax to machine language, any errors by the programmer due to use of incorrect syntax, found by the compiler, and called *compile-time errors*, are flagged with messages. Although these messages are designed to help in locating and rectifying them, they can sometimes be ambiguous and unclear. For example, a missing semi-colon can give rise to over one hundred errors at compile-time!

Returning to the text editor and correcting these mistakes is required before recompiling the revised source code (which in many cases may have to be repeated several times) in order for the compilation to be successful, and produce an *object module* containing the machine language translation of your source code.

A linker is then used to *link* the program which we have compiled, with a selection of other necessary routines (which are found in other existing and built-in libraries). This procedure creates the complete machine executable object program.

The numerous programs included in this set of notes are provided so that the reader may copy them into a C++ text editor and run. The code should then be edited to see how the user might write it themselves or if there are any aspects he/she thinks that need improvement, should revise accordingly. This is the key to gaining proficiency in a new language – the quicker a sense of independence is achieved, the better. Whilst the code can be copied and pasted in to a C++ source file, it is better to get the feel for typing up lines of code; in the same way a server in tennis goes through the ritual of bouncing the tennis ball several times prior to serving.

1.1 Getting started and basic Program Construction

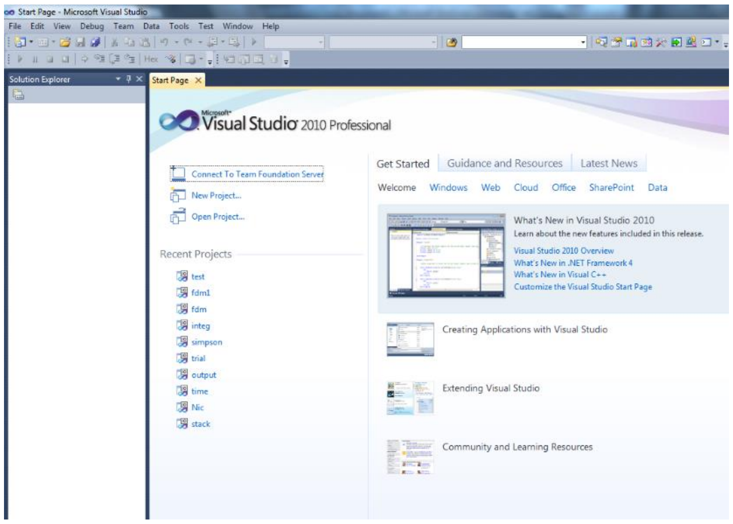
For those who are completely new to programming, a C++ program is essentially a file(s) containing text. Composition of this text follows a particular sentence structure specific to C++; this is called *syntax* and the resulting file is called a *source file*. Such files in C++ can be distinguished (from say *header files*) by the extension .cpp (windows operating system) or .cc (for linux). More on header files later.

Before executing a program, the syntax needs to be checked for any errors and this is done by a piece of software called a compiler. If there any mistakes these are identified, in order for the programme to be corrected. After compiling the edited version this is converted to an executable/binary file, which contains the instructions in a format that the computer can understand.

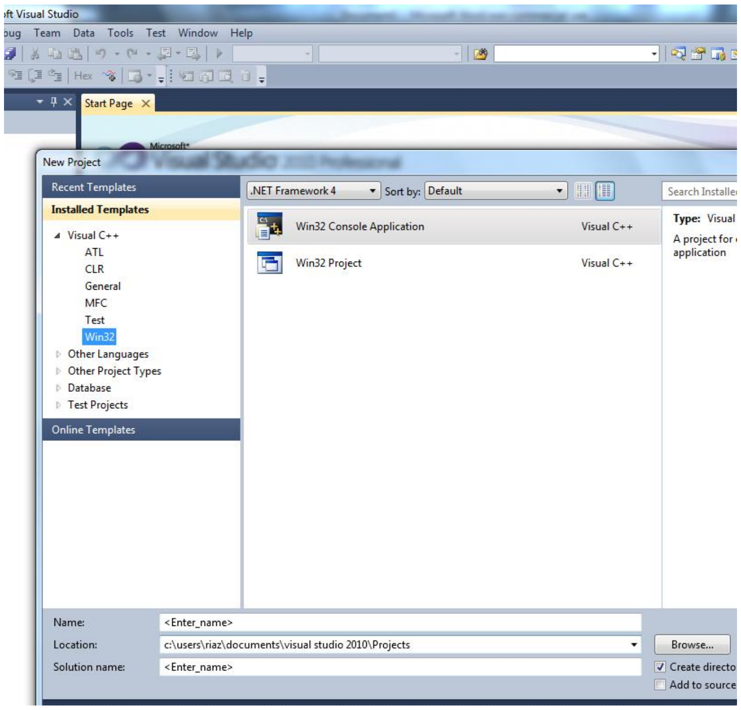
C++ is a *strongly typed* language. That is, everything must be declared before it is used – both variables and functions. Informally this also means that the language ie less forgiving in relation to small errors.

The best way to proceed is to start creating basic Windows desktop applications, i.e. a Win32 console application is the most basic way to begin.

Microsoft Visual Studio 2010 is being used throughout this document. Upon opening this software the screen below appears

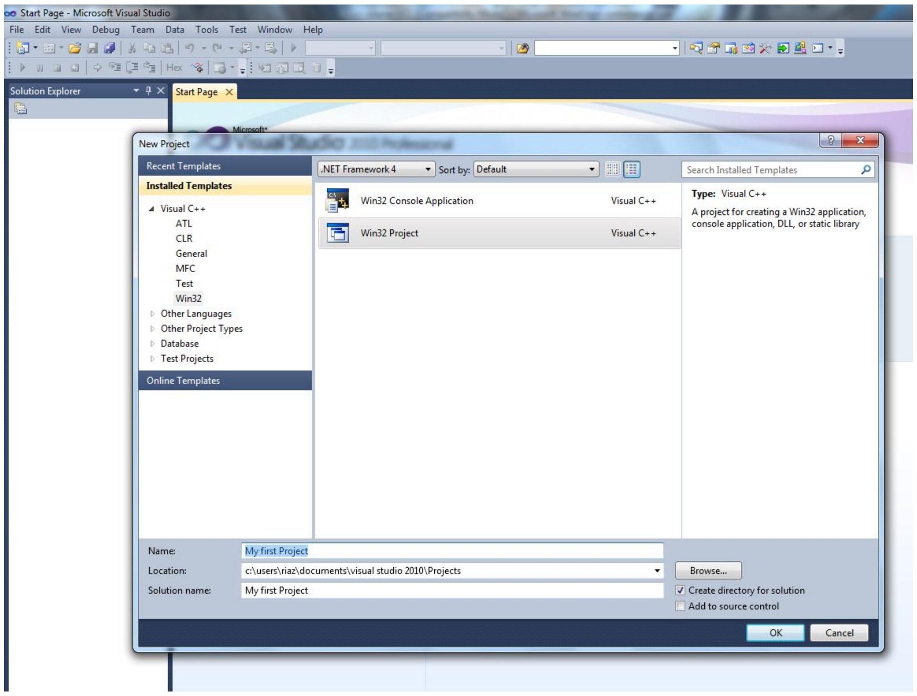

Start Page - Microsoft Visual Studio
File Edit View Debug Team Data Tools Test Window Help
Solution Explorer
Start Page
Visual Studio 2010 Professional
Connect To Team Foundation Server
New Project...
Open Project...
Recent Projects
test
fdm1
fdm
integ
simpson
trial
output
time
Nic
stack
Get Started
Guidance and Resources
Latest News
Welcome
Windows
Web
Cloud
Office
SharePoint
Data
What's New in Visual Studio 2010
Learn about the new features included in this release.
Visual Studio 2010 Overview
What's New in .NET Framework 4
What's New in Visual C++
Customize the Visual Studio Start Page
Creating Applications with Visual Studio
Extending Visual Studio
Community and Learning Resources

Click on New Project... on the left hand side



Name: <Enter_name>
Location: c:\users\riaz\documents\visual studio 2010\Projects
Solution name: <Enter_name>

Click on Win32 Project once and type the name you wish to call your project by, where it displays <Enter_name>. I have called this 'My first Project' which by default becomes the Solution name.



Win32 Console Application	Visual C++
Win32 Project	Visual C++

Type: Visual C++

A project for creating a Win32 application, console application, DLL, or static library

Name: My first Project

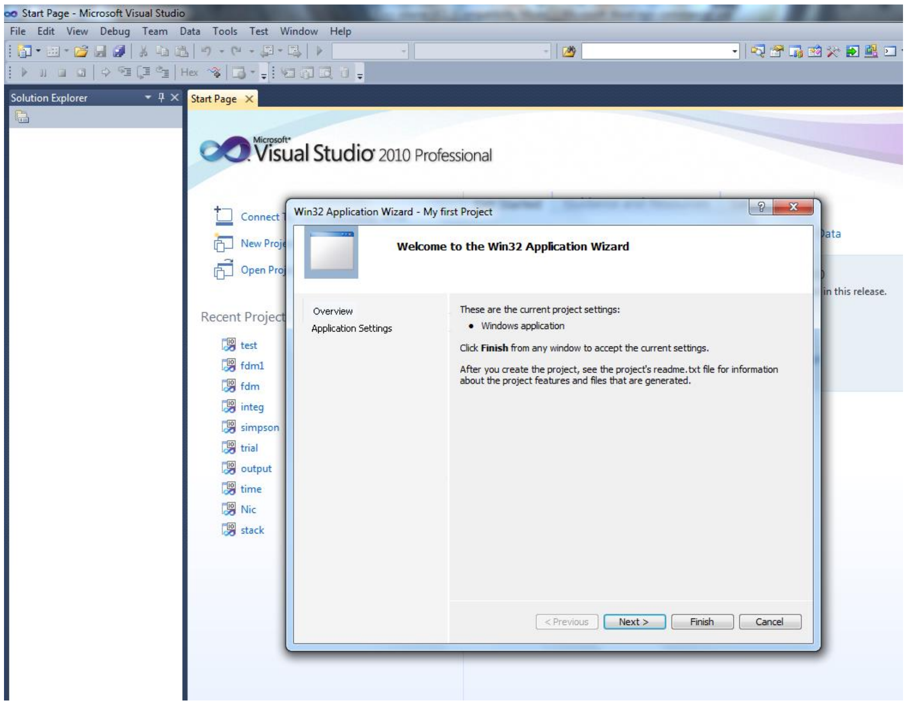
Location: c:\users\riaz\documents\visual studio 2010\Projects

Solution name: My first Project

[✓] Create directory for solution

[ ] Add to source control

Click OK



Win32 Application Wizard - My first Project

Welcome to the Win32 Application Wizard

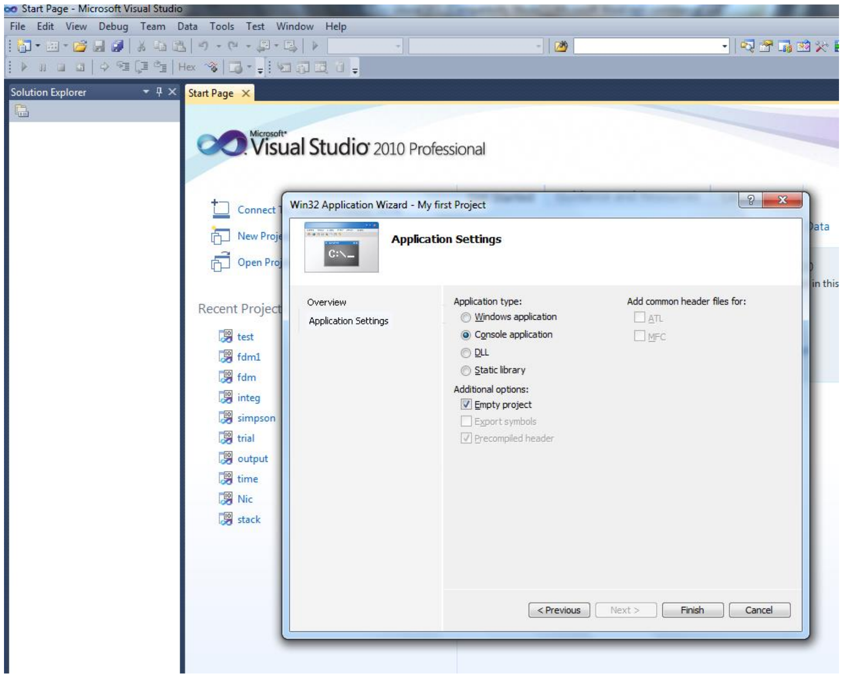
These are the current project settings:

- Windows application

Click **Finish** from any window to accept the current settings.

After you create the project, see the project's readme.txt file for information about the project features and files that are generated.

Now click **Next >** and make sure both **Console application** and **Empty project** are checked



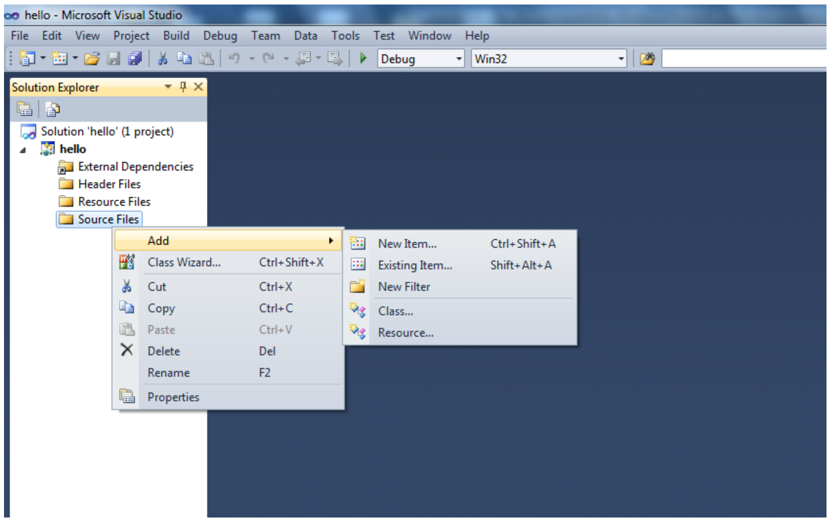
Application type:
- Windows application
- [✓ Console application]
- DLL
- Static library

Additional options:
- [✓] Empty project
- [ ] Export symbols
- [✓] Precompiled header

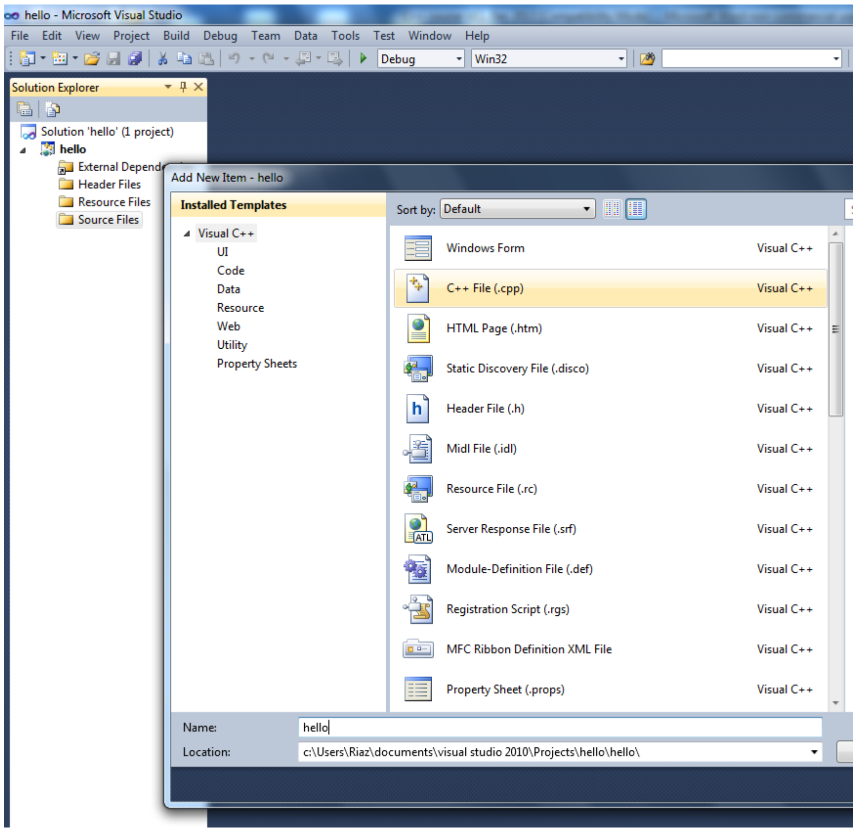
Add common header files for:
- [ ] ATL
- [ ] MFC

Click **Finish**

Now right click on Source Files then choose Add New Item



Which shows the following



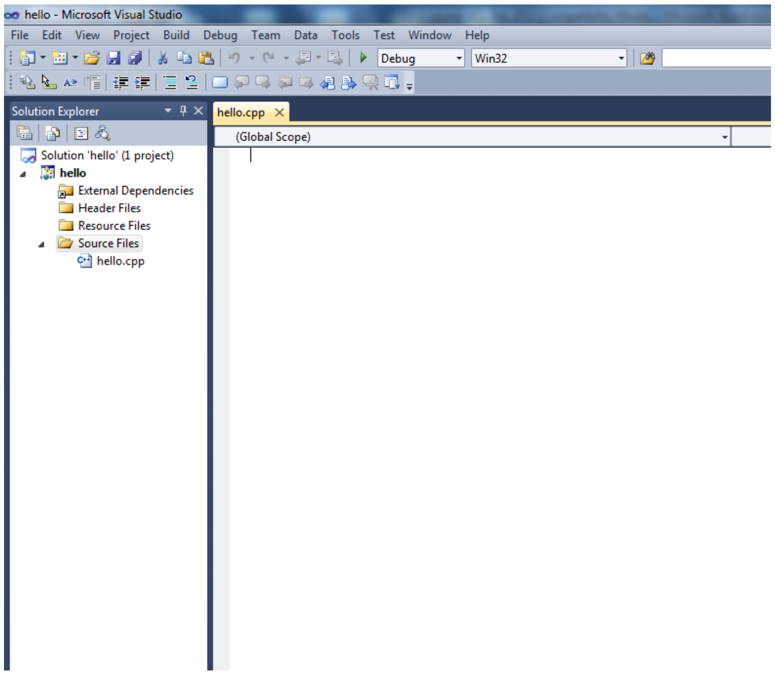
Sort by: Default

Windows Form	Visual C++
C++ File (.cpp)	Visual C++
HTML Page (.htm)	Visual C++
Static Discovery File (.disco)	Visual C++
Header File (.h)	Visual C++
Midl File (.idl)	Visual C++
Resource File (.rc)	Visual C++
Server Response File (.srf)	Visual C++
Module-Definition File (.def)	Visual C++
Registration Script (.rgs)	Visual C++
MFC Ribbon Definition XML File	Visual C++
Property Sheet (.props)	Visual C++

Name: hello

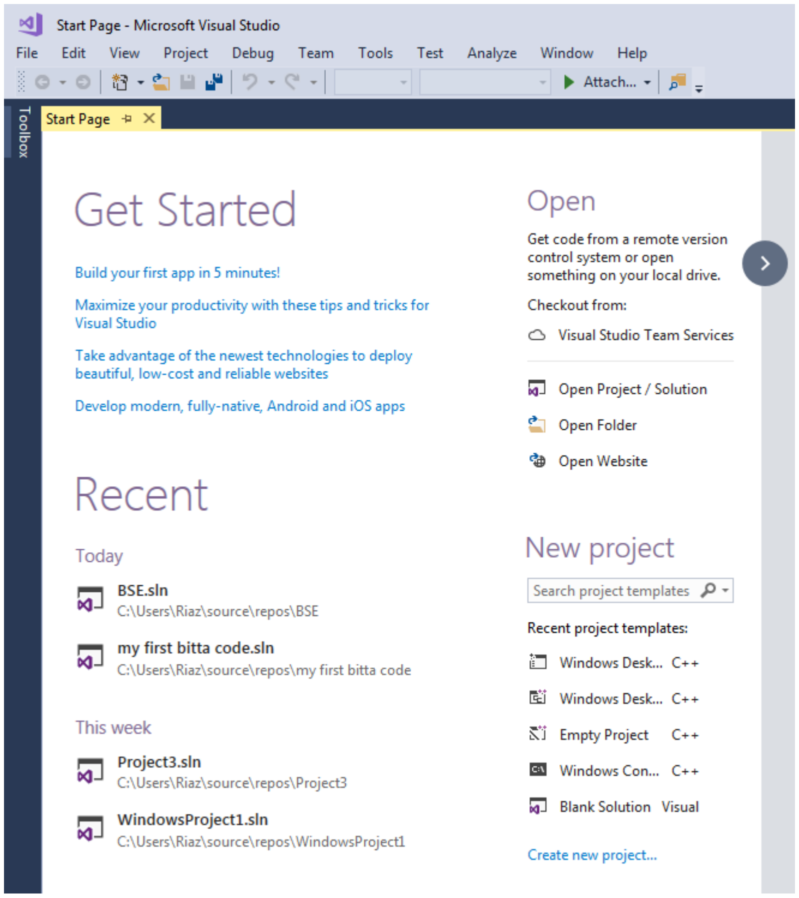
Location: c:\Users\Riaz\documents\visual studio 2010\Projects\hello\hello\

and click on **C++ File (.cpp)**; insert the C++ file name, in this example the source file is called hello (we don't add the extension .cpp as that is done automatically). In Location it shows precosely where on your hard disk the project is stored. Hit **Enter** or click Add to see the followng screen


hello - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
Win32
Solution Explorer
Solution 'hello' (1 project)
hello
External Dependencies
Header Files
Resource Files
Source Files
hello.cpp
hello.cpp
(Global Scope)

You are now ready to start coding in the large blank screen on the right hand side!

This is how to start a project in Visual Studio 2017



Get Started

Build your first app in 5 minutes!

Maximize your productivity with these tips and tricks for Visual Studio

Take advantage of the newest technologies to deploy beautiful, low-cost and reliable websites

Develop modern, fully-native, Android and iOS apps

Recent

Today

BSE.sln
C:\Users\Riaz\source\repos\BSE

my first bitta code.sln
C:\Users\Riaz\source\repos\my first bitta code

This week

Project3.sln
C:\Users\Riaz\source\repos\Project3

WindowsProject1.sln
C:\Users\Riaz\source\repos\WindowsProject1

Open

Get code from a remote version control system or open something on your local drive.

Checkout from:

Visual Studio Team Services

Open Project / Solution

Open Folder

Open Website

New project

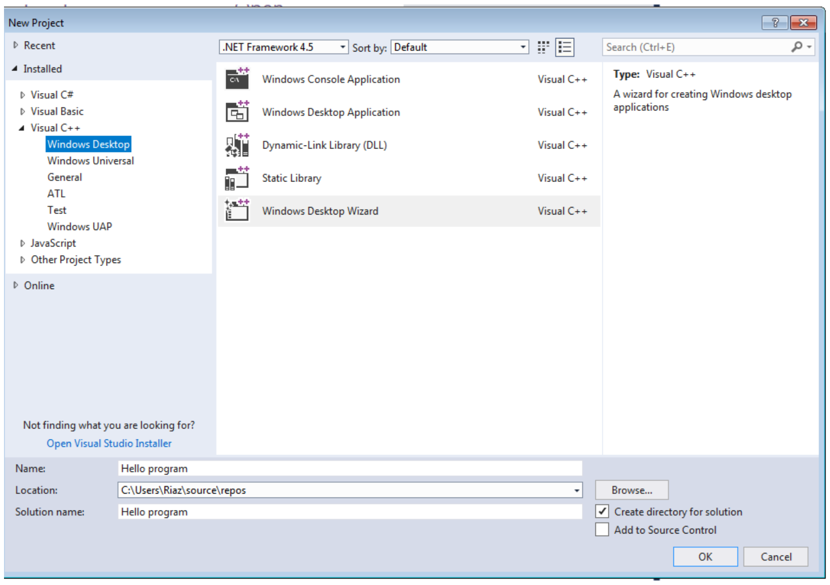
Search project templates

Recent project templates:

Windows Desk...	C++
Windows Desk...	C++
Empty Project	C++
Windows Con...	C++
Blank Solution	Visual

Create new project...

Click on create new project (at the bottom) and the following appears



Windows Console Application	Visual C++
Windows Desktop Application	Visual C++
Dynamic-Link Library (DLL)	Visual C++
Static Library	Visual C++
Windows Desktop Wizard	Visual C++

Type: Visual C++

A wizard for creating Windows desktop applications

Not finding what you are looking for?
Open Visual Studio Installer

Name: Hello program

Location: C:\Users\Riaz\source\repos

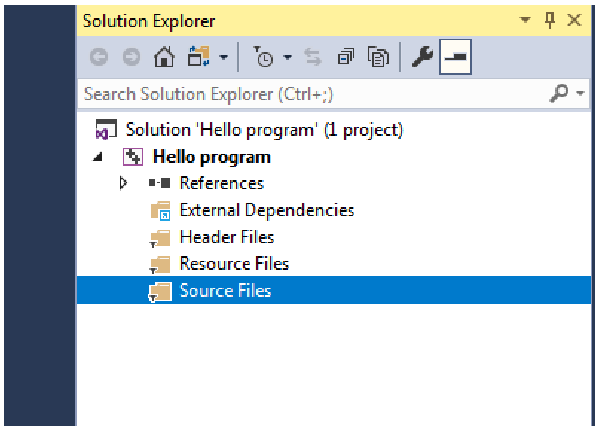
Solution name: Hello program

[✓] Create directory for solution

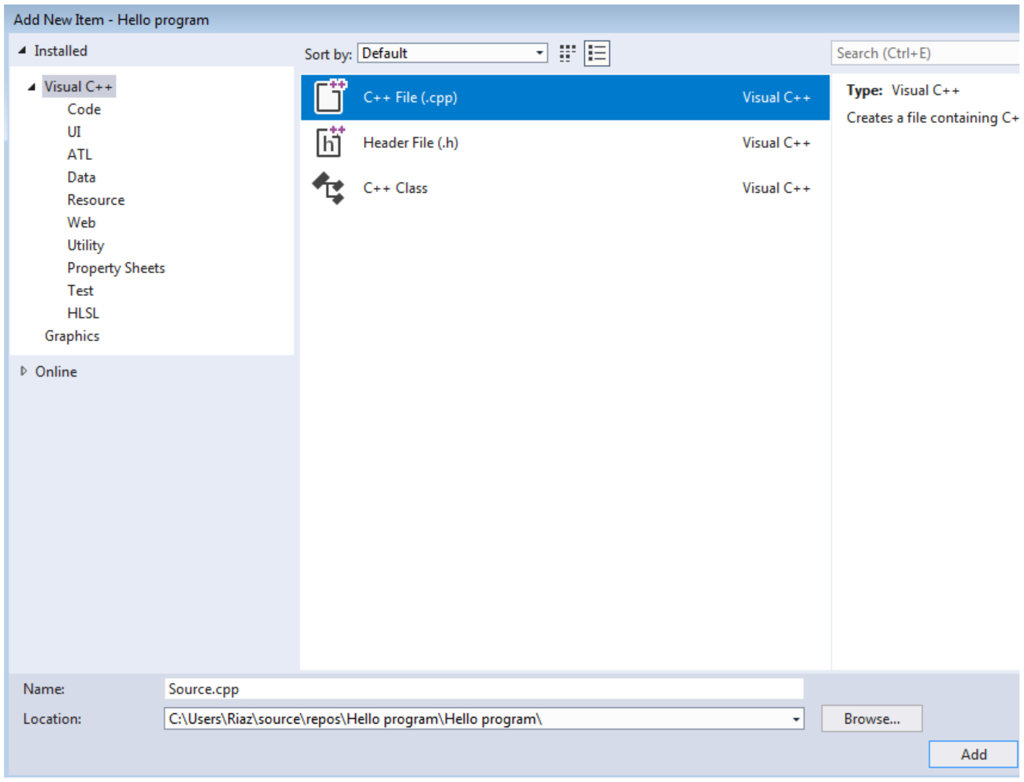
[ ] Add to Source Control

Windows desktop on the right hand pane should be highlighted, click on Windows desktop wizard and enter project below (in my example it is) Hello World and click OK.

On the resulting view, the ‘wiindows desktop project’ ensure the only box that is checked is **empty project**.



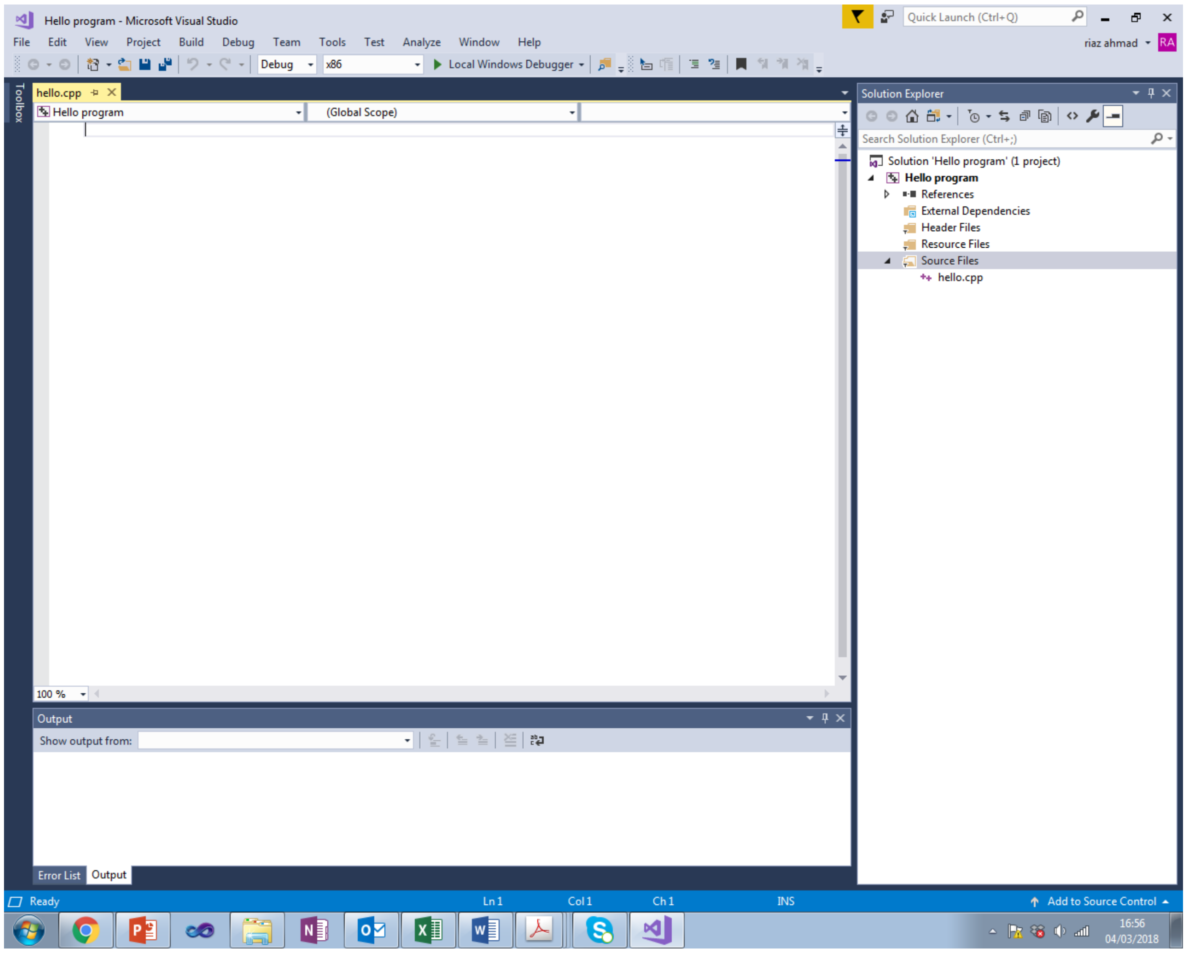
Right click on source files and choose Add → New Item



Name: Source.cpp

Location: C:\Users\Riaz\source\repos\Hello program\Hello program\

Click the first icon (C++File .cpp) and choose Name: ; then Add and you will have a Source File titled hello.cpp

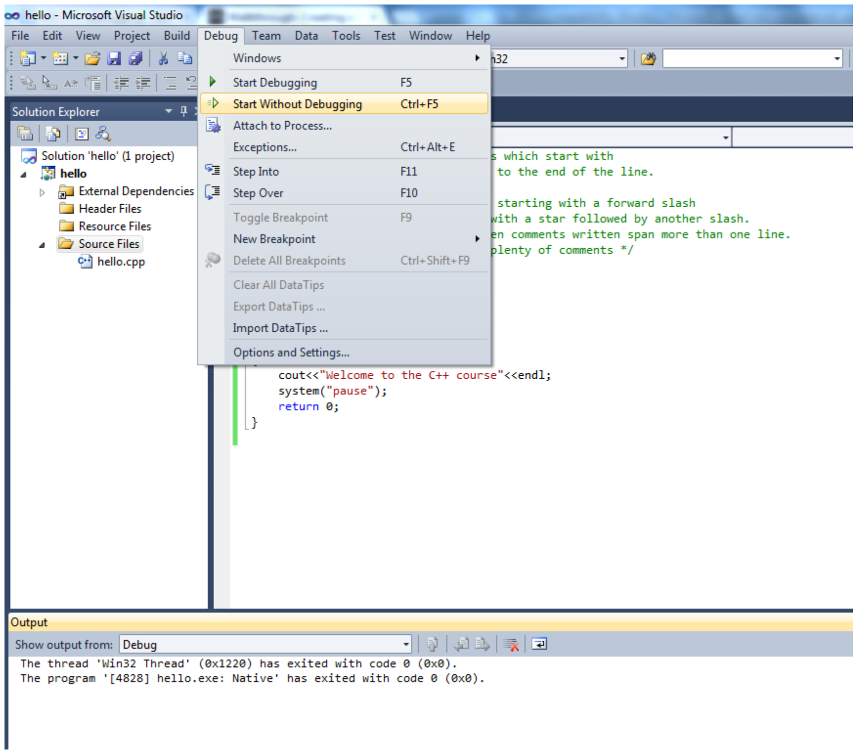


And you are ready to code.

One of the first programs written when learning any new language is to output a “hello” message to the screen. In conforming to tradition, we begin by looking at a basic C++ program, which does this and introduces the idea of simple output in C++.

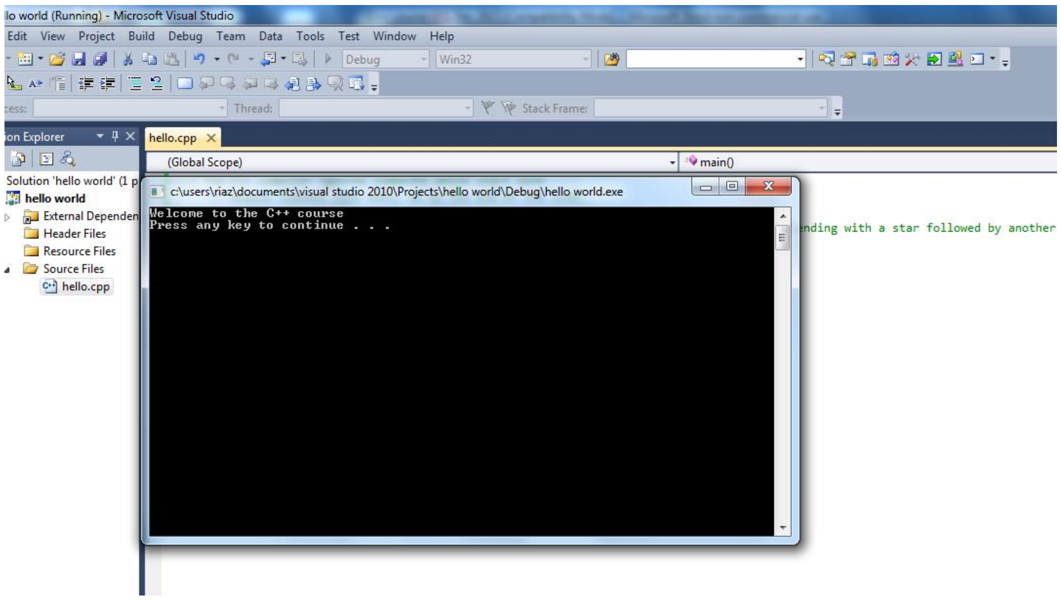
```
// The C++ compiler ignores comments which start with
// double slashes like this, up to the end of the line.

/* Comments can also be written starting with a forward slash followed by a star,
and ending with a star followed by another slash. The purpose of this style is when
comments written span more than one line. Programs should ALWAYS include plenty of
comments */


#include<iostream>
using namespace std;

int	main()
{
	// Interesting code commences

	cout<<"Welcome to the C++ course"<<endl;

	// Interesting code ends

	return 0;
}
```

The quickest way to start running programs is to choose Debug on the toolbar and click on the ‘Start’ options as shown below. In this case ‘Start without Debugging’ is shown. The Output pane displays relevant messages during the course of this process.



```
s which start with
 to the end of the line.

 starting with a forward slash
with a star followed by another slash.
en comments written span more than one line.
plenty of comments */

    cout<<"Welcome to the C++ course"<<endl;
    system("pause");
    return 0;
}
```

```
The thread 'Win32 Thread' (0x1220) has exited with code 0 (0x0).
The program '[4828] hello.exe: Native' has exited with code 0 (0x0).
```


lo world (Running) - Microsoft Visual Studio
Edit View Project Build Debug Team Data Tools Test Window Help
Debug
Win32
Thread:
Stack Frame:
ion Explorer
hello.cpp
(Global Scope)
main()
Solution 'hello world' (1 p
hello world
External Dependen
Header Files
Resource Files
Source Files
hello.cpp
c:\users\riaz\documents\visual studio 2010\Projects\hello world\Debug\hello world.exe

```
Welcome to the C++ course
Press any key to continue . . .
```

```
nding with a star followed by another
```

Note from the previous screenshot an additional line of code is included

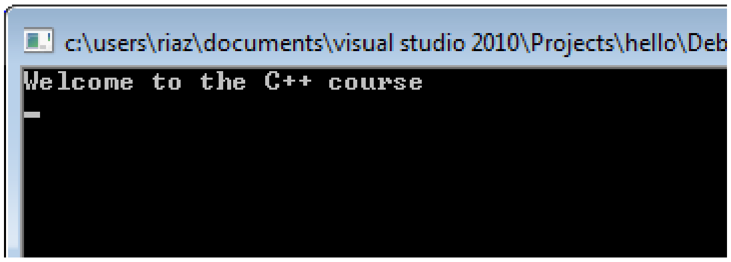
`system(“pause”);`

This will ensure that the DOS screen above does not flash in front of you and close. This is just one way of keeping the screen open; pressing any key will then close it.

There is another way to ensure the DOS screen remains open until otherwise specified. In place of the system(“pause”); statement, type

```
cin.get();
```

to produce the following to the screen



```
c:\users\riaz\documents\visual studio 2010\Projects\hello\Deb
Welcome to the C++ course
_
```

Note the cursor on the next line is waiting to be prompted. Hitting the enter key will close this screen.

Despite the size and simplicity, this program illustrates several general features of all of C++ programs. As a first exercise, it is important to insert the above into a C++ source file and ensure it compiles and runs properly; then the change the 'welcome' message to something of your choosing!

The two lines immediately following the comments are directives. They are examples of a

- preprocessor directive
- using directive

in turn. The first line of the program is

```
#include <iostream>
```

This statement is called an include directive. It is not part of any function and neither does it end in a semi-colon – which indicates the end of a statement. Program statements instruct a computer to perform tasks, for example print a message to the screen or do a mathematical calculation. So for this reason such a directive is not an example of a program statement. In C++ all pre-processor directives start with the hash symbol #called an **octothorp**. What makes C++ powerful is that many functions have already been written for us; for this reason our programs often refer to variables and code that lie outside the source code the programmer actually writes. For example, obtaining the cosine of a number can be a fairly complex mathematical task; the creators of C++ have written functions that perform trigonometric. So a programmer choosing to do this can include the function `cos()` in their code but only if the program can find it when linked to outside files to create executable files.

In our short program above the include directive tells the compiler and the linker that the program will need to be linked to a library of routines that handle input from the keyboard and output to the screen (specifically the `cout` statement that appears later). The header file "`iostream`" contains basic information about this library. **Header files** are files that contain predefined values and routines, such as `cout`. Their filenames usually have no extension or end in `.h`. In order for your C++ program to use these predefined routines, you must include a **preprocessor directive**—a statement that tells a program, called the **preprocessor**, what to do before compiling the program. We will learn more about libraries of code later in this course.

After the include directives is the line:

```
using namespace std;
```

This statement is called a *using* directive. The latest versions of the C++ standard divide names (e.g. cout) into subcollections of names called *namespaces*. The basic role of namespaces is to arrange code into **logical** groups in order to avoid name conflicts and organise a large amount of code (such a library). Such conflicts can occur especially when your code base includes **multiple** libraries.

This particular *using* directive says the program will be using names that have a meaning defined for them in the `std` namespace (in this case the `iostream` header defines meanings for `cout` in the `std` namespace). Without the use of the `using` directive, the `std` prefix would have to appear in many program statements, e.g.

```
std::cout<<"Welcome to the C++ course"<<endl;
```

Some C++ compilers may not support namespaces. In this case you can use the older form of the include directive (that does not require a *using* directive, and places all names in a single global namespace):

```
#include <iostream.h>
```

Some of the code encountered in industry may be written using this older style for headers. All the programs written will start in this way.

C++ programs consist of modules called functions. In our first program there is a single function called `main()`.The use of functions is a basic building block of good design code in C++. Each statement within every C++ program is contained in a function; every function consists of two parts: a function header and a function body (more on this later). A C++ program may contain many functions, but every C++ program contains at least one function, and that function is called `main()`. It is special because when executing any program, this is the first function that is called.

The parenthesis `()`, which are not always empty, following `main` are the distinguishing features of a function. These will be studied in greater detail later. Braces surround the body of the function, and their role is to begin and end the function. We note the word `int` precedes the function name `main` (again predefined) and this indicates that the particular function returns a type that is integer to the operating system in which you are running the program – this is regarded as good programming practice.

The function body contains only two lines:

```
cout<<"Welcome to the C++ course"<<endl;
```

and the return statement

```
return 0;
```

The first line is simply displaying the message in quotes to the screen using the command `cout` (console output and read "C out") and is predefined in C++. (Text which is enclosed in a set of quotes is called a *string constant*). It is followed by the operator << called the ***insertion operator***.

The word `endl` has the effect of taking the cursor to the start of the next line, and is called the *endline manipulator*. It has the same effect as `'\n'`, and it is important that the direction of the slash (i.e. a **backslash**) is correct. A mistake at this point is not flagged by a compiler as a syntax error, and the result is simply a peculiar looking output.

When at the end of the main program, the line

```
return 0;
```

means "return the value 0 to the computer's operating system to signal that the program has completed successfully". In older versions of C++ we could give `main()` the return type of void and dispense with the return statement, but this is considered incorrect in standard C++.

More generally, *return statements* signal that the particular sub-program has finished, and return a value, along with the flow of control, to the program level above. We will look at this in greater detail later.

Because the program is short, it is easily packaged up into a single list of program statements and commands. After the include and using directives, the basic structure of any program is:

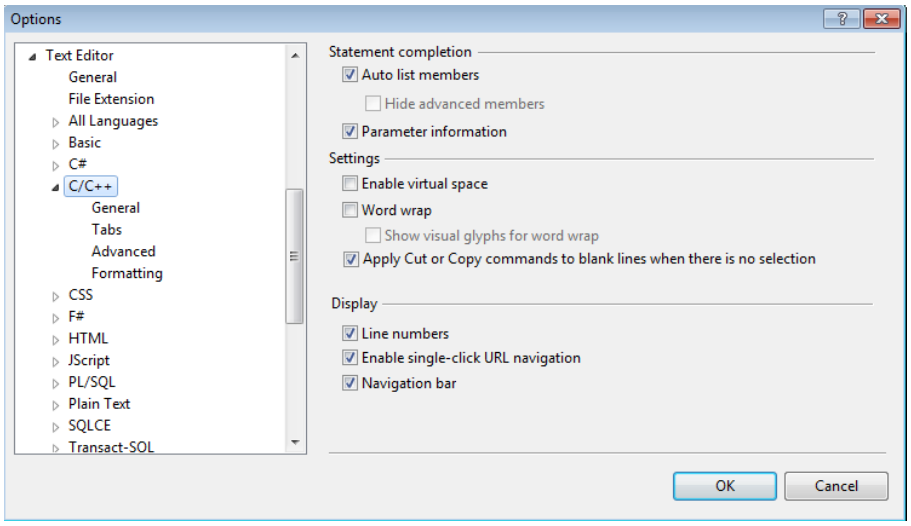
```
int main()
{
          First statement;
          ...
          ...
          Last statement;

          return 0;
}
```

All C++ programs have this basic "top-level" structure. Notice that each statement in the body of the program ends with a semicolon. In a well-designed large program, many of these statements will include references or calls to sub-programs (called functions), listed after the main program or in a separate file. These sub-programs have roughly the same outline structure as the program here, but there is always exactly one such structure called `main`.

Having line numbers in the code is often helpful. Many textbooks will rely on these to refer to specific sections of the code. In MS C++ these can added by following the given sequence of steps below

Tools Options$\Rightarrow$

Text Editor C/C++ On the right hand side under Display make sure Line numbers is ticked

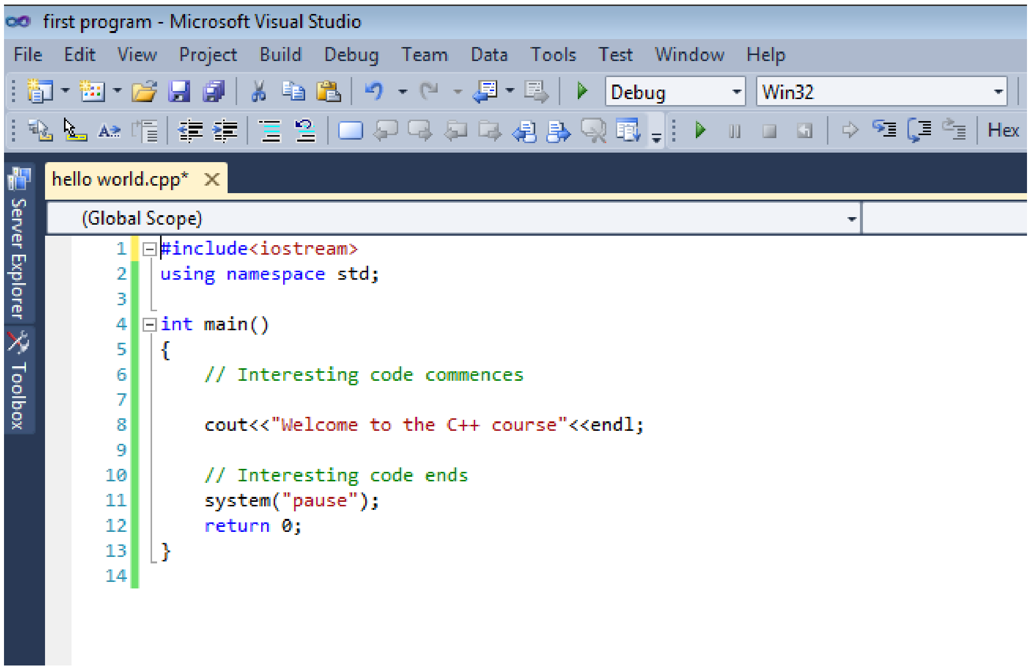


Statement completion
- ✓ Auto list members
- ☐ Hide advanced members
- ✓ Parameter information

Settings
- ☐ Enable virtual space
- ☐ Word wrap
- ☐ Show visual glyphs for word wrap
- ✓ Apply Cut or Copy commands to blank lines when there is no selection

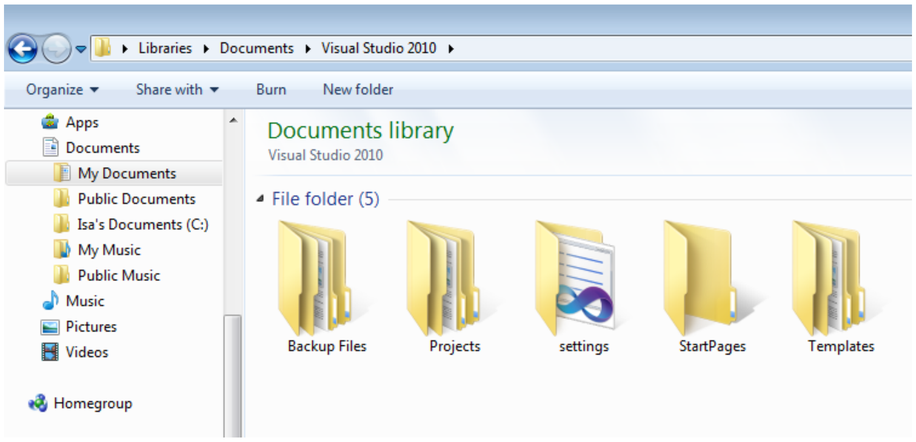
Display
- ✓ Line numbers
- ✓ Enable single-click URL navigation
- ✓ Navigation bar

The result is that the first program we typed now looks like


first program - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Test Window Help
Debug
Win32
Hex
Server Explorer
Toolbox
hello world.cpp*
(Global Scope)

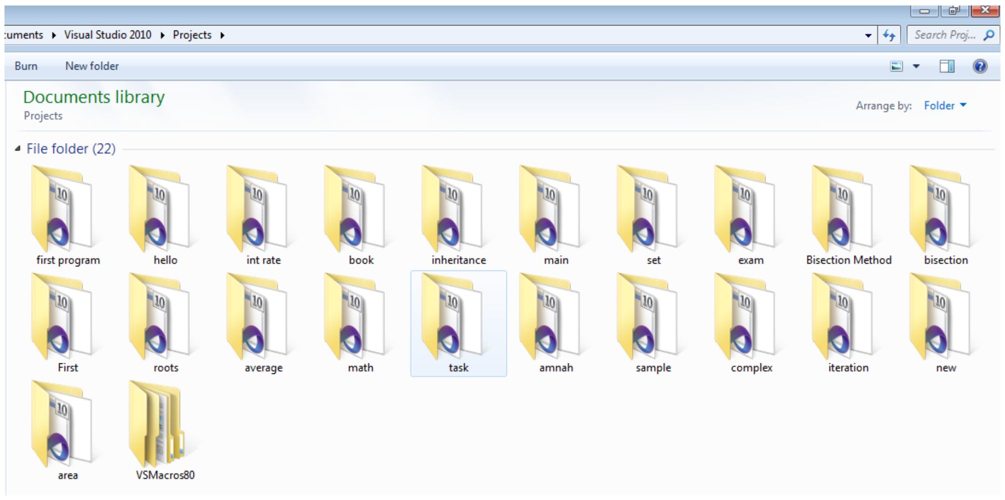
```
1   #include<iostream>
2   using namespace std;
3
4   int main()
5   {
6       // Interesting code commences
7
8       cout<<"Welcome to the C++ course"<<endl;
9
10      // Interesting code ends
11      system("pause");
12      return 0;
13  }
14
```

Note everything either side of the two comment statements – lines 1, 2, 12, 13. The first and last few lines of code are is called **boiler plate code** or simply **boilerplate**. This is a term for boring code that has to be included for technical reasons but which doesn't do much – it's an essential form of preamble. Having said that, without this inclusion the program will throw up errors at compile time. It is called "boiler plate" because boilers often come with a steel plate attached saying who made them and perhaps containing some warnings about how to operate the boiler. Nobody ever reads this, but it has to be there "for legal reasons". The same is true for the first few lines of all our code. These have nevertheless been discussed earlier.



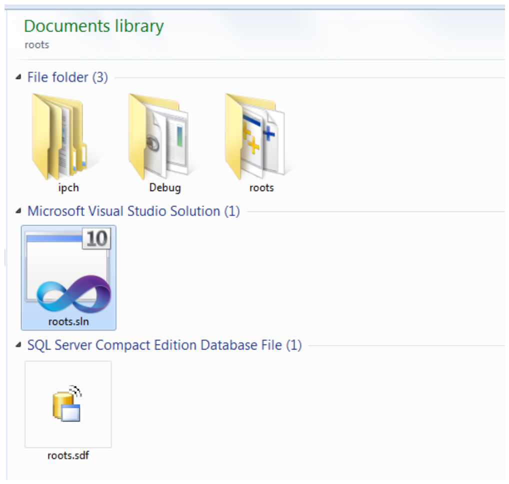
On a windows platform MS Visual Studio files are located in the Documents library. Here you will also find a folder containing all Projects that have been created.

Clicking on **Projects** lists all the C++ code files you have created. So for example on my drive I have the following list.



Each self-contained folder here comprises of a number of files associated with the project. Choosing for example,

roots gives the following view. The important icons at this stage are the folder named **roots** and the file below that called **roots.sln**


Documents library
roots
File folder (3)
ipch
Debug
roots
Microsoft Visual Studio Solution (1)
roots.sln
SQL Server Compact Edition Database File (1)
roots.sdf

Clicking on the .sln solution file opens up the project in MS Visual Studio and is ready to run. It has the advantage that all source and header filles will appear on the left hand side.

1.2 An Example C++ Program – Quadratic Equation

We have already seen how to output information to the screen using `cout` together with the insertion operator `<<`. Input can be accomplished from the keyboard using the predefined C++ word `cin` followed by ***extraction operator*** `>>`.

As a motivating example for mathematical applications, we consider a basic program that solves for the zeroes of $ax^2 + bx + c = 0$: The formula obtained from completing the square is $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. This code will be modified later to include the different cases of the discriminant $b^2 - 4ac$

```
/* This program solves a quadratic equation by prompting the user
for the coefficient values */

#include<iostream>
#include<cmath>

using namespace std;

int    main()
{
       float  a, b, c;  // 3 variables of type real.
       float  root; // float is a real type.
       float  x1,    x2;

       cout<<"Enter the values of a, b & c"<<endl; // Line 9
       cin>>a; // Line 10
       cin>>b;
       cin>>c;

       root= sqrt(b*b-4*a*c);

       x1=(-b+root)/(2*a);
       x2=(-b-root)/(2*a);

       cout<<"The roots are " << x1 << " and " <<x2<<endl;  // Line 16

       return 0;
}
```

We see a second header file "`cmath`", which contains standard (predefined) mathematical functions – in this case the square root function. This mathematics file will be essential for our work in applying C++ to solve mathematical problems (more later).

The user is prompted to input data at line 9. Data is then read in at lines 10 - 12. This can also be achieved using a more compact way, by cascading the operator, which is legal:

```
cin>>a>>b>>c;
```

The integers can now be read in as **`a b c`** or as
`a`
`b`
`c`

This idea is again used in line 16, where we cascade the insertion operator to format the output.
Our example program uses *six variables*: `a, b, c, root, x1, x2.`

Program variables are not like variables in mathematics, e.g. $y = f(x)$. They are more like symbolic names for "pockets of computer memory" which can be used to store different values at different times during the program execution. These variables are first introduced in our program in the variable declaration

```
    int   a, b, c;

    float root;

    float x1,   x2;
```

which signals to the compiler that it should set aside enough memory to store three variables of type "`int`" (integer) and three of type "`float`" (real) during the rest of the program execution. Hence variables should always be declared before being used in a program. Indeed, it is considered good style and practice to declare all the variables to be used in a program or sub-program at the beginning. Variables can be one of several different types in C++, and we will discuss variables and types at some length.

Note: Due to the simple construction of this C++ code which lacks any form of logical tests, when executing this program care should be taken to ensure the discriminant $b^2 - 4ac$ is greater than zero. Later logical statements will be used to consider all three possible cases of two real distinct roots; one repeated root; two complex conjugate roots.

1.3 Very Simple Input, Output and Assignment

After we have compiled the program above, we can *run* it. The program will prompt us for three real variables, `a, b, c.` If we enter

```
    1 5 -6
```

(we note that $b^2 - 4ac > 0$), or separate with return statement

```
1

5

-6
```

then the result will be something like

```
        The roots are 1 and -6
```

This line above is produced on the screen by the program. In general, the program statement

```
        cout << Expression1 << Expression2 << ... << ExpressionN;
```

will produce the screen output

```
        Expression1Expression2...ExpressionN
```

The series of statements

```
        cout << Expression1;
        cout << Expression2;
        ...
        ...
        cout << ExpressionN;
```

will produce an identical output. If spaces or new lines are needed between the output expressions, these have to be included explicitly, with a `"  "` or a `"\n"` respectively. The expression `endl` can also be used to output a new line, and in many cases is preferable to using `"\n"` since it has the side-effect of flushing the output buffer (output is often stored internally and printed in chunks when sufficient output has been accumulated; using `endl` forces all output to appear on the screen immediately).

The program statement `cout << '\n';` is equivalent to the code `cout << endl;`

A space can also be inserted using the *tab manipulator* `'\t'`, e.g. the program statement

```
cout<<"The root is" << '\t'<< x1 << endl;
```

will produce the output

```
The root is        1
```

(Note the tab inserted between the string and the value of the root x1).

Manipulators are operators which are used with the insertion operator << to modify or manipulate the way data is displayed. So both `'\n'` and `'\t'` are examples of manipulators.

Programs can also include assignment statements, a simple example of which is the statement

```
x1=(-b+root)/(2*a);
```

Hence the symbol = means "is assigned the value of". ("Equals" is represented in C++ as ==)

1.4 Example Mathematical Finance code – Compounding

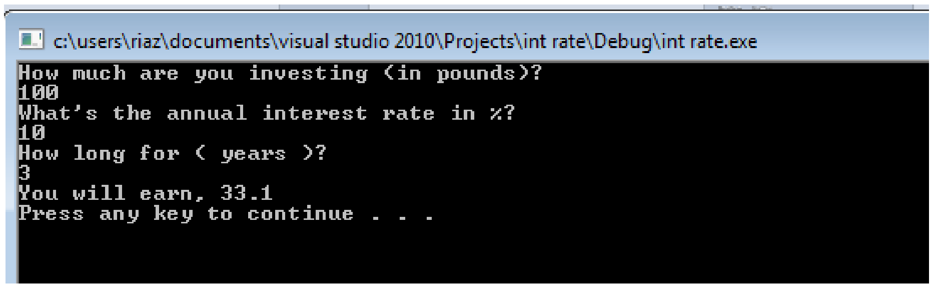
Now consider a fundamental but very important problem in finance

$$TV = PV(1+r)^N$$

where TV is the terminal (final) value of our initial investment PV for a given number of years N, receiving interest at the rate r. This can be calculated using the following simple program. Note the inclusion of the maths library which contains the power function where x^n is evaluated in C++ as `pow(x,n)`. In the code the interest accrued is calculated from $TV - PV$.

```
#include<iostream>
#include<cmath>
using namespace std;
int main(){

       int principal;
       double int_rate;
       int years;
       cout<<"How much are you investing (in pounds)? "<<endl;
       cin>>principal ;
       cout<<" What's the annual interest rate in %?"<<endl;
       cin>>int_rate ;
       cout<<" How long for ( years )?"<<endl;
       cin>>years;
       double finalBalance=
              pow(1.0+int_rate*0.01,years)*principal;
       double interest = finalBalance - principal;
       cout<<"You will earn, " <<interest<<endl;

       return 0;
}
```

With the following output


c:\users\riaz\documents\visual studio 2010\Projects\int rate\Debug\int rate.exe

```
How much are you investing (in pounds)?
100
What's the annual interest rate in %?
10
How long for ( years )?
3
You will earn, 33.1
Press any key to continue . . .
```

1.5 Preliminary Remarks about Program Style

The end of a line is unimportant to a C++ compiler, as it almost completely ignores *whitespace*. Whitespace is defined as spaces and tabs, and to the compiler are invisible. As far as the C++ compiler is concerned, the following program is exactly the same as the program in Section 1.2

```
#include<iostream>
#include<cmath>
using namespace std; int main() { float a, b, c;  floatroot; float x1, x2;
cout << "Enter the values of a, b & c" << endl; cin >> a; cin >> b; cin >> c;
root = sqrt(b*b - 4 * a*c); x1 = (-b + root) / (2 * a); x2 = (-b - root) / (2 * a);
cout << "The roots are " << x1 << " and " << x2 << endl;  return 0; }
```

However, the lack of *program comments*, *spaces*, *new lines* and *indentation* makes this program quite unacceptable, and very difficult to read. Whilst there is much more to developing a good programming style than learning to lay out programs properly, this is nevertheless a good start.

Be consistent with your program layout, and make sure the indentation and spacing reflects the logical structure of your program. In the MS C++ environment, indentation can be achieved by highlighting the required and typing Ctrl K, Ctrl F. Or from the menu select **Edit** , **Advanced** , **Format Selection** which has the same effect.

It is also a good idea to choose meaningful names for variables. Remember that your code might need modification by other programmers at a later date. Even if a programmer refers to their own code (say) a year later, without proper documentation and use of meaningful variable names, they will need to spend considerable time refreshing them self on the various parts of the code.

There are two important examples where whitespace is not ignored by the compiler:

1. The pre-processor directive `#include` must be written on one line

2. A string constant such as `"Welcome to the C++ course"` cannot be broken into separate lines. If a longer string is required, then it can be divided into two separate strings, each surrounded by quotes.

Although there is no general rule of thumb relating to programming style, the most important point to note is that one adheres to the style guidelines in their place you work (or those suggested by tutor) and that the style is applied in a consistent manner.

1.6 Identifiers

As we have seen, C++ programs can be written using many English words. Every programming language contains a few vocabulary words needed in order to use the language.

Reserved Words. These are **keywords** such as `if`, `int` and `else`, which have a predefined meaning that cannot be changed, i.e. they must never be used for any other purpose such as defining variables. There is a small variation, so a keyword in one compiler may not be a reserved word on another. The following list is a fairly complete one:

The reserved words used by C++ can be grouped into several categories.

- Data Types (the various types of data the computer can store):

char	**short**	**int**	**long**	(integer types)
enum				(enumerated type)
bool	**true**	**false**		(boolean type and values)
double	**float**			(real numbers)
void				(for functions that do not return values)
typedef				(for creating user defined types)

- Type modifiers (which control aspects of how the computer stores data):

```
signed   unsigned
const    static
```

- Flow of Control (which control the order of instruction execution):

if	**else**		(two-way branch)
switch	**case**	**default**	(multi-way branch)
for	**while**	**do**	(loops)
break	**continue**		(loop iteration control)
return			(immediate return from function)

- Dynamic memory allocation:

```
new       delete
```

- Object-oriented keywords:

```
class
private    protected  public
virtual
this
friend
template
operator
```

Namespace control:

```
using      namespace
```

- Miscellaneous:

```
sizeof     inline
```

1. Library Identifiers: These words are supplied default meanings by the programming environment, and should only have their meanings changed if the programmer has strong reasons for doing so. Examples are `cin`, `cout` and `sqrt` (square root).
2. Programmer-supplied Identifiers. These words are "created" by the programmer, and are typically variable names, such as `root` and `x1`.

An identifier cannot be any sequence of symbols. A valid identifier must start with a letter of the alphabet (upper or lower case) or an underscore ("_") and must consist only of letters, digits, and underscores. Note that C++ is case sensitive.

In the following example it is assumed that all declarations are part of the same program.

```
int sum;    // legal

int SUM;    // legal

int _sum;   // legal

int ruOK2;  // legal

int 3KO;    // not allowed - identifier cannot start with a digit

int large sum; // not allowed - gaps are illegal

int class;   // not allowed - "class" is a reserved word

int sum;    // not allowed - identifier redefinition

int small.sum // not allowed - dot cannot be used
```

The following table is list of key reserved words in C++.

and	continue	if	public	try
and_eq	default	inline	register	typedef
asm	delete	int	reinterpret_cast	typeid
auto	do	long	return	typename
bitand	double	mutable	short	uchar_t
bitor	dynamiccast	namespace	signed	union
bool	else	new	sizeof	unsigned
break	enum	not	state_cast	using
case	explicit	not_eq	static	virtual
catch	extern	operator	struct	void
char	false	Or	switch	volatile
class	float	or_eq	template	wchar_t
compl	for	overload	this	while
const	friend	private	throw	xor
constcast	goto	protected	true	xor_eq

1.7 Data Types

1.7.1 Integers

C++ requires that all variables used in a program be given a data type. We have already seen the data type `int`. Variables of this type are used to represent integers (whole numbers).

Declaring a variable to be of type `int` signals to the compiler that it must associate enough memory with the variable's identifier to store an integer value or integer values as the program executes. But there is a (system dependent) limit on the largest and smallest integers that can be stored. Hence C++ also supports the data types `short int` and `long int` which represent, in turn, a smaller and a larger range of integer values than `int`. Adding the prefix `unsigned` to any of these types means that you wish to represent non-negative integers only. For example, the declaration

```
        unsigned short int a, b, c;
```

reserves memory for representing three relatively small non-negative integers.

Some rules have to be observed when writing integer values in programs:

1. Decimal points cannot be used; although 26 and 26.0 have the same value, "`26.0`" is not of type "`int`".
2. Commas cannot be used in integers, so that (for example) 23,897 has to be written as "`23897`".
3. Integers cannot be written with leading zeros. The compiler will, for example, interpret "011" as an octal (base 8) number, with value 9.

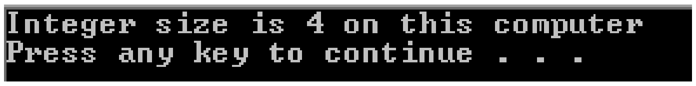
Type	**Size in bytes**	**Range of possible values**
`char, signed char`	1	-128 to 127
`int, signed int, short, signed short`	2	-32768 to 32767
`int`	4	-2,147,483,648 to 2,147,483,647
`long`	4	-2,147,483,648 to 2,147,483,647
`unsigned char`	1	0 to 255
`unsigned short int`	2	0 to 65535
`unsigned int`	4	0 to 4,294,967,295

`unsigned long`	4	0 to 4,294,967,295

The size of variables can be determined on the system being used with the `sizeof()` operator. So for example to find out much memory an integer uses, the following code

```
cout<<"Integer size is "<< sizeof(int) <<" on this computer"<< endl;
```

will produce the output



```
Integer size is 4 on this computer
Press any key to continue . . .
```

Symbol	Operation	Example	Value
`+`	Addition	`3+5`	`8`
`-`	Subtraction	`43-25`	`18`
`*`	Multiplication	`4*7`	`28`
`/`	Division	`9/2`	`4`
`%`	Modulus	`20%6`	`2`
`abs`	Absolute	`abs(-3)`	`3`

1.7.2 Real numbers

Variables of type "`float`" are used to store real numbers. Plus and minus signs for data of type "`float`" are treated exactly as with integers, and trailing zeros to the right of the decimal point are ignored. Hence "`+523.5`", "`523.5`" and "`523.500`" all represent the same value.

The computer also accepts real numbers in *floating-point* form (or "scientific notation"). Hence 523.5 could be written in scientific format as "`5.235e+02`" (i.e. 5.235 x 10^2), and -0.0034 as "`-3.4e-03`". The "`e`" is called an *exponent* and this format is also called *exponential notation*. In addition to "`float`", C++ supports the types "`double`" and "`long double`", which give increasingly precise representation of real numbers, but at the cost of more computer memory.

Type	Size in bytes	Range of possible values	Digits of Precision
`float`	4	$3.4x10^{-38}$ to $3.4x10^{38}$	7
`double`	8	$1.7x10^{-308}$ to $1.7x10^{308}$	15
`long double`	10	10^{-4932} to $1.7x10^{4932}$	19

1.7.3 Type Casting

Sometimes it is important to guarantee that a value is stored as a real number, even if it is in fact a whole number. A common example is where an arithmetic expression involves division. When applied to two values of type `int`, the division operator "`/`" signifies integer division, so that (for example) `7/2` evaluates to 3. In this case, if we want an answer of 3.5, we can simply add a decimal point and zero to one or both numbers - "`7.0/2`", "`7/2.0`" and "`7.0/2.0`" all give the desired result. However, if both the numerator and the divisor are variables, this trick is not possible. Instead, we have to use a type cast. For example, we can convert "`7`" to a value of type `double` using the expression "`double(7)`". Hence in the expression

```
        answer = double(numerator) / denominator
```

the "`/`" will always be interpreted as real-number division, even when both "`numerator`" and "`denominator`" have integer values. Other type names can also be used for type casting. For example, "`int(14.35)`" has an integer value of 14.

1.7.4 Characters

Variables of type "`char`" are used to store character data in the form of integers. In standard C++, data of type "`char`" can only be a single character (which could be a blank space). These characters come from an available character set which can differ from computer to computer. However, it always includes upper and lower case letters of the alphabet, the digits `0`, ... , `9`, and some special symbols such as `#`, `£`, `!`, `+`, `-`, etc. Perhaps the most common collection of characters is the ASCII character set.

Type	**Size in bytes**	**Range of possible values**
`char`	1	-128 to 127

Most windows system extend to 255 to cater for foreign languages and graphics characters.

Character constants of type "`char`" must be enclosed in single quotation marks when used in a program, otherwise they will be misinterpreted and may cause a compilation error or unexpected program behaviour. For example, "`'A'`" is a character constant, but "`A`" will be interpreted as a program variable. Similarly, "`'9'`" is a character, but "`9`" is an integer.

There is, however, an important (and perhaps somewhat confusing) technical point concerning data of type "`char`". Characters are represented as integers inside the computer. Hence the data type "`char`" is simply a subset of the data type "`int`". We can even do arithmetic with characters. For example, the following expression is evaluated as true on any computer using the ASCII character set:

'9' - '0' == 57 - 48 == 9

The ASCII code for the character '9' is decimal 57 (hexadecimal 39) and the ASCII code for the character '0' is decimal 48 (hexadecimal 30) so this equation is stating that

57(dec) - 48(dec) == 39(hex) - 30(hex) == 9

It is often regarded as better to use the ASCII codes in their hexadecimal form.

However, declaring a variable to be of type "`char`" rather than type "`int`" makes an important difference as regards the type of input the program expects, and the format of the output it produces. For example, the program

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

int main()
{
	int number;
	char character;

	cout << "Type in a character:\n";
	cin >> character;
```

```
        number = character;

        cout << "The character '" << character;
        cout << "' is represented as the number ";
        cout << number << " in the computer.\n";

        return 0;
}
```

produces output such as

```
        Type in a character:
        9
        The character '9' is represented as the number 57 in the
computer.
```

Some programmers pronounce `char` as “care” because it comes from “character”. Others pronounce it as “char” as in “scorch” , while some say “car” as in automobile.

1.7.5 Constants

The data items encountered so far have all been variables. Another useful type is the constant. This is defined in a similar way to a variable, except that the definition is preceded by the word const, and it must always be initialised, e.g.

```
const float pi = 3.14159;
```

The syntax for defining a constant of any type is

```
const <type> name;
```

so for example in a mechanics type computation we could define the constant acceleration due to gravity which is a real number as

```
const float gravity = 9.81;
```

1.7.6 Enumerations

Constants of type "`int`" may also be declared with an enumeration statement. For example, the declaration

```
        enum { MON, TUES, WED, THURS, FRI, SAT, SUN };
```

is shorthand for

```
        const int MON   = 0;
        const int TUES = 1;
        const int WED = 2;
        const int THURS = 3;
        const int FRI = 4;
        const int SAT = 5;
        const int SUN = 6;
```

By default, members of an "`enum`" list are given the values 0, 1, 2, etc., but when "`enum`" members are explicitly initialised, uninitialised members of the list have values that are one more than the previous value on the list (so the change in value is +1):

```
        enum Day { MON = 1, TUES, WED, THURS, FRI, SAT = -1, SUN };
```

In this case, the value of "`FRI`" is 5, and the value of "`SUN`" is 0.

Another example: Roman numerals

```
enum Roman{ I = 1, V=5, X=10, L=50, C=100, D=500, M=1000};

cout<< L+C << endl;  // this gives 150
```

C++ is based largely on modelling real-world problems and hence central to this language is the ability to create new data types and objects which help describe the problem being solved. The examples shown of type enum introduce this feature.

1.7.7 Further examples

1. enum Monthlength{Jan=31, Feb=28, Mar=31, Apr=30, ... , Dec=31};

Hence more than one constant can be assigned the same integer value.

2. enum Number{one=17, two, three, four = -3, five};

Then two=18, three=19, five= -2

1.7.8 Boolean Values

In subsequent chapters we will look more closely at branch and loop statements such as "for" and "while" loops and "if ... else" statements. All these constructs involve the evaluation of one or more logical (or "Boolean") expressions, and so we begin by looking at different ways to write such expressions.

C++ represents "True" as the integer 1, and "False" as 0. However, expressions such as

```
condition1 == 1
```

or

```
condition2 == 0
```

are not particularly clear – a more intuitive way would be to write

```
condition1 == True
```

and

```
condition2 == False
```

Furthermore, it is desirable to have a separate type for variables such as "`condition1`", rather than having to declare them as of type "`int`".

We can achieve all of this with a *named enumeration*:

```
        enum Logical {False, True}
```

which is equivalent to

```
        enum Logical {False = 0, True = 1}
```

This line acts as a kind of *type definition* for a new data type "`Logical`", so that lower down the program we can add variable declarations such as:

```
Logical condition1, condition2;
```

Indeed, we can now use the identifier "`Logical`" in exactly the same way as we use the identifiers "`int`", "`char`", etc.

Note: The Identifiers "**`true`**" and "**`false`**" in C++

Note that C++ implicitly includes the named enumeration

```
enum bool {false, true};
```

So you cannot (re)define the all-lower-case constant identifiers "`true`" and "`false`" for yourself.

In addition, you can use the type `bool` in the same way as we used `Logical` in our example.

1.8 The `typedef` construct

This is an excellent example of how C++ allows us to create new data types from existing ones. Suppose we wish to define many variables of type `unsigned short int` throughout a program (rather long!). The `typedef` construct allows us to express this in shorthand notation (whilst retaining a meaningful name). The program statement

```
typedef unsigned short int ushort;
```

creates a new type `ushort` which can now be used to define variables which are `unsigned short int`, so the code

```
ushort a, b, c;
```

sets aside memory for three (unsigned) short integers with names a, b and c. Once defined in this way, the type `ushort` can now be used throughout the program with this meaning. We will see later in the section on arrays, how this is very useful for defining matrices.

1.9 The Maths File `<cmath>`

All our work will involve solving mathematical problems arising in applied mathematics, which have applications in quantitative finance. Hence we will be frequently using mathematical functions. Including the maths library can access these. The first table lists trigonometric functions:

Function(s)	Return value	Example code
`acos`	arccosine	`acos(x)`
`asin`	arcsine	`asin(x)`
`atan2`	arctangent of *y/x*	`atan2(y,x)`
`cos`	cosine	`cos(x)`
`cosh`	hyperbolic cosine	`cosh(x)`
`sin`	sine	`sin(x)`
`sinh`	hyperbolic sine	`sinh(x)`
`tan`	tangent	`tan(x)`
`tanh`	hyperbolic tan	`tanh(x)`

The next contains a miscellany of commonly used basic functions:

Function(s)	Return value	Example
`abs`	absolute value of integer parameter	`abs(-23)=23`
`ceil`	smallest integer which is greater than parameter	`ceil(3.14159)=4`
`exp`	Exponential of *x* - e^x	`exp(x)`

`fabs`	absolute value of floating number	`fabs(-1.45)=1.45`

`floor`	the largest integer that is less than or equal to *x*	`floor(3.14159)=3`
		`floor(-3.8)=-4`
		`floor(-2.3)=-3`

`labs`	absolute value of a long integer parameter	`labs(-21500000)`
`log`	natural logarithm (base *e*) - $\ln x$	`log(x)`
`log10`	logarithm base 10	`Log10(100)=2`
`pow`	numeric power - x^a	`pow(x,a)`
`sqrt`	square root of x	`sqrt(x)`
`fmod`	Computes remainder of division	`fmod(12.0/7)=5`

The following example code shows some of the uses:

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
        const double pi=4*atan(1.0); // using tan(pi/4)=1.0

        cout<<"ceil(pi) = "<<ceil(pi)<<endl;
        cout<<"floor(pi)= "<<floor(pi)<<endl;
        cout<<"sqrt(pi) = "<<sqrt(pi)<<endl;

        cout<<"pow(pi, 0.5)= "<<pow(pi, 0.5)<<endl;
        cout<<"log(pi)               = "<<log(pi)<<endl;

        cout<<"exp(pi)       = "<<exp(pi)<<endl;
        cout<<"log10(pi)= "<<log10(pi)<<endl;
        cout<<"sin(pi/2)= "<<sin(pi/2)<<endl;
        cout<<"cos(pi/2)= "<<cos(pi/2)<<endl;


        return 0;}
```

The output is

```
        ceil(pi)            = 4
        floor(pi)           = 3
        sqrt(pi)            = 1.77425
        pow(pi, 0.5)        = 1.77425
        log(pi)             = 1.14473
        exp(pi)             = 23.1407
        log10(pi)           = 0.49715
        sin(pi/2)           = 1
        cos(pi/2)           = 6.12303e-017
```

More Operators

Symbol	Operation	Example	Value

		`int a=3`	
`++`	Pre-increment	`cout << ++a`	4
`++`	Post-increment	`cout << a++`	3
`--`	Pre-decrement	`cout << --a`	2
`--`	Post-decrement	`cout << a--`	3

Examples – in general

```
a=b++;      // a is assigned the value of b and then b is incremented
x=++y;      // y is incremented and then x is assigned the new value of y
a=b--;      // a is assigned the value of b and then b is decremented
x=--y;      // y is decremented and then x is assigned the new value of y
```

Operation	**Equivalent**	**Example** `int a=6; int b=3`
`a+=b`	`a=a+b`	`a=9`
`a-=b`	`a=a-b`	`a=3`
`a*=b`	`a=a*b`	`a=18`
`a/=b`	`a=a/b`	`a=2`

1.10 Initialisation

This is an essential part of responsible and safe coding. Each time a new variable is declared, it should be initialised at the time of declaration. So for example instead of producing the following statement

```
float x;
```

write

```
float x=0;
```

This ensures that when the variable is created and memory set aside for a float, x is initially assigned the value zero instead of some random real number which the programmer has no control over. The main part of the earlier quadratic program looks like

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

int    main()
{
       float  a=0, b=0, c=0;
       float  root=0;
       float  x1=0,  x2=0;
       cout<<"Enter the values of a, b & c"<<endl;
       cin>>a;
       cin>>b;
       cin>>c;
       root= sqrt(b*b-4*a*c);
       x1=(-b+root)/(2*a);
       x2=(-b-root)/(2*a);
       cout<<"The roots are " << x1 << " and " <<x2<<endl;
       return 0;
}
```

1.11 Debugging

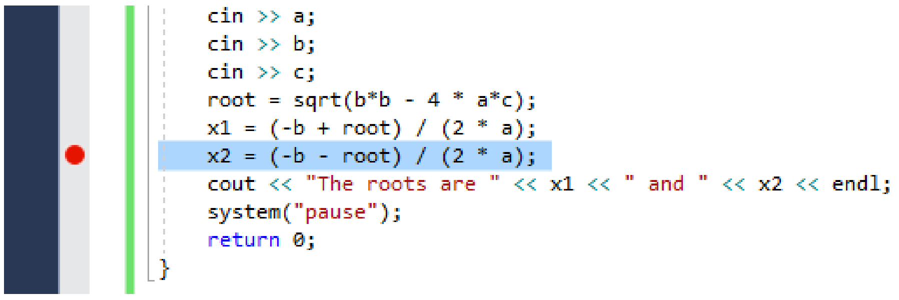
1.11.1 Introduction

In software development, writing code often requires less time than fixes bugs; even correcting syntax errors can be a relatively straight forward task but there is no guarantee that the program will work correctly or as the coder expects; it simply means that the program will compile. The most arduous and time consuming component can be testing and defect fixing – this can require more time than the writing of the program itself. The process of identifying the cause of errors in the logic and design and subsequent fixing is termed as *debugging*. In informal terms, it's the process of finding bugs in your code and reducing them! If the program is particulalrly badly written, the best option is to 'bin' it and start again!

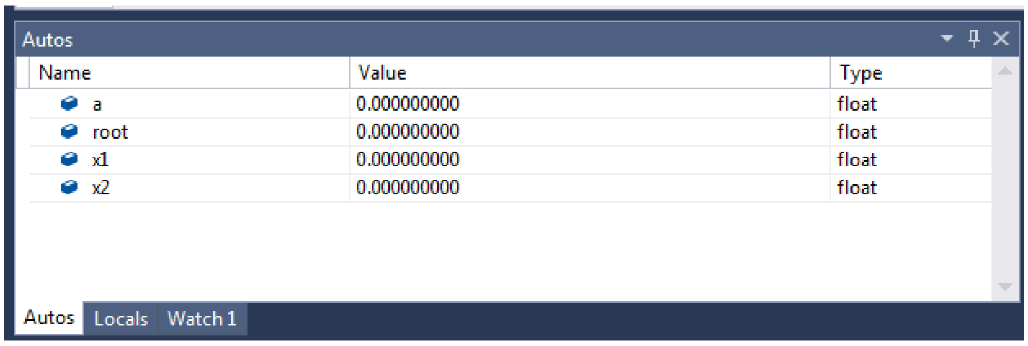
The software suite that consolidates the basic tools developers need to write and test software is called an integrated development environment (**IDE**). MS Visual Studio IDE provides a user-friendly environment with applications and tools. More importantly, all in a windows based setting!

1.11.2 How to start?

Easiest way to start is to either press F5 or from the Debug Menu, select "Start Debugging". If you have placed breakpoints in your code, then execution will begin automatically. To insert a breakpoint on a line, place cursor in the grey vertical pane on the left hand side, left click and a red circle appears. F11 steps over each line.



```
    cin >> a;
    cin >> b;
    cin >> c;
    root = sqrt(b*b - 4 * a*c);
    x1 = (-b + root) / (2 * a);
    x2 = (-b - root) / (2 * a);
    cout << "The roots are " << x1 << " and " << x2 << endl;
    system("pause");
    return 0;
}
```



Name	Value	Type
a	0.000000000	float
root	0.000000000	float
x1	0.000000000	float
x2	0.000000000	float

1.12 Review Exercises

This section is to

1. The symbol used with the cout object is called the ______________ operator .
2. Which of the following declarations are legal (this is part of the same program code)?

   `int var1; int VAR1;int _var;    int small sum;     int 4U;`

   `int small.sum ; int big_sum; float rUOK2; float _var`
3. How can both short and long comments be included in C++ programs?
4. What must a function name be followed by?
5. What is used to enclose the code in a function?
6. What kind of program elements are the following?
   a. 23
   b. **'a'**
   c. 3.14159
   d. foo()
7. Write statements that display on the screen (1 line for each)
   a. The character 'x'
   b. The name *Snaevar*
   c. The number 283
8. What header file must you **#include** with your source code to read from the keyboard using cin?
9. What does the expression 11%3 evaluate to?
10. What does the expression 14/4 evaluate to?
11. The increment operator increases the value of a variable by how much?
12. Assuming **var1** starts with the value 20, what will the following code fragment print out?

```
cout<<var1--<<endl;
cout<<++var1<<endl;
```

In the following exercises where you are asked to predict the precise output, perform a dry run on paper first and then to check your output copy and paste the source code in to a C++ project and run.

1. What is the exact output of the program below? Indicate a blank space in the output by writing the symbol _ . Indicate a blank line in the output by writing ____________________ .

```
#include<iostream>
using namespace std;
int main()
{
       int n = 8, k = 3;
       cout << ++n << endl;
       cout << n << endl;
       cout << n++ << endl;
       cout << n << endl;
       cout << -n << endl;
       cout << n << endl;
       cout << --n << endl;
       cout << n << endl;
       cout << n-- << endl;
       cout << n << endl;
       cout << n + k << endl;
       cout << n << endl;
       cout << k << endl;
       cout << n << k << endl;
       cout << n << endl;
       cout << " " << n << endl;
       cout << " n" << endl;
       cout << "\n" << endl;
       cout << " n * n = ";
       cout << n * n << endl;
       cout << 'n' << endl;

       return 0;
}
```

2. What does the following produce

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

       int main()
       {
              int n;
              cout << (n = 4) << endl;
              cout << (n == 4) << endl;
              cout << (n > 3) << endl;
              cout << (n < 4) << endl;
              cout << (n = 0) << endl;
              cout << (n == 0) << endl;
              cout << (n > 0) << endl;
              cout << (n && 4) << endl;
              cout << (n || 4) << endl;
              cout << (!n) << endl;
```

```
        system("pause");
        return 0; }
```

3. What is the output of the program below?

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

int main()
{
       enum color_type {red, black, white, green, blue, yellow};
 color_type shirt, pants;
 shirt = red;
 pants = yellow;
 cout << shirt << '\t' << pants << endl;
 cout << shirt << " " << pants << endl;


       system("pause");

       return 0; }
```

What is the <u>precise</u> difference between the two output lines?

4. What does the following portion of code print to the screen when it is run?

```
int i = 5, j = 6, k = 7, n = 3;
cout << i + j * k - k % n << endl;
cout << i / n << endl;
```

5. What happens when the following fragment of code is executed?

```
int n = 3;
       while (n >= 0)
       {
              cout << n * n << endl;
              --n;
       }
       cout << n << endl;
       while (n < 4)
              cout << ++n << endl;
       cout << n << endl;
       while (n >= 0)
              cout << (n /= 2) << endl;
```

1.13 Exercises

1. Write C++ code to output the following to the screen

   I am learning C++

   I'm really looking forward to C++!

2. Write a program to declare three variables – two integers and one float.
   a. Assign the to them the values 3, 12, 3.14 respectively and print to the screen.
   b. Now ask the user to enter values of their choice and output to the screen.
   c. Change the third variable to the product of the first two.
3. Write a program that returns the Celsius value for a given temperature measured in Fahrenheit. The relation between these two is given by $5(F-32)=9C$. As an example, the input 50 returns 10.

4. The time period *T* for a simple pendulum can be determined from $T=2\pi\sqrt{\frac{l}{g}}$, where *l* is the length of string and *g* is acceleration due to gravity (9.81 ms^{-2}). Write a program, which calculates values of T for varying lengths of string. Recall the result: $\tan(\pi/4)=1.$

5. Write a program that calculates the current value of future cash flows. For example consider the following investments:
   1 million pounds in 3 years time with constant interest rate of 5%
   100,000 pounds in 1.5 years with a variable rate of interest given by
   $$r=\begin{cases}4\% & \text{first } 6 \text{ months}\\ 6.2\% & \text{thereafter}\end{cases}$$
   Use both discrete and continuous compounding.
6. Experiment with different mathematical functions in the `cmath` library by writing programs for various numerical expressions.

2. Control Structures, Flow and Output

2.0 Introduction

Most programs do not execute their statements in strict order, but do so in response to changing circumstances. Program statements that cause jumps from one part of a program to another, depending on the calculations performed are called control statements.

In this section we introduce the idea of control statements and decision making, which form two main categories, i.e. loops and decisions. This requires an understanding of logical expressions, which involve use of a type of operator called a relational operator – which compares two values. The result of the comparison is true or false, and any subsequent calculation/program execution will depend upon this. The following basic comparison/relational operators are given.

Symbol	**Meaning**	**Example**	**Value**
<	less than	3 < 5	TRUE
<=	less than or equal to	43 <= 25	FALSE
>	greater than	4 > 7	FALSE
>=	greater than or equal to	9 >= 2	TRUE
= =	equal to	20 == 6	FALSE
!=	not equal to	20 != 6	TRUE

2.1 Blocks and Scope

A *block* is a sequence of statements enclosed in braces {}. It can be used wherever a single statement can be used. When used in place of a single (or zero) statement it is also called a *compound statement.*

Example

Enclosing the statements within braces forms a statement block:

```
{       int n;
        cin>>n;
        cout<<2*n<<endl;
}
```

The variables declared in a block have the block as their *scope*. This means that variables are created/destroyed upon entering/exiting the block. If the same identifier is used to define a variable both within the block and outside, then the variables are unrelated. Inside the block, the program will assume by default that the identifier refers to the inner variable. It only looks outside the block for the variable if it fails to find a variable declaration within the block. Consider the following program which contains three blocks, of which one serves as the main body of the program, with the other two nested within `main()`.

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

int main()

{ // block 1 begins

        int x=22;

        { // block 2 begins
               int x=44;
               cout<<"Integer x = "<<x<<endl;
        } // block 2 ends

        cout<<"Integer x = "<<x<<endl;

        { // block 3 begins
               int x=66;
               cout<<"Integer x = "<<x<<endl;
        } // block 3 ends

        cout<<"Integer x = "<<x<<endl;

        return 0;
} // block 1 ends
```

The corresponding output is

```
Integer x = 44
Integer x = 22
Integer x = 66
Integer x = 22
```

The justification for the use of variables local to a block is that it saves on memory, or because it releases an identifier for re-use in another part of the program.

Example

What is the output from the following program?

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

int    integer1 = 1;
int    integer2 = 2;
int    integer3 = 3;

int    main()
{
       int    integer1 = -1;
       int    integer2 = -2;
       {
              int    integer1 = 10;
              cout <<       "integer1 = " << integer1 << endl;
              cout <<       "integer2 = " << integer2 << endl;
              cout <<       "integer3 = " << integer3 << endl;
       }
       cout <<       "integer1 = " << integer1 << endl;
       cout <<       "integer2 = " << integer2 << endl;
       cout <<       "integer3 = " << integer3 << endl;



       return 0;
}
```

2.2 The `if` and `if...else` Statements

The simplest way of making a decision in C++ is to use the `if` statement, and is also one of the most commonly used control flow statement.

The `if` statement is used for conditional execution, and its syntax is

```
if (condition)
{

         Statement1
                  .       // statements for a true condition
                  .
                  StatementN

}
else
{

         StatementN+1
                  .                         // statements for a false
                  .                         // condition
                  StatementN+M
}
```
where `condition` is a logical expression such as those given in the previous table.

Example: `if` and `if...else` statement - Using the modulus operator (%) we determine whether an integer is odd or even.

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

int main()
{
       int n=0;
       cout << "type in a non-zero integer value"<<endl;
       cin >> n;

       if (n%2==0) // tests if integer n is divisible by 2
       {
              cout<<n<<": is an even number"<<endl;
       }

       else  // else if odd …
       {
              cout <<n<<": is an odd number"<<endl;
       }
       return 0;
}
```

Note: if we were just trying to ascertain whether the integer was even, the `else` statement, together with braces (and output statement) would be discarded, to give an `if` statement (only).

Example: The quadratic program revisited

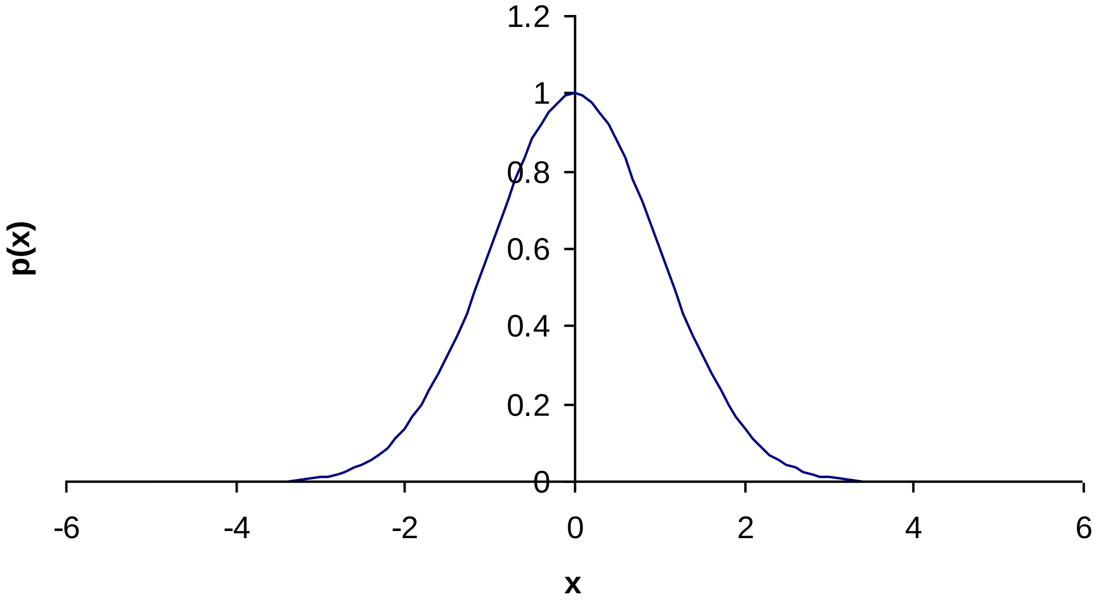
In 1.2 we looked at a simple program to solve the quadratic equation $ax^2 + bx + c = 0$, with careful choice of values for *a*, *b* and *c* to ensure a real solution. By use of the `if...else` statement we now can extend that program to consider all possible cases, i.e. in addition to $b^2 - 4ac > 0$ we may now also compute the situations of $b^2 = 4ac$ and complex roots given by $b^2 - 4ac < 0$.

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

double a,b,c;
double d, x1, x2, x;

int main()

{

       cout << "Enter values of a, b & c in turn\n";
       cin >> a;
       cin >> b;
       cin >> c;

       d= (b*b - 4*a*c);


       if (a==0) //simple case of a linear equation when a=0
       {

              x=-c/b;

              cout << "One root exists = " << x<<endl;
       }

       else

              if (d>0) // 2 real distinct roots
              {

                     x1=(-b +sqrt(d))/(2*a);
                     x2=(-b-sqrt(d))/(2*a);

                     cout << "There are two roots, x=" << x1 << " & " << x2 <<endl;

              }

              else

                     if (d==0) // 1 real b*b-4ac=0
```

```
                    {

                           x=-b/(2*a);

                           cout << "One root exists = " << x<<endl;

                    }
                    else

                           if (d<0) // pair of complex conjugate roots p±iq

                           {
                                  cout << "This equation has complex roots\n";

                                  double real=-b/(2.0*a); // this is p
                                  double im=sqrt(fabs(d))/(2.0*a); /* now calculate
q – coefficient of

        imaginary part*/


                                  cout<<"x1 = "<<real<<"+i*"<<im<<endl; // = p+iq
                                  cout<<"x2 = "<<real<<"-i*"<<im<<endl; // =p-iq

                           }
                           return 0;
}
```

2.3 Approximating a Cumulative Distribution Function (CDF) – First Numerical Example

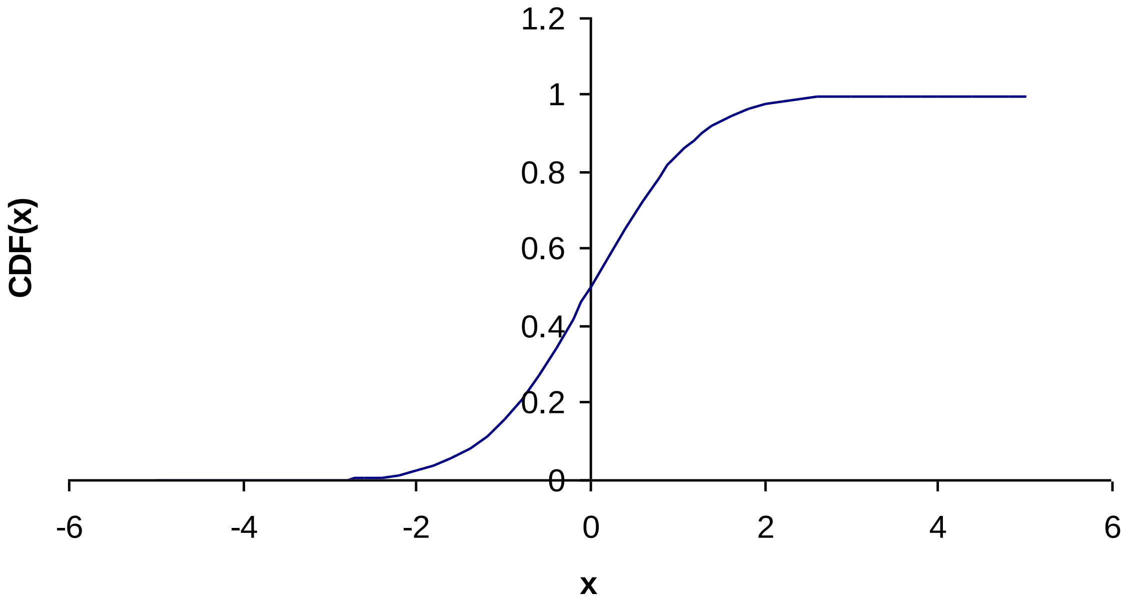
A random variable *X* which is normally distributed with zero mean and unit variance has a probability density function (pdf) $p(x)$ given by $p(x)=\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}x^2\right)$



To approximate the CDF for the Normal Distribution $N(x)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x}\exp\left(-\frac{1}{2}s^2\right)ds$, where $N(x)$ is the probability that the random variable X lies in the region bounded by $-\infty$ and the point X in the diagram above, i.e. the area under the curve to the left of the line $y=x$:

$$P(-\infty < X \le x) = N(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x}\exp\left(-\frac{1}{2}s^2\right)ds$$

The integral above cannot be simplified analytically (for a closed form solution) and requires computational treatment. There are a number of numerical schemes available and we present one such technique – a polynomial approximation.



We can approximate this definite integral by using the numerical scheme which is accurate to 6 decimal places; obtained from Abramowitz and Stegun (1974).

$$N(x) = \begin{cases} 1 - n(x)\left(a_1 k + a_2 k^2 + a_3 k^3 + a_4 k^4 + a_5 k^5\right) & x \geq 0 \\ 1 - N(-x) & x < 0 \end{cases}$$

where $k = \dfrac{1}{1 + 0.2316419x}$ and

$$a_1 = 0.319381530,\ a_2 = -0.356563782,\ a_3 = 1.781477937,$$
$$a_4 = -1.821255978,\ a_5 = 1.330274429$$

$$\text{and } n(x) = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}x^2\right)$$

The code for this is as follows:

```
#include<iostream>
#include<cmath>

using namespace std;

const double pi=4.0*atan(1.0); //define constant pi=3.142

int main()
{
       const double  a1=0.319381530, a2=-0.356563782, a3=1.781477937,
              a4=-1.821255978, a5=1.330274429; // define all constants a(i)

       double X=0, x=0; // X is the random variable, & x its absolute value
       double k=0;
       double N, CDF, n;

       cout<< "Enter the value of the random variable X"<<endl;
       /* upper limit of integral*/

       cin>>X;

       x=fabs(X);  //we set x to be the absolute value of the R.V. X

       k=1/(1+0.2316419*x);

       n=(1/sqrt(2*pi))*exp(-0.5*x*x);
       N=1-n*(a1*k+a2*k*k+a3*pow(k,3)+a4*pow(k,4)+a5*pow(k,5));

       CDF=N;

       if (X<0) /* we calculate for +ve X and then use the symmetry property
               of the distribution to obtain the CDF for –ve values*/
               CDF=1-N;

       cout<< CDF <<endl; //output value of N(x)


       return 0;
}
```

Defining such approximations can be done in a simple fashion. At a basic level, use of Taylor series followed by integrating the expansion gives simple polynomial expansions.

More Logical Operators (`&&` and `||`)

We demonstrate the use of the "`&&`" (and) and "`||`" (or) operators by way of example. These are used when we have more than one condition. For example, if we enter the integer 3, the following code:

```
int x;
cin>>x;
              if (x>0 && x<10)

              {
                   cout<< x <<" is an integer in the open interval between
                   0 and 10."<<endl;
              }
```

will output :

3 is an integer in the open interval between 0 and 10.

Both conditions must be satisfied, for the message to be outputted.

The following conditional expression

```
if (x<0 || x>=10)
{
statement;
}
```

will test whether the variable x is negative valued **or** at least 10. If either of these conditions is true then the statement in the braces will be executed, else it jumps to the next part of the program.

Both logical operators are defined below in the truth table

p	q	p && q		p	q	p || q
True	True	True		True	True	True
True	False	false		True	False	True
False	True	false		False	True	True
false	false	false		false	false	false

Consider the following numerical example:

Expression:	**True or False:**
`(6 <= 6) && (5 < 3)`	`false`
`(6 <= 6) || (5 < 3)`	`true`
`(5 != 6)`	`true`
`(5 < 3) && (6 <= 6) || (5 != 6)`	`true`
`(5 < 3) && ((6 <= 6) || (5 != 6))`	`false`
`!((5 < 3) && ((6 <= 6) || (5 != 6)))`	`true`

The fourth of these expressions is true because the operator "`&&`" has a higher precedence than the operator "`||`".

2.4 The `for` Statement

The “`for`” loop, is an example of a repetition statement. Its syntax is

```
for (initialisation; repetition condition; expression)
{

                Statement1;
                        .                       // statements to be
executed
                        .
                        StatementN;
        }
```

C++ executes such statements as the following steps:

1. executes the `initialisation` statement
2. checks to see if `repetition condition` is true. If not, the loop is ended completely. However if it is true, it executes each of the statements `1` to `N` in turn, and then executes `expression`.
3. Goes back to the beginning of step 2.

Example: Using a `for` loop to calculate $\sum_{n=1}^{100} n^2$

```
        int sum=0;  // very important to initialise with value 0
        for (int n=1; n<=100; n++)
        {
                sum+=n*n;
        }

                cout<<sum<<endl;  //  prints 338350
```

What happens if the `cout` statement is included in the scope of the loop?

The `for` loop is the most commonly used program statement when coding up mathematical problems (and is an essential part of using arrays, which we will discuss later).

2.5 The while Statement

The “`while`” statement, is the third main type of loop supplied by C++. This loop repeats the execution of a statement while its control condition is true. Its syntax is

```
while (condition)
{

                Statement1;
                        .                       // statements to be
executed
                        .
                        StatementN;
        }
```

The dynamics of this structure are: repeatedly evaluate the `condition` and execute the `statement` until the `condition` is false.

Example: Using a `while` statement

Here we use a `while` loop to sum the first 100 integers as in the previous example.

```
int main()
{
        int n=1;        // initialising is done outside the loop
        int sum=0;
        while (n<=100)  // repetition condition
        {
                sum+=n*n;
                ++n;   // increment
        }
        cout<<sum<<endl;  //  prints 338350

        return 0;

}
```

This `while` loop can be written more compactly using a `for` loop as follows:

```
for (int n=1; n<=100; ++n) {
                sum+=n*n; }
```

Exercise: How might we evaluate $\sum_{n=1}^{50}(2n)^2$ or $\sum_{n=1}^{50}(2n-1)$?

2.6 The do … while Statement

A simple variation of the previous loop is the `do … while` loop. This differs from “`for`” and “`while`” loops in that statements inside the braces are always executed once, before the repetition is even checked. These loops are particularly useful, (for example) when ensuring that the program user's keyboard input is of the correct format. Its syntax is of the form

```
do

{

statement(s);

}

while (condition);
```

Example: Euclid's algorithm is used to find the greatest common divisor of two positive integers

```
do{
           while (m<=n)

           n-=m;
           temp=m;
           m=n;
           n=temp;

     }

     while (m>0);
```

2.7 The continue and break Statements

The `for`, `while` and `do … while` loops have one drawback – they evaluate their given condition only at the beginning or end of each iteration. If the body of the loop consists of a block of statements, the loop will execute them all before evaluation of the control condition occurs.

This minor inflexibility can be overcome by use of the `continue` and `break` **statements.**

What output can we expect from the code in the next two examples?

Example: Using the `continue` statement

```
int n=7;
```

```
        for (int k=50; k<=75; k++){
        if (k%n==0) continue;
        cout<<k<<endl;
      }
```

If the "`if`" statement is true then "`continue`" is triggered and goes to the loop condition, else the statement is executed. So it terminates the current iteration without executing the rest of the statements within the loop block. Then it goes back to the control condition to determine whether to terminate the loop itself or continue with the next iteration.

When `k%n==0` then the "`continue`" statement skips over the remaining code in the block and returns to the "`for`" condition. We would then expect the following output (which would actually appear in a column)

```
50,…..,55, _ , 57, …… , 62, _ , 64 , ……, 69, _ , 71, …., 75
```

Example: Using the `break` statement

```
int n=7;
        for (int k=50; k<=75; k++)
        {
        if (k%n==0) break;
        cout<<k<<endl;
      } \\ $$
```

With a "`break`" statement we would have a similar effect (as "`continue`"), but the first time it is encountered, i.e. the statement is true and hence "`break`" triggered, the loop is exited and the program jumps to the next part of the code that follows the loop, i.e. (`$$`). So the output would be

```
50, 52, 53, 54, 55
```

2.8 Nested Loops

A nested loop is one which occurs within the body, or statement part, of another loop. Nested loops are a very powerful construct, particularly in mathematical programming (e.g. defining matrices). The code below shows how nested loops are constructed:

```
#include <iostream>
#include<cstdlib>

using namespace std;

int main()

{
	unsigned int n, j;
	cout << "Enter number: " << endl;
	cin >> n;

	for (j = 2; j <= n / 2; j++)
		if (n%j == 0)
		{
			cout << "It is not a prime as it is divisible by " << j << 
endl;

			exit(0);
		}
	cout << "It is a prime" << endl;


	return 0;

}
```

Library Function `exit()`

In the program above once we have ascertained that an input integer is prime, it is pointless to then move to the next part of the code and check for the converse. By using the library function `exit()`, the program exits immediately, regardless of which stage of the program it is at. The single zero argument is returned to the operating system as a successful termination of the program. Other values can be used, which indicate errors. The `exit()` function belongs to the library `cstdlib`.

2.9 The Conditional Operator ?

A very useful operator, which addresses the common programming situation: A variable is given one value if something is true and another value if it is false. Consider the `if` … `else` statement that gives the variable `minimum` the value of `alpha` or the value of `beta`, depending on which is smaller:

```
if (alpha<beta){
            minimum=alpha;}
      else
            minimum=beta;
```

This can be expressed in the highly convenient and compact form: using the conditional operator:

```
minimum=(alpha<beta) ? alpha : beta;
```

How would you use "`?`" to define a maximum variable (of two real numbers)?

The syntax form of the conditional operator is:

```
Variable=(Test expression)? expression1:expression2 ;
```

Where the term on the right hand side of the equal sign is called the *conditional expression*.

In C++ the operator "`abs`" gives the absolute value of an integer variable x, i.e. $|x|$. How can this be represented by use of "`?`" ?

2.10 Multiple Selection and `Switch` Statements

`if` and `else…if` combinations can become quite confusing when nested too deeply. There is another mechanism for making multiple choices in C++ and that is the `switch` statement. This allows us to make a decision depending on the value of an expression or variable like the `if` statement. However, with the `switch` statement we can check for a number of different values and define different outcomes for each of them. The syntax of the `switch` statement is

```
switch (n)
{
case 1:
        statement 1;
        break;

case 2:
        statement 2;
        break;

.
.
.

case N:
        statement N;
        break;

default;
        statement;
}
```

There are several things to note about such "switch" statements:

- The statements that are executed are exactly those between the first label, which matches the value of selector, and the first "`break`" after this matching label.
- The "`break`" statements are optional, but they help in program efficiency and clarity and should ideally always be used to end each case.
- The selector can have a value of any ordinal type (e.g. "`char`" or "`int`" but not "`float`").

The "`default`" is optional, but is a good safety measure.
We will look at a mathematical example (applied to finance) in the next class when we look at option pricing, but for now we consider a popular case used to show the use of the switch statement.

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

int main(){

       char grade;

       cout<<"Enter exam grade for feedback"<<endl;
       cin>>grade;

       switch (grade) {

       case 'A' : cout << "Excellent - you deserve a cookie!" << endl;
              break;
       case 'B' : cout << "Good" << endl;
              break;
       case 'C' : cout << "OK – not bad" << endl;
              break;
       case 'D' : cout << "Spend less time on social media!" << endl;
              break;
       case 'F' : cout << "Awful! - switch off your mobile in class" << endl;
              break;
       default : cout << "Cannot recognize this grade – please re-enter." << endl;
              break;
       }

       return 0;
}
```

Now consider the code below for the same type of problem, which shows that if more than one case has a similar outcome, then it only needs one explicit definition.

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

int main(){

       int score;
       int test_score;

       cout<<"Please enter your exam score"<<endl;
       cin>>test_score;

       score = test_score/10;

       switch (score)
       {
       case 0:       // for cases 0 – 4 there is only 1 definition
       case 1:
       case 2:
       case 3:
       case 4:     cout << "You have failed on this occasion - you ";
              cout << "must study much harder"<<endl;
              break;
```

```
        case 5:     cout << "You have just scraped through the test."<<endl;
                break;

        case 6:
        case 7:     cout << "You have done quite well."<<endl;
                break;

        case 8:
        case 9:
        case 10:    cout << "Your score is excellent - well done."<<endl;
                break;

        default:    cout << "Incorrect score - must be between ";
                cout << "0 and 100."<<endl;
        }
        return 0;
}
```

2.11 Formatting Real Number Output & the <iomanip> Library

When a program output contains values of type "`float`", "`double`" or "`long double`", we may wish to restrict the precision with which these values are displayed on the screen, or specify whether the value should be displayed in fixed or floating point form. The following example program uses the library identifier "`sqrt`" to refer to the square root function, a standard definition of which is given in the header file `cmath`.

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

int main()
{
        float number;

        cout << "Type in a real number.\n";
        cin >> number;
        cout.setf(ios::fixed);    //       LINE 9
        cout.precision(5);     //   outputs to 5 dp
        cout << "The square root of " << number << " is approximately ";
        cout << sqrt(number) << ".\n";

        return 0;
}
```

This produces the output

```
        Type in a real number.
        200
        The square root of 200.00 is approximately 14.14.
```

whereas replacing line 9 with "`cout.setf(ios::scientific)`" produces the output:

```
Type in a real number.
200
The square root of 2.00e+02 is approximately 1.41e+01.
```

For a natural number *n* the statement `cout.precision(n);` will print real numbers to *n* decimal places.

We can also include tabbing in the output using a statement such as "`cout.width(20)`". This specifies that the next item output will have a width of at least 20 characters (with blank space appropriately added if necessary). This is useful in generating tables.

However the C++ compiler has a default setting for this ***member*** function that makes it right justified. In order to produce output left-justified in a field we need to use some smart input and output manipulation. The functions and operators, which do the manipulation, are to be found in the library file `iomanip` and to do left justification we need to set a flag to a different value (i.e. left) using the `setiosflags` operator:

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

int main()
{
	int number;

	cout << setiosflags ( ios :: left );
	cout.width(20);

	cout << "Number" << "Square Root"<<endl<<endl;

	cout.setf(ios::fixed);
	cout.precision(2);

	for (number = 1 ; number <= 10 ; number = number + 1)
	{
		cout.width(20);
		cout << number << sqrt((double)number) << "\n";
	}
	return 0;
}
```

This program produces the output:

```
Number                  Square Root


1                       1.00
2                       1.41
3                       1.73
4                       2.00
5                       2.24
6                       2.45
7                       2.65
8                       2.83
9                       3.00
10                      3.16
```

The above programs work because "`cout`" is an identifier for an object belonging to the class "`stream`", and "`setf(...)`", "`precision(...)`" and "`width(...)`" are member functions of "`stream`". (Do not worry too much about this for now - you will learn more about objects, classes and member functions later in the object-oriented part of the course.

An important point to note in the output line is the typecasting `((double)number)`. This is required as the `sqrt` function takes a real argument. `number` is an integer so we need to pass this through as a real – try compiling the code without it.

2.12 Outputting Results to Files

As all our work is concerned with mathematical modelling, we will often be representing our results graphically. Displaying these in the DOS environment therefore is of little use.

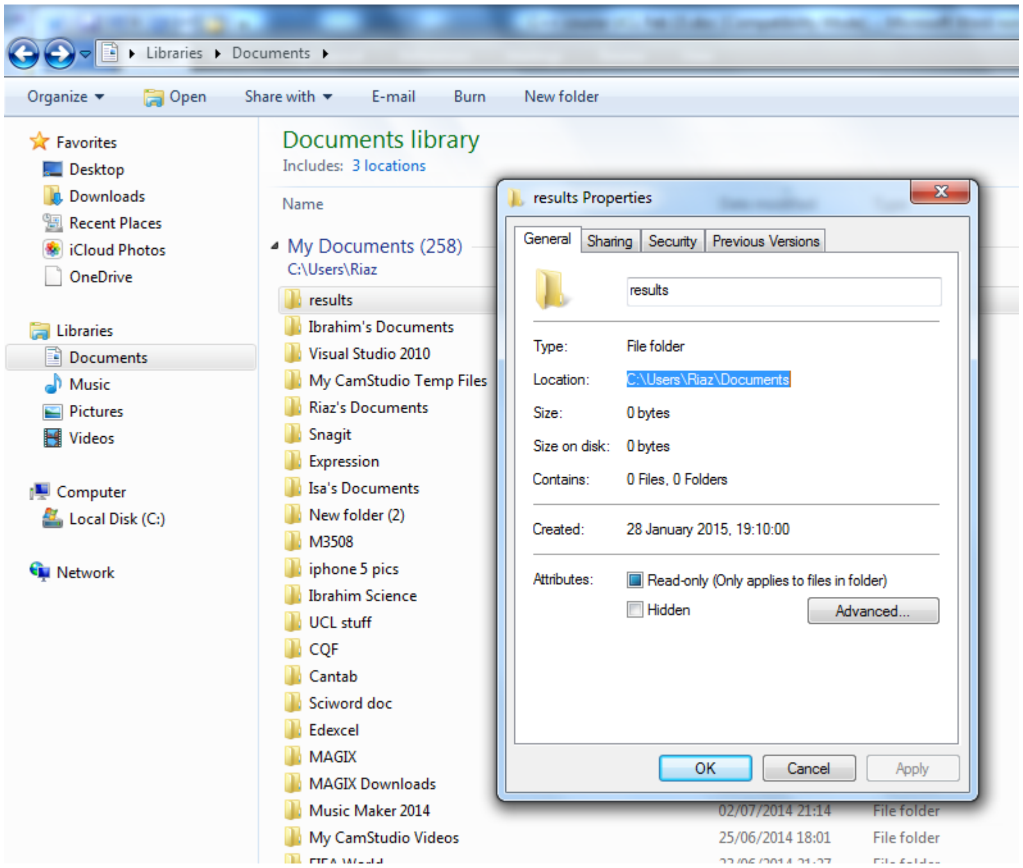
We can output results in any format to excel files, which have the .xls extension. This is of particular advantage due to the graphics capabilities. The following code prints out two columns of data, i.e. x and x^2 to a file called table.xls

```
#include <fstream> // library containing files

using namespace std;

int main()
{

      ofstream out; // create new object (out) belonging to class ofstream
```

```
        out.open("table.xls");  /* out will be used instead of cout to
                                                 output results to an excel file
called table.xls */

        for(short unsigned int t = 0; t <= 10; t++ )
        {
                out << t  << '\t'<< t*t <<endl ;

                // every time out is used, results are stored in table.xls
        }

        out.close(); // file table.xls is now closed

        return 0;

}
```

We can have other file extensions such as .dat and .txt

On the windows platform the documents folder is the default location for all MS Visual Studio projects. So for example if the above code is in a project called ‘Printout’ then table.xls will be located in this particular folder. For any other location the precise pathway has to be stated. Suppose I wanted this location to be a file called ‘results’ in my Documents folder. Right clicking on the results file and looking at the properties gives the Location.



results

Type: File folder

Location: C:\Users\Riaz\Documents

Size: 0 bytes

Size on disk: 0 bytes

Contains: 0 Files, 0 Folders

Created: 28 January 2015, 19:10:00

Attributes: ☑ Read-only (Only applies to files in folder)

☐ Hidden

In this case we wish to printout the table.xls to C:\Users\Riaz\Documents\results\table.xls. In our code we use a double back slash so in the ealier program we have the line

```
out.open("C:\\Users\\Riaz\\Documents\\results\\table.xls");
```

2.13 Reading from a file

In 2.11 we have made use of writing data to a file in formats such as text, excel and dat. Reading in data from files is of equal importance. The definitions of file manipulation classes (which we will define shortly) are in the library called `fstream`, which has to be included as in the case of `iostream, cmath`, etc.

Naturally before using a file, we must open it. The two basic modes are *read* and *write*, meaning that we can open a file in order to read data from it or open it to write data to it.

If we open a file for reading, then clearly it must exist. However, if we open a file for writing then it may or may not exist.

Consider the following (complete program), which reads in a column of integers from a file called test.txt

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

int main ()

{
       int k;

       fstream input;

       input.open("C:\\Users\\Riaz\\Documents\\Visual Studio 
2010\\Projects\\First\\First\\test.txt", ios::in);

       while (! input.eof() )

       {
              input>> k;
              cout<< k <<endl;
       }
       input.close();

       return 0;
}
```

What have we done here? We have created an object of the class `fstream` called `input`, open a file `test.txt` and read in 10 integers from it. The value is then displayed on the screen using `cout`. Note that the way we read from the file is similar to reading in from the keyboard when using `cin`.

As we are reading in, we require a mechanism for detecting that we have reached the end of the file. The member function `eof` will return true if we have read all of the data from the file; else if there is more data it will return false. It uses a `while` loop to read the data and stops only when the end of file is reached.

If you are using an older version of C++, then you need only declare `fstream.h` as `iostream.h` is a subset of `fstream.h`

So in summing up, we open a file using the member function `open` giving the file name as the first parameter and the special term `ios::out` as the second to specify write mode. We open a file for reading by declaring an object of the class `fstream`, and then open it using the member function `open` again, giving the filename as the first parameter but this time with the special term `ios::in` as the second parameter that specifies the mode. We can then use the file object names to read and write data in the usual way by `cin` and `cout`.

We end by looking at appending data to a file. This is done in the same way as writing to a file, except that the mode parameter used is `ios::app`. The code below shows how this can be done, by appending the integers 13 and 14 to the existing file `test.txt`

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

int main ()

{

       ofstream output;

       output.open("test.txt", ios::app);

       output<<13<<endl;
       output<<14<<endl;

       output.close();

       return 0;

}
```

2.14 Review Exercises

1. Explain the various parts of a **for** statement.

```
for (initialisation; repetition condition; expression)
```

2. What is output after executing the following segment of code?

```
int x = 3;
int y = 6;
if(x < 0 && y == 2)
       x = 40;
cout << x << " " << y << endl;
```

3. What is output after executing the following segment of code?

```
int x = 20;
int y = 1;
if(x < 0 || x > y && y != 9 )
{
       --x;
       --y;
}
cout << x << " " << y << endl;
```

4. Rewrite the following **for** loop as a **while** loop:

```
for (int i=10; i<20; ++i){
       cout<<i*i<<endl;}
```

2.15 Exercises

- Write programs to check the following formulae by inputting *n* (your choice) and then computing and comparing both sides of the equation

  a) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ b) $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ c) $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$

  Rewrite the code to sum odds and evens for each of the above.

- Use the conditional operator to translate the following code into a single assignment:

```
if (x>=0) sqrtx=sqrt(x);
            else sqrtx=sqrt(-x);
```

- The factorial function is defined by $n! = n(n-1)(n-2).....3.2.1$ - write a program that inputs an integer value of n and outputs *n*!.

- Rewrite the following `while` loop as a `for` loop:

```
int i=4;
while (i<20)
{
      cout<<1.0/i<<endl;
      ++i;
}
```

  Rewrite the following `for` loop as a `while` loop:

```
for (int j=22; j>8; j--)
      sum+=log(j*j);
```

- Consider the function $f(x) = x^2$ over the closed interval $1 \le x \le 2$. Discretise the function over this range in steps of 0.1, i.e.

x	$f(x)$
1	1
1.1	1.21
1.2	1.44
….	….

- Experiment with different formats of output using the `iomanip` library, by changing precision and look at the effects of varying widths in tables.

3. Functions

3.0 The Need for Sub-programs

A natural way to solve large problems is to break them down into a series of sub-problems, which can be solved independently and then combined to arrive at a complete solution.

In programming, this methodology reflects itself in the use of *sub-programs*, and in C++ all sub-programs are called *functions*. A function is executed by being called from within another function.

Recall that every C++ program must include a function named `main()`. This main body of the program is where most program executions start.

We have already been using sub-programs. For example, in the program, which solved a quadratic equation, we used the “in-built” function `sqrt` provided by C++.

The function "`sqrt(...)`" is defined in a sub-program accessed via the library file `cmath` (old header style `math.h`).

Recall: The sub-program takes "`b*b-4*a*c`", uses a particular algorithm to compute its square root, and then returns the computed value back to the program. In using this, our concern is not what the algorithm is as long as it gives the correct result.

In this lecture we will discuss how the programmer can define his/her own functions. Initially, we will put these functions in the same file as "`main`". Later we will see how to place different functions in different files.

3.1 Function Declarations and Definitions

Like a variable, a function must (obviously) be declared before it is called.

A function declaration has three components: its return type, its name and its parameter list. Good programming also encourages the use of comments to briefly explain the role of the function.

Example: Consider a function, which takes three real numbers and returns the average. A typical function declaration called a prototype would be

```
float Average(float x, float y, float z);
```

What does each part refer to?

`float`	the return type, i.e. a real of type float will be returned upon completion
`Average`	name of function
`(float x, float y, float z)`	parameter/argument list together with types of parameter

A function prototype is a statement therefore it must end with a semi-colon.

So the declaration tells the C++ compiler the name, return type and parameters of the function. The function tells the compiler how the function works – this is where the bulk of code goes.

The function prototype does not need to contain actual names of the parameters, just the type. So the declaration `float Average(float, float, float);` is also perfectly legal. However adding parameter names makes your function prototype clearer and is an example of good programming practice.

If a function does not return a value, its return value will be type `void`. So as an example if we wish to write a function which simply prints out a message to the screen then its declaration might be

`void print_message()`. We could even write the "hello world" program from the first lecture as

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

void main()

{cout<<"hello world "<<endl;}
```

3.2 User-defined Functions

Here is a trivial example of a program which includes a user defined function, called "`Average(...)`". The program computes the average of three real numbers:

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

float Average(float, float, float);      //function declaration

/* MAIN PROGRAM: */
int main()
{
       float x, y, z;

       cout << "Enter numbers: ";              /* <--- line 11 */
       cin >> x >> y >> z;
       cout << endl;


       cout << "The average of " << x <<", "<< y << " & ";

       cout << z <<" is " << Average(x, y, z)<<endl;
       return 0;
}

/* END OF MAIN PROGRAM */


/* FUNCTION TO CALCULATE AVERAGE: */

float Average(float x, float y, float z)    /* start of function definition */
{
       float aver;

       aver = (x+y+z)/3;

       return aver;
}                                  /* end of function definition */
/* END OF FUNCTION */
```

Although this program is not written in the most succinct form possible, it serves to illustrate a number of features concerning functions:

- The structure of a function definition is like the structure of "`main()`", with its own list of variable declarations and program statements.
- A function can have a list of zero or more parameters inside its brackets, each of which has a separate type.

- A function has to be declared in a function declaration at the top of the program, just after any global constant declarations, and before it can be called by "`main()`" or in other function definitions.

- Function declarations are rather like declarations - they specify which type the function will return.

A function may have more than one "return" statement (two at most), in which case the function definition will end execution as soon as the first "return" is reached. For example:

```
        double absolute_value(double number)
{
        if (number >= 0)
               return number;
        else
               return 0 - number; // can have brackets
}
```

3.3 CDF Function

In module 2 we wrote a program `CDF.cpp` to calculate the Cumulative Distribution Function $N(x)$. We now write this as a function, which is naturally the most efficient way.

```
#include<iostream>
#include<cmath>

using namespace std;

const double pi=4.0*atan(1.0); //define constant pi=3.142 (global)

double CDF(double); // function declaration – takes 1 real i.e. RV X

int main()
{
       double X=0;  // X is the random variable


       cout<< "Enter the value of the random variable X" <<endl;
       /* upper limit of integral*/

       cin>>X;

       cout<< CDF(X) <<endl; //output value of N(x) by calling the function
       system("pause");
       return 0;

}

/* Now define the function */

double CDF(double X)
{
       const double  a1=0.319381530, a2=-0.356563782, a3=1.781477937,
              a4=-1.821255978, a5=1.330274429;

       double x=0, k=0; //always good to initialise variables

       double N, CDF, n;

       x=fabs(X);

       k=1/(1+0.2316419*x);

       n=(1/sqrt(2*pi))*exp(-0.5*x*x);


       N=1-n*(a1*k+a2*k*k+a3*pow(k,3)+a4*pow(k,4)+a5*pow(k,5));


       if (X>=0)

              CDF=N;

       else

              CDF=1-N;

       return CDF;
}
```

We can now start doing more interesting problems.

3.4 Option Pricing Formula

The Black-Scholes equation for the price of an option $\boldsymbol{V(S,t)}$ in the absence of dividends is

$$\frac{\partial V}{\partial t}+\frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2}+rS\frac{\partial V}{\partial S}-rV=0.$$

that is solved together with two boundary conditions and one final condition (called the Payoff function). If $\boldsymbol{V(S,t)}$ is a European call option $\boldsymbol{C(S,t)}$ then the conditions are

1. $\boldsymbol{C(S,t)=0}$ when $\boldsymbol{S=0}$
2. $C(S,t)\to S$ as $S\to\infty$
3. $C(S,T)=\max(S-E,0)$

The solution gives a pricing formula for Call Option $C(S,t)$ with

$$C(S,t)=SN(d_1)-E\exp(-r(T-t))N(d_2)$$

where

$$d_1=\frac{\log(S/E)+\left(r+\frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}}\qquad d_2=\frac{\log(S/E)+\left(r-\frac{1}{2}\sigma^2\right)(T-t)}{\sigma\sqrt{T-t}}$$

$$N(x)=\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x}\exp\left(-\frac{1}{2}\phi^2\right)d\phi$$

And $d_2=d_1-\sigma\sqrt{T-t}$

Consider the following information

$$S=100,\ (T-t)=1.0;\ r=5\%,\ \sigma=20\%;\ E=100$$

The following code placed in the main body of CDF.cpp calculates d_1 and d_2 for the given variables and parameters, which are then passed through the CDF function as arguments. $N(d_1)$ and $N(d_2)$ are then subsequently used to calculate the price of a call option.

```
      double Option, d1, d2;

      double S=100, E=100, r=0.05, vol=0.2, tau=1.0; //tau=time to expiry
      d1=(log(S/E)+(r+0.5*vol*vol)*tau)/(vol*sqrt(tau));
      d2=d1-vol*sqrt(tau);

    Option=S*CDF(d1)-E*exp(-r*tau)*CDF(d2);
```

For this choice of parameters the theoretical Black-Scholes option price is 10.45

The price of a put option $P(S,t)$ for the same input values is 5.57 and is obtained from

$$P(S,t) = E\exp(-r(T-t))N(-d_2) - SN(-d_1).$$

However the Call and Put prices are related via the ***Put-Call Parity***

$$C - P = S - E\exp(-r(T-t))$$

So if we have the price of one option the other can be calculated in a most trivial manner, i.e.

$$P(S,t) = C(S,t) - S + E\exp(-r(T-t))$$

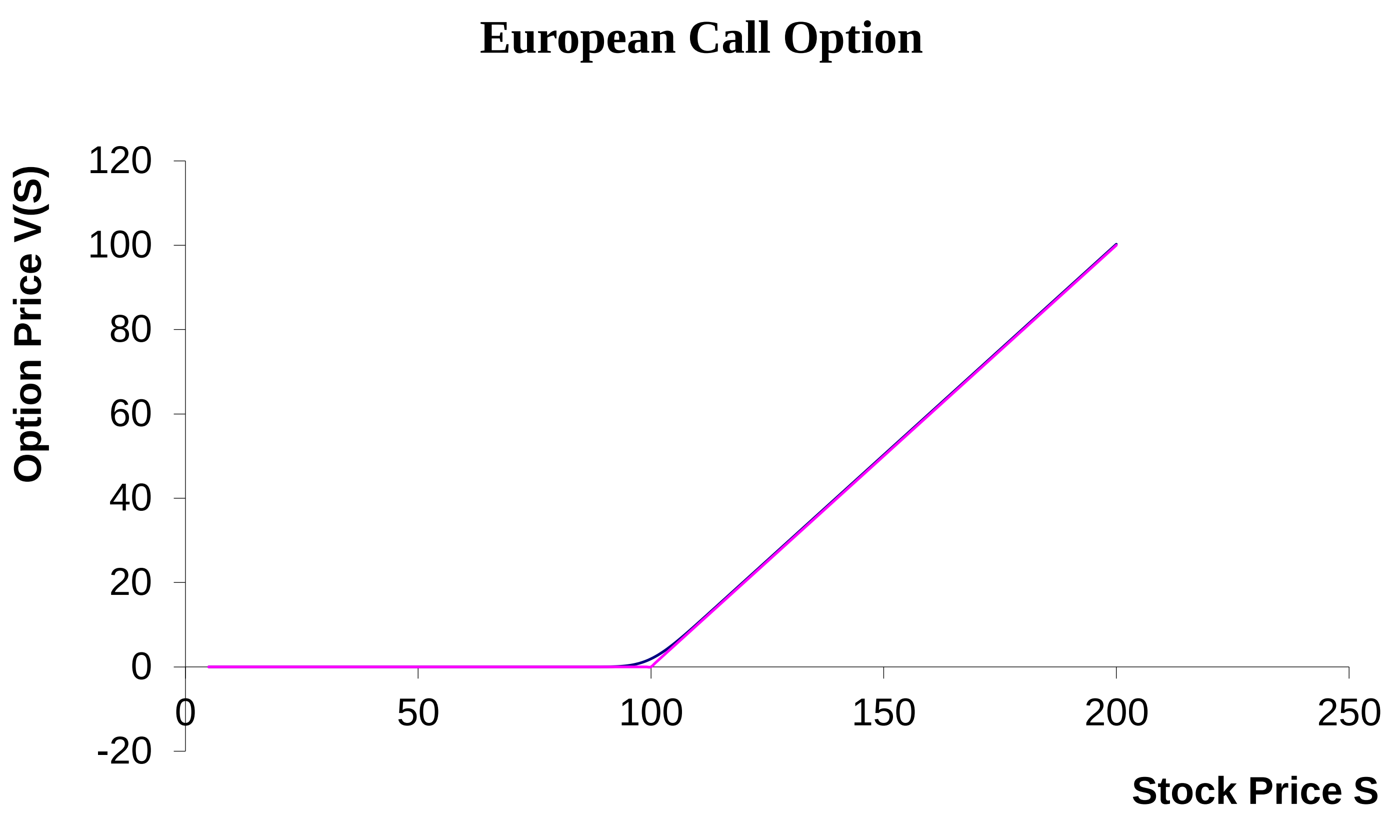
Exercise

The price of a put option $P(S,t)$ for the same input values as taken for the call option evaluation is 5.57. Using both the formula for the price of a put as well as the put-call parity, use CDF.cpp to verify this.

By using a for loop in asset price or time we can look at the option price by varying either (or both) variables. The following main body

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

using namespace std;


const double pi=4.0*atan(1.0); //define constant pi=3.142

double CDF(double);

int main()
{

      double Call_Option, d1, d2;

      double S=100, E=100, r=0.05, vol=0.2, tau=0.05;


      ofstream out; // create object “out” to printout to excel file

      out.open("BSE.xls"); //out.open(“BSE.xls”); - 2 parameters

      for (S=0; S<=200; S+=5){

      d1=(log(S/E)+(r+0.5*vol*vol)*tau)/(vol*sqrt(tau));
      d2=d1-vol*sqrt(tau);

      Call_Option=S*CDF(d1)-E*exp(-r*tau)*CDF(d2);

      out<<S<<'\t'<<Call_Option<<endl;
```

```
        }
        out.close();
        return 0;
```



```
}
```

together with the CDF function, gives the following results in an excel file:
The option value shortly before expiry is plotted together with the price at expiry, which is the payoff function.

<u>Exercise</u>: Repeat this for different times to expiry $(T-t)$.

3.5 Value and Reference Parameters

The parameters in the functions above are all ***value parameters***. When the function is called within the main program, it is passed the values currently contained in certain variables. For example, "`Average(...)`" is passed the current values of the variables "`x, y, z`". The function "`Average(...)`" then stores these values in its own private variables, and uses its own private copies in its subsequent computation.

<u>IMPORTANT</u>: Functions which use Value Parameters are Safe

The idea of value parameters makes the use of functions "safe", and leads to good programming style. It helps guarantee that a function will not have hidden *side effects*. Here is a simple example to show why this is important. Suppose we want a program that produces the following dialogue:

```
Enter a positive integer:
4
The factorial of 4 is 24, and the square root of 4 is 2.
```

It would make sense to use the predefined function "`sqrt(...)`" in our program, and write another function "`factorial(...)`" to compute the factorial n! of any given positive integer n. Here is the complete program:

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

int factorial(int number); // Function declaration

/* MAIN PROGRAM: */
int main()
{
      int whole_number;

      cout << "Enter a positive integer:"<<endl;
      cin >> whole_number;
      cout << "The factorial of " << whole_number << " is ";
      cout << factorial(whole_number); // Function call

      cout << ", and the square root of " << whole_number << " is
";
      cout << sqrt(whole_number) << ".\n";

      return 0;
}
/* END OF MAIN PROGRAM */

/* FUNCTION TO CALCULATE FACTORIAL: */
int factorial(int number)
{
      int product = 1;   //base case

      for ( ; number > 0 ; number--)
            product *= number;

      return product;
}
  /* END OF FUNCTION */
```

By the use of a value parameter, we have avoided the (correct but unwanted) output

```
        Enter a positive integer:
        4
        The factorial of 4 is 24, and the square root of 0 is 0.
```

which would have resulted if the function "`factorial(...)`" had permanently changed the value of the variable "`whole_number`".

Reference Parameters

Under some circumstances, it is legitimate to require a function to modify the value of an actual parameter that it is passed. We can achieve this as follows using reference parameters, whose types are post-fixed with an "&":

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

        int area(int length, int width);

        void get_dimensions(int& length, int& width);

        /* MAIN PROGRAM: */
        int main()
        {
                int this_length, this_width;

                get_dimensions(this_length, this_width);

                cout << "The area of a " << this_length << "x" <<
this_width;
                cout << " rectangle is " << area(this_length, this_width);

                return 0;
        }
        /* END OF MAIN PROGRAM */

        /* FUNCTION TO INPUT RECTANGLE DIMENSIONS: */
        void get_dimensions(int& length, int& width)
        {
                cout << "Enter the length: ";
                cin >> length;
                cout << "Enter the width: ";
                cin >> width;
                cout << "\n";
        }
        /* END OF FUNCTION */

        /* FUNCTION TO CALCULATE AREA: */
        int area(int length, int width)
        {
                return length * width;
        }
        /* END OF FUNCTION */
```

Notice that, although the function "`get_dimensions`" permanently alters the values of the parameters "`this_length`" and "`this_width`", it does not return any other value (i.e. is not a "function" in the mathematical sense). This is signified in both the function declaration and the function heading by the reserved word "`void`".

There are another two ways that an argument can be passed to a parameter in a function call:

1. by `constant` value

   Passing an argument this way is the same as passing it by value except that the `const` keyword makes the parameter a constant, prohibiting the function from changing its value. This type of restriction that good programmers often force upon themselves to prevent coding errors. To avoid the remotest risk of a function changing its parameter value, make it a constant.

2. by `constant` reference

   Passing an argument this way is the same as passing it by value except that the `const` keyword makes the parameter a constant, prohibiting the function from changing its value. This type of restriction that good programmers often force upon themselves to prevent coding errors. To avoid the remotest risk of a function changing its parameter value, make it a constant.

Example: Consider the factorial program. What would happen if we did actually pass by reference? That is the function declaration `int factorial(int number);`

3.6 Polymorphism and Overloading

C++ allows *polymorphism*, i.e. it allows more than one function to have the same name, provided all functions are either distinguishable by the typing or the number of their parameters. Using a function name more than once is sometimes referred to as ***overloading*** the function name. Here is an example:

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


      float average(int first_number, int second_number, int
      third_number);

      float average(int first_number, int second_number);



      /* MAIN PROGRAM: */
      int main()
      {
            int number_A = 5, number_B = 3, number_C = 11;

            cout << "The average of " << number_A << " and ";
            cout << number_B << " is ";
            cout << average(number_A, number_B) << "."<<endl;

            cout << "The average of " << number_A << ", ";
            cout << number_B << " and " << number_C << " is ";
            cout << average(number_A, number_B, number_C) << "."<<endl;

            return 0;
      }
      /* END OF MAIN PROGRAM */

      /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 3 INTEGERS: */
      float average(int first_number, int second_number, int
third_number)
      {
            return ((first_number + second_number + third_number) / 3.0);
      }
      /* END OF FUNCTION */

      /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 2 INTEGERS: */
      float average(int first_number, int second_number)
      {
            return ((first_number + second_number) / 2.0);
      }
      /* END OF FUNCTION */
```

This program produces the output:

```
        The average of 5 and 3 is 4.

        The average of 5, 3 and 11 is 6.33333.
```

3.7 Procedural Abstraction and Good Programming Style

One of the main purposes of using functions is to aid in the *top down* design of programs. During the design stage, as a problem is subdivided into tasks (and then into sub-tasks, sub-sub-tasks, etc.), the problem solver (programmer) should have to consider only what a function is to do and not be concerned about the details of the function.

The function name and comments at the beginning of the function should be sufficient to inform the user as to what the function does. (Indeed, during the early stages of program development, experienced programmers often use simple "dummy" functions or *stubs*, which simply return an arbitrary value of the correct type, to test out the control flow of the main or higher level program component.)

Developing functions in this manner is referred to as *functional* or *procedural abstraction*. This process is aided by the use of value parameters and local variables declared within the body of a function.

Functions written in this manner can be regarded as "black boxes". As users of the function, we neither know nor care why they work.

3.8 Splitting Programs into Different Files

As we have seen, C++ makes heavy use of predefined standard libraries of functions, such as "`sqrt(...)`". In fact, the C++ code for "`sqrt(...)`", as for most functions, is typically split into two files:

- The *header file* "`cmath`" contains the function declarations for "`sqrt(...)`" (and for many other mathematical functions). This is why in the example programs which call "`sqrt(...)`" we are able to write "`#include<cmath>`", instead of having to declare the function explicitly.

- The *implementation file* "`math.cpp`" contains the actual function definitions for "`sqrt(...)`" and other mathematical functions. (In practice, many C++ systems have one or a few big file(s) containing all the standard function definitions, perhaps called "`ANSI.cpp`" or similar.)

It is easy to extend this library structure to include files for user-defined functions, such as "`factorial(...)`" and "`average(...)`". As an example, consider the earlier program, but split into a main program file, a header file for the two average functions, and a corresponding implementation file.

The code in the main program file is as follows:

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

        int main()
        {
                int number_A = 5, number_B = 3, number_C = 10;

                cout << "The average of " << number_A << " and ";
                cout << number_B << " is ";
                cout << average(number_A, number_B) << ".\n\n";

                cout << "The average of " << number_A << ", ";
                cout << number_B << " and " << number_C << " is ";
                cout << average(number_A, number_B, number_C) << ".\n";

                return 0;
        }
```

Notice that whereas "`include`" statements for standard libraries such as "`iostream`" delimit the file name with angle ("<>") brackets, the usual convention is to delimit user-defined library file names with double quotation marks - hence the line " `#include"averages.h"` " in the listing above.

The code in the header file "`averages.h`" is listed below. Notice the use of the file identifier "`AVERAGES_H`", and the reserved words "`ifndef`" ("if not defined"), "`define`", and "`endif`". "`AVERAGES_H`" is a (global) symbolic name for the file. The first two lines and last line of code ensure that the compiler (in fact, the *preprocessor*) only works through the code in between once, even if the line "`#include"averages.h"`" is included in more than one other file.

Constant and type definitions are also often included in header files.

```
        #ifndef AVERAGES_H
        #define AVERAGES_H

        /* (constant and type definitions could go here) */

        /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 3 INTEGERS: */
        int average(int first_number, int second_number, int
third_number);

        /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 2 INTEGERS: */
        int average(int first_number, int second_number);

        #endif
```

Finally, the code in the implementation file "averages.cpp" is as follows:

```
        /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 3 INTEGERS: */
        int average(int first_number, int second_number, int third_number)
        {
                return ((first_number + second_number + third_number) /
3);
        }
        /* END OF FUNCTION */

        /* FUNCTION TO COMPUTE INTEGER AVERAGE OF 2 INTEGERS: */
        int average(int first_number, int second_number)
        {
                return ((first_number + second_number) / 2);
        }
        /* END OF FUNCTION */
```

Note the modularity of this approach. We could change the details of the code in "averages.cpp" without making any changes to the code in "averages.h" or in the main program file.

Example: Consider decomposing the program CDF.cpp into source and header files.

1. main.cpp   Main Body of program

```
#include<iostream>
#include<cmath>
#include<fstream>
#include "CDF.h"

using namespace std;


int main()
{

      double Call_Option, d1, d2;

      double S=100, E=100, r=0.05, vol=0.2, tau=1.0;


      ofstream out;

      out.open("BSE.xls");

      for (S=0; S<=200; S+=5){

      d1=(log(S/E)+(r+0.5*vol*vol)*tau)/(vol*sqrt(tau));
      d2=d1-vol*sqrt(tau);

      Call_Option=S*CDF(d1)-E*exp(-r*tau)*CDF(d2);

      out<<S<<'\t'<<Call_Option<<endl;
      }

      out.close();

      return 0;

}
```

2. CDF.h Contains Function prototype(s) and global declarations

```
#ifndef CDF_H
#define CDF_H

#include<cmath>  // needed for the function atan

const double pi=4.0*atan(1.0); //define constant pi=3.142

double CDF(double); // function prototype


#endif
```

3. CDF.cpp Function definition – all mechanics of procedure/subroutine go here

```
#include"CDF.h" //need this for deifinition of pi


double CDF(double X)
{
      const double  a1=0.319381530, a2=-0.356563782, a3=1.781477937,
                           a4=-1.821255978, a5=1.330274429;

      double x=0, k=0;
      double N, CDF, n;

      x=fabs(X); /* cmath is included in the header file, so do not need
                              to redefine here */

      k=1/(1+0.2316419*x);

      n=(1/sqrt(2*pi))*exp(-0.5*x*x);


N=1-n*(a1*k+a2*k*k+a3*pow(k,3)+a4*pow(k,4)+a5*pow(k,5));


      if (X>=0)

      CDF=N;

      else

            CDF=1-N;

      return CDF;
}
```

3.9 Monte Carlo Code – Box Muller Transformation

The function prototype and definition for producing normal deviates are in turn

```
double normal(double, double);


double normal(double std, double mean)
{


        static int iset = 0;
        static double gset;
        double fac, r,v1, v2;

            // create two normally-distributed numbers
        if (iset == 0)
        {
                r = 0;
                do
                      {     //compute two possibles
                            v1 = 2.0 *  rand()/RAND_MAX - 1.0;
                    v2 = 2.0 *  rand()/RAND_MAX - 1.0;
                            // they define radius
                    r = v1*v1 + v2*v2;
                }
                while (r >= 1.0 || r==0.0);
                  // in unit circle? if not try again

                fac = sqrt((-2*log(r))/r); // Box-Muller transform
                gset = (v1 * fac) ;
                iset = 1;  // save one
                v2 =v2*fac*std+mean; // scale and return one
                return v2;
        }
        else
        {
                iset = 0;
                return (gset*std)+mean;
                  //scale and return the saved one
        }
}
```

Each time the `rand()` function is called it generates the same values from a fixed "seed" . The role of `srand()` is to change the seed every time the program is excuted. Since it uses the time as the seed value the `rand()` function becomes even more random.

We note use of a new type, i.e. `static`. When used in a function it means that the static value is still stored and kept in memory even after the function is ended.

To simulate the random walk the following code will produce an asset price realisation.

The main body of the program to perform the Monte Carlo simulations is

```
#include<iostream>
#include<cmath>
#include<ctime>
#include<fstream>


using namespace std;

double normal(double, double ); // function prototype


int main()
{
      srand((unsigned)time(NULL));
      ofstream print;

      print.open("results.xls");
      long N=1000;


      double asset=100, IR=0.05, vol=0.2; // variables & parameters

      double dt=1.0/N;  // step-size for time

      print<<0<<'\t'<<asset<<endl;

      for (unsigned short int i=1; i<=N; i++){
            double time=i*dt;
            double dX=normal(1.0, 0.0)*sqrt(dt);
            double dS=asset*(IR*dt+vol*dX);
             asset+=dS;

             print<<time<<'\t'<<asset<<endl;

      }

      print.close();

      return 0;
}
```

3.10 A First Order Differential Equation

Consider the problem

$$\frac{dy}{dx} = f(x, y); \quad y(x_0) = y_0$$

which is a first order ordinary differential equation (ODE) together with an initial condition that at $x = x_0$, $y = y_0$. This is called an *initial value problem* (IVP). The form of $f(x, y)$ can be of the separable type, or one that makes the ODE linear, Bernouilli or exact – types that will be familiar. Very often, when modelling complex situations the ODE will not have an exact solution.

Consider the IVP given above, to be solved over the interval $a \le x \le b$ and initial/starting condition $y(a) = \alpha$.

We begin by discretising the independent variable x. If we choose to have N steps in our computation with equal step lengths or **step size** of $h = \frac{b-a}{N}$ then

$$x_i = a + ih \quad 0 \le i \le N .$$

The approximation for y_i at each x_i will be denoted w_i , i.e. $w_i = y(x_i)$. A popular and robust numerical scheme for solving is the fourth order **Runge-Kutta** method, which in difference scheme is given by:

$$\begin{aligned}
w_0 &= \alpha \\
k_1 &= hf(x_i, w_i) \\
k_2 &= hf\left(x_i + \frac{h}{2}, w_i + \frac{1}{2}k_1 \right) \\
k_3 &= hf\left(x_i + \frac{h}{2}, w_i + \frac{1}{2}k_2 \right) \\
k_4 &= hf(x_{i+1}, w_{i+1} + k_3)
\end{aligned}$$

$$w_{i+1} = w_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

for $i = 0, 1, ..., N-1$.

The associated local error is of fourth order, i.e. $O(h^4)$. The Runge-Kutta method is an example of a direct method as opposed to an iterative scheme. Naturally the finer the grid in x, the better the sequence of solutions obtained for y.

Here we will output the results into a file, from inside a function (rather than the main body). There are two ways to achieve this.

1. We can create the printing object (say `out`) of type `ofstream` in the main body and pass as a function parameter.
2. Simply create the object within the function where we wish to output results.

In the following code we illustrate method 1. As an **exercise** try the second suggestion given.

main.cpp

```
#include <iostream>
#include <cmath>
#include <fstream>
#include "runge.h"
using namespace std;
```

```
int main()
{

    ofstream out;
      out.open("res.xls");

      double a, b, no_steps, IC;


      double h=read(a,b,IC,no_steps);

      runge(out, a, IC, no_steps, h);


      cout<<"step size "<<h<<endl;
      cout<<" range of x: " <<a<<"<= x >=" << b <<endl;



      out.close();


return 0;
}
```

runge.h

```
#ifndef RUNGE_H
#define RUNGE_H
#include <iostream>

double read(double& ,double& , double& , double&);
double runge(std::ofstream&, double ,double , double , double);
double f(double,double);

#endif
```

runge.cpp

```
#include <iostream>
#include <cmath>
#include <fstream>
#include "runge.h"
using namespace std;


double read(double& a,double& b,double& IC, double& no_steps)

      {


      cout<<"Enter the range of integration\n";
      cout<<"a = " ; cin>>a;
      cout<<"b = " ; cin>>b;
      cout << "Enter initial condition y(a)"<<endl;
      cin >> IC;

      cout<<"How many steps"<<endl;
      cin>> no_steps;

      double h=(b-a)/no_steps;

            return h;

      }




double runge(ofstream& out, double a,double IC, double no_steps, double
h)
{
      double K1, K2, K3, K4;

      double x=a;
    double w=IC;

      out.precision(12);

      out<<a<<'\t'<<w<<endl;
      for (int i=1; i<=no_steps; i++)
      {

            K1= h*f(x,w);
            K2= h*f(x+0.5*h,w+0.5*K1);
            K3= h*f(x+0.5*h,w+0.5*K2);
            K4= h*f(x+h,w+K3);

            w=w+(K1+2*(K2+K3)+K4)/6.0;

            x=a+i*h;

            out<<x<<'\t'<<w<<endl;  /* results output to excel file*/
            cout<<w<<endl;


      }

      return 0 ;

}

/* The function f defines the right hand side of the diff eq. - change
this
to look at other equations - provided you know the initial condition*/



double f(double x, double y)

{
    double function= (2/x)*y+(x*x)*exp(x); // 1<=x<=2  IC y(1)=0  h=0.1

      return function;

}
```

3.11 Numerical Integration Code

The accompanying set of notes on various numerical methods outlines two integration techniques. In this section we present the code for performing this by writing two functions `Simpson` and `Trapezoidal`. The header file `integ.h` contains function prototypes for both the above, together with the definition for the integrand. Header file `data.h` has the function declarations for reading in the data and printing out a message to the user confirming their input.

1. `integ.h`

```
#ifndef INTEG_H
#define INTEG_H


double f(double x);
double Simpson(double,double, double, double);
double Trapezoidal(double,double, double, double);


#endif
```

2. `data.h`

```
#ifndef DATA_H
#define DATA_H


const double pi=4*atan(1.0);
double read(double&,double&, double&, double&);
void confirm(double&,double&, double&);



#endif
```

3. `main.cpp`

```
/* A numerical integration program which uses both
Simpson and Trapezoidal rules to evaluate definite integrals
```

```
The integrand is evaluated separately in a function called "f".
So to change the function you wish to integrate, simply involves
editing one line.*/

/* a and b are lower and upper limits of integration, in turn
   no_steps = no. of steps we discretize the range of integration into
   h = (b-a)/no_steps i.e. step length  */


#include <iostream>
#include <cmath>
#include "data.h"
#include "integ.h"


using namespace std;

int main()
{

      double a, b, no_steps, h;


      read(a,b,no_steps,h);

      confirm(a,b,no_steps);

      cout<< "The integral using Simpson's rule is "<<
Simpson(a,b,no_steps,h)<<endl;
      cout<<endl;
      cout<< "The integral using the Trapezoidal rule is "
<<Trapezoidal(a,b,no_steps,h)<<endl;


return 0;

}
```

4. integ.cpp

```
// function definition for integ.h

#include <iostream>
#include <cmath>
#include "data.h"
#include "integ.h"

using namespace std;


double f(double x) // integrand definition

{ /* 3 different functions to try - remove the comment command
      when using the function of interest */
```

```
	//double function=x*x; // this is the only line to change.

  // double function= 1.0/(sqrt(2*pi))*exp(-x*x/2.0);

	double function= sin(2*x)*(1.0/(1+x*x))*exp(-x*x/2.0);

	return function;

}



double Simpson(double a, double b, double no_steps, double step_size)
{

	double sum_even=0.0;

	double sum_odd=0.0;


	for (int i=1; i<=no_steps-1; i++)
	{
		double x=a+i*step_size;

		if (i%2==0){

				sum_even+=f(x);

		}

		else
		{
			sum_odd+= f(x);



		}


	}

	double inner_pts=4*(sum_odd)+2*(sum_even);


	double f_x=step_size*(f(a)+inner_pts+f(b))/3.0;


	return f_x;
}
double Trapezoidal (double a, double b, double no_steps, double
step_size)
{



	double sum=0.0;
```

```
      for (int i=1; i<=no_steps-1; i++)
      {
            double x=a+i*step_size;


                  sum+=f(x);



      }

      double inner_pts=2*sum;



      double f_x=0.5*step_size*(f(a)+inner_pts+f(b));


      return f_x;
}
```

5. data.cpp

```
#include <iostream>
#include<cmath>
#include "data.h"
#include "integ.h"

using namespace std;



      double read(double& a,double& b,double& no_steps, double& h )
      {

      cout<<"Enter the range of integration\n";
      cout<<"a = " ; cin>>a;
      cout<<"b = " ; cin>>b;

      cout<<"How many steps"<<endl;
      cin>> no_steps;

       h=(b-a)/double(no_steps);

      return h;

      }


      void confirm(double& a,double& b,double& no_steps)

{
cout << "f(x) will be integrated over the range : ["
```

```
                <<a<<" , "<< b<<"], using "<<
                 no_steps<< " steps."<<endl;

}
```

4. Arrays and Strings

4.0 The Basic Idea and Notation

Think of a vector or matrix in maths and this helps explain the idea of an **array**. The use of arrays permits us to set aside a group of memory locations (i.e. a group of variables) that we can then manipulate as a single entity, but that at the same time gives us direct access to any individual component.

Arrays are simple examples of structured data types - they are basically lists of variables all of the same data type ("`int`", "`float`", etc) placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

Declaring an array:

As with variables, an array has to be declared before it can be used. The general syntax for an array declaration is:

```
        <component type>  <name>[<integer value>];
```

For example, consider the array declaration:

```
        int column[6];
```

or better,

```
        const int size = 6;

        int column [size];
```

In each case, we end up with 6 variables of type "`int`" with identifiers

```
        column [0]   column [1]   column [2]   column [3]   column [4]
        column [5]
```

The size of an array must also be specified when it is declared so as to set aside the required amont of memory; it cannot be entered at run time, but must be known at compile time. Each of these is referred to as an *element* or *component* of the array. Note the use of square brackets here. The numbers `0`, ..., `5` are the *indexes* or *subscripts* of the components. An important feature of these 6 variables is that they are allocated consecutive memory locations in the computer. We can picture this as:

`column` [0]
`column` [1]
`column` [2]
`column` [3]
`column` [4]
`column` [5]

Note: Since array indexes always begin at 0, the array size will always be one more than the index number of the last element.

<u>Assignment Statements and Expressions with Array Elements</u>:

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared. Having declared our array, we can treat the individual elements just like ordinary variables (of type "`int`" in the particular example above). In particular, we can write assignment statements such as

```
        column [4] = 34;
        column [5] = column [4]/2;
```

One way of initialising arrays is to specify values. In the case of a small array as above, this can be easily achieved by

```
int column[6]={1, 8, 2, 3, 12, 0};

The following code

cout<< column [0]<<'\t'<< column [4]<<endl;

cout<< column [2]+ column [3]<<endl;

would then give
```

1 12

5

To initialise **<u>all</u>** elements of the array to zero would simply require `int column [6]={0};`

The most common way to assign values to an array is by using a "for" or "while" loop. The following code assigns each element *i* of the array by the square root of $10i$.

```
  const int size=10;

        float x[size]={0.0};

        for (int i=1; i<=size; i++)
        {
                x[i-1]=sqrt(10.0*i);
        }
```

<u>Duplicating an array</u>: Arrays cannot be assigned, e.g. the statements

```
List2=List1;
List2={12, 19, 32, 76};
```

are both illegal. The important point to remember is that an array name itself (without parameters) is a constant, and it is illegal for a constant to be on the LHS of an assignment statement. The best way to copy one array into another is by using a `for` loop, as shown in the next example:

<u>Copying an array</u>: Here we use the array `x[size]` defined earlier

```
float y[size];
        for ( i=1; i<=size; i++)
        {
                y[i-1]=x[i-1];
        }
```

<u>Reading in an array from the keyboard</u>:

```
#include<iostream>

using namespace std;
int main()
{
        const unsigned int n=10;
        typedef int vector[n];
        vector A={0};  // initialise array to zero
        cout<< "Enter components of array"<<endl;

        for (unsigned int i=1; i<=n; i++)

        {

                cin>>A[i-1];

                int x=A[i-1];


        }


        for (i=1; i<=10; i++)// what is wrong with this line?
        {
                cout<<A[i-1]<<endl;
        }


        return 0;

}
```

So we have now seen the two distinguishable purposes of the pair of brackets [] related to arrays, which are to specify

- the size of arrays when they are declared
- indices for concrete array elements when they are accessed

so e.g.

```
int A[10];        // declaration of a new array A
A[2] = 12;        // access to an element of the array (in this case the third).
```

What does the following program output?

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

int A [] = {166, 23, 747, 340, 123};
int n, sum=0;

int main ()
{
      for ( n=0 ; n<5 ; ++n )
      {
            sum += A[n];
            cout << sum<<endl;
      }
      cout << sum<<endl;


      return 0;
}
```

Mathematical Example:

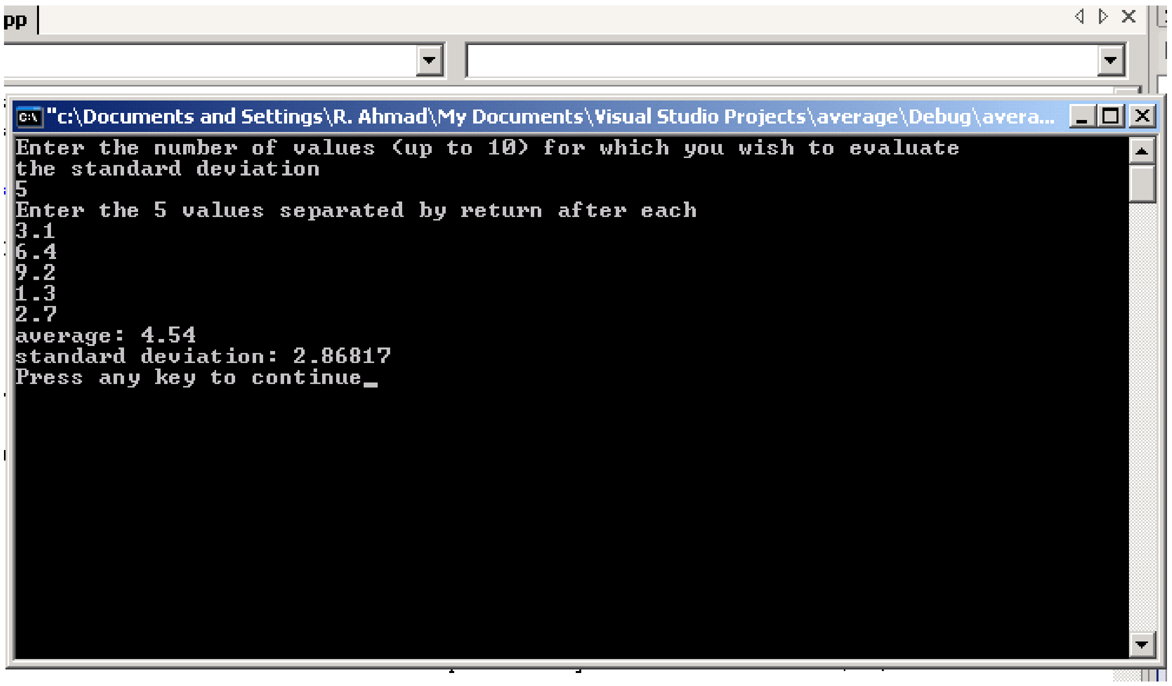
The standard deviation of a sequence of numbers $\{x_j\}$ $i = 0,.......,n-1$ with mean average μ, is given by

$$\frac{1}{n}\sqrt{\sum_{i=1}^{n}(x_i-\mu)^2}$$

The following code calculates the average and then standard deviation of a set of numbers (up to ten):

```
#include <iostream>
#include <cmath>

using namespace std;

const unsigned int MAX=10;

int main()
{
	double A[MAX]={0.0};
	double average,ans,st_dev;
	int i,N;
	average=0.0;
	ans=0.0;

cout << "Enter the number of values (up to 10) for which you wish to evaluate "
<<endl; // ensure this carries over to second line correctly
	cout << 	"the standard deviation" << endl;
	cin >> N;

	cout << "Enter the " << N << " values separated by return after each\n";
	for (i=1; i<=N; i++)
	{
		cin >> A[i-1];
	}
	for (i=1; i<=N; i++) {

		average+=A[i-1]/N;}
	cout << "average: " << average << endl;

	for (i=1; i<=N; i++) {
		ans+=((A[i-1]-average)* (A[i-1]-average));}

	st_dev=sqrt(ans/double(N));

	cout << "standard deviation: " << st_dev << endl;

	return 0;
}
```

The output is then …….



```
Enter the number of values (up to 10) for which you wish to evaluate
the standard deviation
5
Enter the 5 values separated by return after each
3.1
6.4
9.2
1.3
2.7
average: 4.54
standard deviation: 2.86817
Press any key to continue
```

Note: Until you become familiar with using arrays, it is easy to make errors when manipulating them. Common errors include forgetting that arrays are zero-based, that is, begin with element 0, with a knock-on effect being that the highest legitimate subscript is one less than the array size, that is, accessing locations beyond the limits of the array (range bound error). It is important to note this as such errors are not picked up by the compiler but result in running errors which can require more work to rectify.

4.1 Arrays as Parameters in Functions

Functions can be used with array parameters to maintain a structured design. An array is passed to a function in the same way as any other variable, except that the array name must be followed by “[ ]” to indicate that it is the name of an array. In the following program, an array of real numbers is passed to a function, which returns the average – as part of the function parameter list, we include the length of the array.

```
        float mean(float a[], int n)      // n is length of array
{
        float sum= 0;
        float average=0;
        for (int i= 0 ; i< n ; i++){
                sum += a[i];}
        average=sum/n;
        return average;
}
```

Exercise: Re-write the earlier standard deviation program using two functions. A function may be called in any other function – remember `int main()` itself is a function.

Important Note:

Although array parameters are not declared with an "`&`" character in function declarations and definitions, they are effectively reference parameters (rather than value parameters). In other words, when they execute, functions do not make private copies of the arrays they are passed (this would potentially be very expensive in terms of memory).

Hence, like the reference parameters we have seen earlier, arrays can be permanently changed when passed as arguments to functions.

As a safety measure, we can add the modifier "`const`" in the function head:

```
float std_dev(const float a[], int n)
```

The compiler will then not accept any statements within the function's definition that potentially modify the elements of the array.
Here is a program which calculates the standard deviation as earlier, but made up entirely of functions. Note that the main body of the program has been used merely for calling the functions:

```
#include <iostream>
#include <cmath>
```

```
using namespace std;

void  fill_up(double a[], int& ); // read in the numbers
void display(double a[], int& ); // display entered information
double average(double a[], int ); // calculate the average
double stan_dev(double a[], int, double ); // calculate sigma

const int MAX=100;  // will take a max of 100 numbers

int main(){

        int size;
double a[MAX];

fill_up(a,size); // note in function call, no []
display(a,size);
average(a,size);
double mean= average(a,size); /* assign the average calculated above to
                                                       mean */

stan_dev(a,size,mean); /* mean is now passed to function sigma*/


return 0;
}
 /*  _________________________________________________  */

void fill_up(double a[], int& size)
{

cout << "Enter size" << endl;

cin >> size;
cout << "Now Enter numbers" << endl;

for (int i=1; i<=size; i++)
{cin >> a[i-1];
}


}


void display(double a[], int& size)

{
cout << "You have entered the following values: " << endl;
for (int i=1; i<=size; i++)
cout << a[i-1] << endl;
}



double average(double a[],int size)
{  double av=0.0;

for (int i=1; i<=size; i++)
{
av+=a[i-1];}

double mean=av/size;
cout << "The average is " << mean << endl;

return mean;
}

double stan_dev(double a[], int size, double mean)
{

 double sum=0.0;
 for (int i=1; i<=size; i++)
{
sum+=((a[i-1]-mean)*(a[i-1]-mean));

}
double std=sum/size;
cout << sqrt(std) << endl;
return 0;

}
```

<u>Exercise</u>: Write the program above using the `typedef` construct to create a new array type.

4.2 Two-dimensional Arrays

Arrays can have more than one dimension; in theory any number of dimensions. In this section we briefly examine the use of two-dimensional (2D) arrays to represent two-dimensional structures such as (n x m) matrices.

A 2D array is an array that has two independent subscripts. So the matrix A_{ij} would be represented as `A[i][j]` preceded by its type (`int, float`, etc.). The syntax for its declaration is

```
<component type>  <name>[<number of rows>][<number of columns>];
```

Individual elements are accessed in exactly the same way as 1D arrays.

A 2D array can be imagined as an array of 1D arrays.

Examples:

```
int matrix[3][6]; // matrix has 18 elements – 3 rows & 6 columns
matrix[2][4]=12;  // the value 12 has been assigned to the element in
                              row 2 and column 4
```

We can initialise as in the 1D case. For example

```
int matrix[3][6]={0};
```

would initialise all elements of matrix to zero. The initialisation

```
int matrix[3][6]={ {1,2,3,4,5}, {8}, {6, 4, 2} };
```

would then produce the structure

1	2	3	4	5	0
8	0	0	0	0	0
6	4	2	0	0	0

Using `typdef` to declare an array:

We can use `typedef` to create a new data type `matrix`. Suppose we are working with 2D arrays (of equal number of rows and columns, `dimen`, say) whose elements are floats. We can do the following

```
typedef float matrix[dimen][dimen];
```

to create a new data type consisting of (dimen x dimen) matrices. Then every time we wish to declare new matrices (e.g. A, B and C), we would write the following:

```
matrix A, B, C;
```

and also pass these as function parameters, e.g.

```
float product ( …., matrix A, ……..)
```

As with 1D arrays, it is far better to use loops to initialise, which in this case would require nested "`for`" loops – the outer handling the rows and the inner responsible for the columns.

Example: Printing the earlier (3 $\times$ 6) matrix

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

typedef int    matrix[3][6];

int main()
{

        matrix A={ {1,2,3,4,5}, {8}, {6, 4, 2} };
        for (int i=1; i<=3; i++)
        {

                { for (int j=1; j<=6; j++)

                        cout<<A[i-1][j-1]<<'\t';

                cout<<endl;
                }

        }

        return 0;
}
```
gives the output

```
1       2       3       4       5       0
8       0       0       0       0       0
6       4       2       0       0       0
```

If we wanted to read in values to the array (above) then the following loops are required:

```
for (int i=1; i<=3; i++)
        {
                { for (int j=1; j<=6; j++)

                        cin>>A[i-1][j-1];

                }

        }
```

4.3 Reading From a File

The idea of reading from and to a file was considered in chapter 2. Suppose as an example we wish to read data from a text file (called test.txt) in to an array in the program written in section 3.2 where a basic function was presented to calculate the average of three variables.

The following fragment will do this:

```
int a[10]={0};

        int k;

    fstream input;


input.open
("I:\\Documents and Settings\\Riaz\\My Documents\\Visual Studio
Projects\\aver\\test.txt", ios::in);

        while (!input.eof()){

        for(k=1; k<=10; k++){
    input>> a[k-1];}
             }
```

4.4 Options Pricing Example:

Suppose as an example we were pricing a lookback option, whose price depends upon the maximum/minimum value of the underlying - we would have a column of data, which represents the asset price.

Computationally, this converts to a problem whereby we have a one-dimensional array, which we must traverse to find the maximum or minimum entry. Obviously in a Monte Carlo pricing exercise this would be then repeated several times. The following code, consists of two functions – one to pick out the maximum entry from an array passed through, the other to obtain the minimum.

`main.cpp` **Main Body**

```
#include<iostream>
#include<algorithm>
#include "max_min.h"

using namespace std;

int main()

{

        int A[dimen]={5,67,2098,3,23,43,98,12,3,12};// define matrix


        cout<<"The largest element is "<< Maximum(A)<<endl;
```

```
      cout<<"The smallest element is "<<Minimum(A)<<endl;

      /* call the 2 functions for the Max. and Min. entries, and output
to
      screen */

        return 0;

}
```

`max_min.h` **Header file**

```
#ifndef MAX_MIN_H
#define MAX_MIN_H

const int dimen=10;

typedef int matrix[dimen];

int Maximum(matrix );
int Minimum(matrix );

/* The 2 lines above are the function prototypes/declarations, and read in
a 1-D array of type matrix (which we have defined ourselves */


#endif
```

`max_min.cpp` **Function Definitions for both Maximum and Minimum**

```
#include<iostream>
#include<algorithm>
#include "max_min.h"

using namespace std;

int Maximum(matrix A )

{
        int maximum;

        maximum=A[0];  // set maximum equal to the first array element

        for (int i=1; i<=dimen; i++) // now go along each element

        {maximum=_MAX(maximum,A[i-1]);} /* compare each ith element to the
(i+1)th and set maximum to the greater of the two. Keep doing this till
the end of the array is reached */


        return maximum;  // largest element of array is returned.
}
```

```
/* In the next function, exactly the same idea is used, this time to pick
out the smallest component of the array. */
int Minimum(matrix A)
{

        int minimum;
        minimum=A[0];

        for (int i=1; i<=dimen; i++)
        {minimum=(minimum < A[i-1])? minimum:A[i-1];}

        return minimum;

}
```

The output to obtain the maximum and minimum, in turn, from {5, 67, 2098, 3, 23, 43, 98, 12, 3, 12} would be

```
The largest element is 2098
The smallest element is 3
```

4.5 Gaussian Elimination Method

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


typedef const unsigned short int ushort;
ushort dimen=4;
ushort max_=3;
typedef double matrix[dimen][dimen] ;
typedef double vector[dimen] ;


double gauss (matrix , vector, vector, ushort);


int main()
{


        matrix A = {{10,-1,2,0}, {-1,11,-1,3}, {2,-1,10,-1},
        {0,3,-1,8}};
        vector b={6,25,-11,15};
        vector x={0.0,0.0,0.0,0.0};

        gauss (A, x, b,dimen);

        return 0;

}


double gauss (matrix A, vector x, vector b, ushort dimen)

{


        double pivot,sum;



        for (int k=0; k<=max_-1; k++)
        {
                for (int l=k+1; l<=max_; l++)
                {

                        pivot=A[l][k]/A[k][k];



                        for  (int j=k; j<=max_;j++)

                        {

                                A[l][j]-=A[k][j]*pivot;


                        }
                b[l]-=b[k]*pivot;


            }

        }


        x[max_]=b[max_]/A[max_][max_];


        for (int k=max_-1; k>=0; k--)
        {
                sum=b[k];


                for (int j=k+1; j<=max_; j++)
                {

                        sum-=x[j]*A[k][j];
                }

                x[k]=sum/A[k][k];



        }

        for (int i=1; i<=dimen; i++)
                {

                        cout<<x[i-1]<<endl;
                }


        return 0;

}
```

4.6 Gauss Seidel Iterative Scheme

```
#include<iostream>

using namespace std;

const unsigned short int n=4;
typedef double matrix[n][n] ;
typedef double vector[n] ;
double Gauss(matrix, vector, vector, vector);

int main()
{
        matrix A = {{10,-1,2,0}, {-1,11,-1,3}, {2,-1,10,-1},
        {0,3,-1,8}};
        double b[4]={6,25,-11,15};
        double X_old[n]={1.0,1.0,1.0,1.0};
        double X[4];
        int Iter=7;  //try varying this

        for (int i=0; i<n; i++){
                X[i]=X_old[i];}

        double Tol=0.00001;

    for (int iteration=1; iteration<=Iter; iteration++){

        cout<<endl;
        cout<<"Iteration No.= "<<iteration<<endl;
        cout<<endl;
        Gauss(A,b,X,X_old);



}


        return 0;

}



double Gauss(matrix A, vector b, vector X, vector X_old)
{

for (int i=0; i<n; i++)
        {


                double sum1=0.0;
                double sum2=0.0;

                for (int j=0; j<=i-1; j++)

                {

                        sum1+=-A[i][j]*X[j];
                }

                for (j=i+1; j<n; j++)
                {

                        sum2+=-A[i][j]*X_old[j];

                }

X[i]=(sum1+sum2+b[i])/A[i][i];



        X_old[i]=X[i];
}

        for (int i=0; i<4; i++){

                cout<<X[i]<<endl;

        }

return 0;
        }
```

As an exercise, modify the code so as to include some form of convergence criteria.

The program below is a reworking of the earlier Normal CDF code, now using arrays to define the parameters. It is written in terms of header and source files

normal.h

```
#ifndef NORMAL_H
#define NORMAL_H

double NormalDensity(double x);

double CumulativeNormal(double x);

double InverseCumulativeNormal(double x);

#endif
```

main.cpp
```
#include <cmath>
#include <Normal.h>

using namespace std;


const double OneOverRootTwoPi = 0.398942280401433;

// probability density for a standard Gaussian distribution
double NormalDensity(double x)
{
    return OneOverRootTwoPi*exp(-x*x/2);
}

// the InverseCumulativeNormal function via the Beasley-Springer/Moro
approximation
double InverseCumulativeNormal(double u)
{


    static double a[4]={  2.50662823884,
                        -18.61500062529,
                         41.39119773534,
                        -25.44106049637};

    static double b[4]={-8.47351093090,
                        23.08336743743,
                       -21.06224101826,
                         3.13082909833};

    static double c[9]={0.3374754822726147,
                        0.9761690190917186,
                        0.1607979714918209,
                        0.0276438810333863,
                        0.0038405729373609,
                        0.0003951896511919,
                        0.0000321767881768,
                        0.0000002888167364,
                        0.0000003960315187};


    double x=u-0.5;
    double r;

    if (fabs(x)<0.42) // Beasley-Springer
    {
        double y=x*x;

        r=x*(((a[3]*y+a[2])*y+a[1])*y+a[0])/
                ((((b[3]*y+b[2])*y+b[1])*y+b[0])*y+1.0);

    }
    else // Moro
    {

        r=u;
        if (x>0.0)
            r=1.0-u;

        r=log(-log(r));

        r=c[0]+r*(c[1]+r*(c[2]+r*(c[3]+r*(c[4]+r*(c[5]+r*(c[6]+
                r*(c[7]+r*c[8])))))));

        if (x<0.0)
            r=-r;

    }

    return r;
}


// standard normal cumulative distribution function
double CumulativeNormal(double x)
{
    static double a[5] = { 0.319381530,
                          -0.356563782,
                           1.781477937,
                          -1.821255978,
                           1.330274429};

    double result;

    if (x<-7.0)
        result = NormalDensity(x)/sqrt(1.+x*x);

    else
    {
        if (x>7.0)
            result = 1.0 - CumulativeNormal(-x);
        else
        {
            double tmp = 1.0/(1.0+0.2316419*fabs(x));

            result=1-NormalDensity(x)*

(tmp*(a[0]+tmp*(a[1]+tmp*(a[2]+tmp*(a[3]+tmp*a[4])))));

            if (x<=0.0)
                result=1.0-result;

        }
    }

    return result;
}
```

4.7 Introduction to Strings

We have already been using string values, such as **`"Enter num: "`**, etc. in programs involving output to the screen. In C++ you can store and manipulate such values in ***string variables***, which are simply arrays of characters, but used in a particular way.

The Sentinel String Character '\0'

The key point is that, to use the special functions associated with strings, string values can only be stored in string variables whose length is at least 1 greater than the length (in characters) of the value.

This is because extra space must be left at the end to store the *sentinel string character* "`'\0'`" which marks the end of the string value. For example, the following two arrays both contain all the characters in the string value "`Enter num: `", but only the array on the left contains a proper string representation.

Phrase		list	
‘E’	phrase[0]	‘E’	list[0]
‘n’	phrase[1]	‘n’	list[1]
‘t’	phrase[2]	‘t’	list[2]
‘e’	phrase[3]	‘e’	list[3]
‘r’	phrase[4]	‘r’	list[4]
‘ ’	phrase[5]	‘ ’	list[5]
‘n’	phrase[6]	‘n’	list[6]
‘u’	phrase[7]	‘u’	list[7]
‘m’	phrase[8]	‘m’	list[8]
‘:’	phrase[9]	‘:’	list[9]
‘ ’	phrase[10]	‘ ’	list[10]
‘\0’	phrase[11]		
?	phrase[12]		
?	phrase[13]		

Figure 4.7

In other words, although both "`phrase`" and "`list`" are arrays of characters, only "`phrase`" is big enough to contain the string value "`Enter num: `". We do not care what characters are stored in the variables "`phrase[12]`" and "`phrase[13]`", because all the string functions introduced later ignore characters after the "`'\0'`".

4.8 String Variable Declarations and Assignments

String variables can be declared just like other arrays:

```
        char phrase[14];
```

String arrays can be initialised or partially initialised at the same time as being declared, using a list of values enclosed in "{}" braces (the same is true of arrays of other data types). For example, the statement

```
char phrase[14] = {'E','n','t','e','r',' ','n','u','m',':',' ','\0'};
```

both declares the array "phrase" and initialises it to the state in Figure 4.7. The statement

```
        char phrase[14] = "Enter num: ";
```

is equivalent. If the array size "14" is omitted, an array will be created just large enough to contain both the value " "Enter num: " " and the sentinel character "'\0'", so that the two statements

```
        char phrase[] = {'E','n','t','e','r',' ','n','u','m',':','
','\0'};
char phrase[] = "Enter num: ";
```

are equivalent both to each other and to the statement

```
        char phrase[12] = "Enter num: ";
```

However, it is important to remember that string variables are arrays, so we cannot just make assignments and comparisons using the operators "=" and "= =". We cannot, for example, simply write

```
        phrase = "You typed: ";
```

Instead, we can use a special set of functions for string assignment and comparison.

<u>Example</u>:

Like integer and real types, strings can be input and output using the insertion and extraction operators >> and <<. Consider a string called `word` of length 20. In terms of memory, 20 bytes would be allocated to this array. Consider the following fragment of code:

```
#include<iostream>
```

```
using namespace std;

int main()
{
        char word[20]="ABCDEFG"; /* declare word as a string with 20 cpts
                                 (0- 19) */

        cout<<word<<endl;  /* strings can be output like simple types i.e.
                          without the index  */

        word[4]='*';        // although strings are arrays
        cout<<word<<endl;
        cin>>word;                 // again strings can be input like
simple types
        cout<<word<<endl;

        return 0;

}
```

and corresponding output (input from the keyboard is distinguished by **boldface**):

```
ABCDEFG
ABCD*FG

Hello, Maggie

Hello,
```

Note how the extraction operator has ignored whitespace (including blanks, tabs and new lines) that precedes the input string, and stops extracting characters as soon as it encounters a whitespace character. This is the reason why only the six characters "`Hello,`" were read into `word`. The rest of the input "`Maggie`" is still in the input buffer.

4.9 Some Predefined String Functions

The library `cstring` (old style header `string.h`) contains a number of useful functions for string operations. We will assume that the program fragments discussed below are embedded in programs containing the "include" statement

```
#include<cstring>
```

Given the string variable "`a_string`", we can copy a specific string value or the contents of another string to it using the two argument function "`strcpy(...)`". Hence the statement

```
        strcpy(a_string, "You typed: ");
```

assigns the first 11 elements of "`a_string`" to the respective characters in "`"You typed: "`", and assigns the sentinel character "`'\0'`" to the 12th element. The call

```
        strcpy(a_string, another_string);
```

copies the string value stored in "`another_string`" to "`a_string`". But care has to be taken with this function. If "`a_string`" is less than (1 + L), where L is the length of the string value currently stored in "`another_string`", the call to the function will cause a range bound error, which will not be detected by the compiler.

We can, however, check the length of the value stored in "`another_string`" using the function "`strlen(...)`". The call "`strlen(another_string)`" returns the length of the current string stored in "`another_string`" (the character "`'\0'`" is not counted).

The comparison function "`strcmp(...)`" returns "False" (i.e. 0) if its two string arguments are the same, and the two argument function "`strcat(...)`" concatenates its second argument onto the end of its first argument. Program 4.10 illustrates the use of these functions. Again, care must be taken with "`strcat(...)`". C++ does not check that the first variable argument is big enough to contain the two concatenated strings, so that once again there is a danger of undetected range bound errors.

4.10 String Input using "`getline(...)`"

Although the operator ">>" can be used to input strings (e.g. from the keyboard), we have seen its use is limited because of the way it deals with space characters. Supposing a program which includes the statements

```
        ...
        ...
        cout << "Enter name: ";
        cin >> a_string;
        ...
        ...
```

results in the input/output session

```
        ...
        ...
        Enter name: Riaz Ahmad
        ...
        ...
```

The string variable will then contain the string value ""Riaz"", because the operator ">>" assumes that the space character signals the end of input. It is therefore often better to use the two argument function "getline(...)". For example, the statement

```
        cin.getline(a_string,80);
```

allows the user to type in a string of up to 79 characters long, including spaces. (The extra element is for the sentinel character.) The following program illustrates the use of "getline(...)", "strcmp(...)", "strcpy(...)" and "strcat(...)":

```
#include <iostream>
#include <cstring>

using namespace std;
const int MAXIMUM_LENGTH = 80;

int main()

{
        char first_string[MAXIMUM_LENGTH];
        char second_string[MAXIMUM_LENGTH];

        cout << "Enter first string: ";
        cin.getline(first_string,MAXIMUM_LENGTH);

        cout << "Enter second string: ";
        cin.getline(second_string,MAXIMUM_LENGTH);

        cout << "Before copying the strings were ";

        if (strcmp(first_string,second_string))
                cout << "not ";
        cout << "the same.\n";
        strcpy(first_string,second_string);

        cout << "After copying the strings were ";
        if (strcmp(first_string,second_string))
                cout << "not ";
        cout << "the same.\n";

        strcat(first_string,second_string);

        cout << "After concatenating, the first string is: ";
        cout << first_string;

        return 0;
}
```

Program 4.10

An example input/output session is:

```
        Enter first string: G'day Sport.
        Enter second string: G'day Riaz.
        Before copying the strings were not the same.
        After copying the strings were the same.
        After concatenating, the first string is: G'day Riaz.G'day Riaz.
```

Example: The following (again) uses some of the predefined functions in the library cstring.

```
#include <iostream>
#include <cstring>

using namespace std;


int main()
{
 char name[30]= "That is it "; /* Initialised a string with 30 slots, ie
                               array holding characters to a message. */

 char another_string[80];      /* Declaration of another to hold 80
                                  chars, not initialised. */
 strcpy(name, "Riaz Ahmad");   /* This takes string "name", and copies
                                  another string over it including
                                  sentinel O*/

 cout << name;                           /* To print out string, do not
need [] */

 cout << " is of length "<<strlen(name) << endl; /* This built in func.
                                                  counts length of string,
                                                  ignoring \O */
 cout << "Enter a name: " ;

 cin.getline(another_string, 80); /* cin >> stops executing at a space,
              so cin acts on func getline which takes the name of string
                                        together with its max length*/

 cout << another_string << endl;

 if ((strcmp(name,another_string))) /* Compares 2 strings to check
                               equivalence, if so returns False (0) */

cout << "Goodbye";         // comes here when different strings
 else
        cout << "hello"<<endl;  // comes here when same

 cout << strcat(name,another_string); /* This attaches 2nd string on to
                                      the end of 1st, without gap */


return 0;


}
```

5. Pointers and Applications

5.0 Introducing Pointers

We have seen in chapter 4 that C++ also allows the creation of arrays (or lists) of variables that can be used to store related values efficiently. Once proficiency is gained, these variables can be characters, i.e. manipulate character strings of data. C++ is a powerful programming language for a number of reaons. A chief strength lies in being able to examine and manipulate the memory addresses of variables.

In earlier chapters, we have defined and discussed data types in a simple fashion and seen how variables maybe considered as cells within the memory, and can be accessed using their identifiers. We have not seriously considered methods to control the amount of memory used in a program or the physical location of our data. In particular, in all of the programs we have looked at so far, a certain amount of memory is reserved for each declared variable at compilation time, and this memory is retained for the variable as long as the program or block in which the variable is defined is active. If for example, the size of an array can become smaller during run time, then clearly it is inefficient to define a larger array at compile time. Is there any feature of C++ that allows the user to define an array during the course of executing a programme?
We now look at the low-level aspects of C++ that it has inherited from C. *Pointers* are one of the most powerful aspects of C++ (and C). They are also the hobgoblin of this language; a simple idea which also inspires much perplexity. In this chapter, our aim is to demystify the notion of a *pointer*, show practical uses for them and demonstrate how a programmer can achieve a greater level of control over the way the program allocates and de-allocates memory during its execution. This provides C++ users with a very useful memory management feature. Every variable lives at a memory address. A pointer is a special data type that stores such an address.

5.1 Declaring Pointers

A **pointer** is simply the **memory address of a variable**, so that a *pointer variable* is just a variable in which we can store different memory addresses. The idea may seem simple enough, but why are we concerned with addresses that are essentially incomprehensible looking sequences of characters and integers? Recall that the memory of a computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one.
Every variable is located under a unique location within a computer's memory and this unique location has its own address, i.e. the memory address. Normally, variables hold values such as 5 or “hello” and these values are stored under specific location within computer memory. However, a pointer is noticeably different, because it holds the memory address as its value and has an ability to “point” (hence pointer) to certain value within a memory, by use of its associated memory address. For students it can also be a major source of confusion. Every variable that is declared has an address, just like the address of a house, and we can find out what address a particular variable is using.

This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1352 we know that it is going to be right between cells 1351 and 1353, exactly one thousand cells after 352 and exactly one thousand cells before cell 2352.

Pointer variables are declared using a "*", and have data types like the other variables we have seen. Pointers are widely used in C++ to facilitate efficient dynamic processing of data. An asterisk placed between the data type and the variable name means the variable is being declared as a pointer.

For example, the declaration

```
int* number_ptr;
```

states that "number_ptr" is a pointer variable that can store addresses of variables of data type "int".

A useful alternative way to declare pointers is using a "typedef" construct. For example, if we include the statement:

```
typedef int* IntPtr;
```

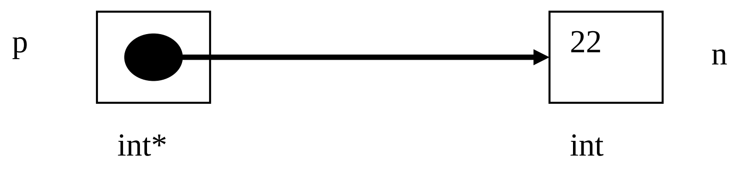
we can then go on to declare several pointer variables in one line, without the need to prefix each with a "*":

```
IntPtr number_ptr1, number_ptr2, number_ptr3;
```

Assignments with Pointers Using the Operators "*" and "&"

Given a particular data type, such as "int", we can write assignment statements involving both ordinary variables and pointer variables of this data type using the *dereference operator* "*" and the (complementary) *address-of operator* "&". Roughly speaking, "*" means "the variable located at the address", and "&" means "the address of the variable".

Example: A pointer to an int conceptually looks like



So what is in a pointer? Referring to the last diagram, the following piece of code

```
cout << " Address is: " << p << endl;
```

would output: `Address is:0x00CBF748`

We can illustrate the uses of these operators with a simple example program:

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

typedef int * IntPtr;

int main()
{
	IntPtr ptr_a, ptr_b;
	int num_c = 4, num_d = 7;

	ptr_a = &num_c;			/* LINE 11 */

	ptr_b = ptr_a;				/* LINE 12 */

	cout << *ptr_a << " " << *ptr_b << "\n";

	ptr_b = &num_d;			/* LINE 16 */

	cout << *ptr_a << " " << *ptr_b << "\n";

	*ptr_a = *ptr_b;			/* LINE 20 */

	cout << *ptr_a << " " << *ptr_b << "\n";

	cout << num_c << " " << *&*&*&num_c << "\n";
	system("pause");
	return 0;
}
```

Program 5.1.1

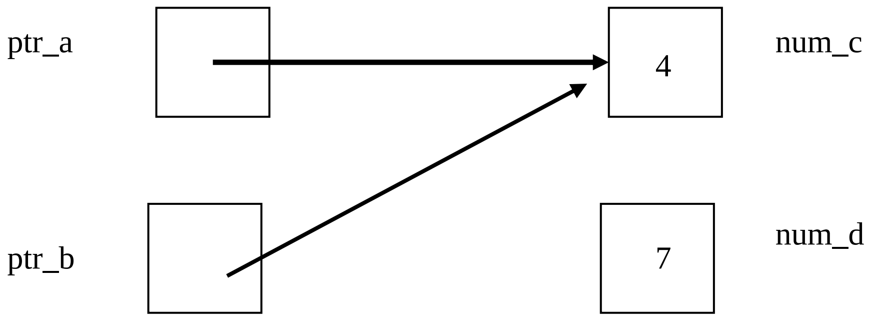
Note: the type of pointer has to match the type of the variable being pointed to. That is e.g. `int*` cannot point to a variable of type float or char. So in the program above, `num_c` cannot be assigned the value of a real number. When declaring a point, it makes no difference if the asterisk is placed next to the data type, the variable name, or in the middle. The location is a personal preference with no particular style being superior. So the following are all equivalent declarations

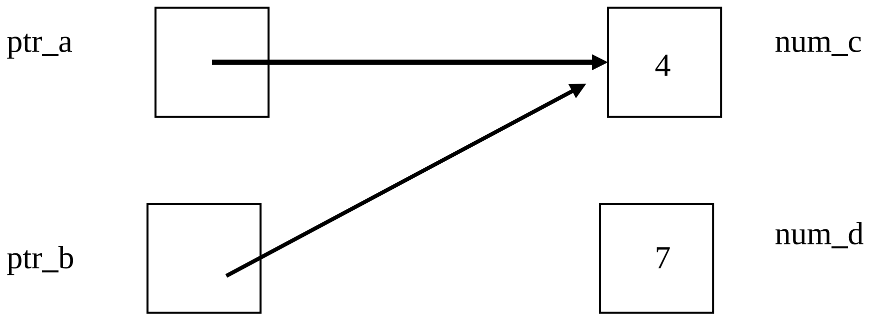
```
int* IntPtr

int * IntPtr

int  *IntPtr
```

The output of this program is:

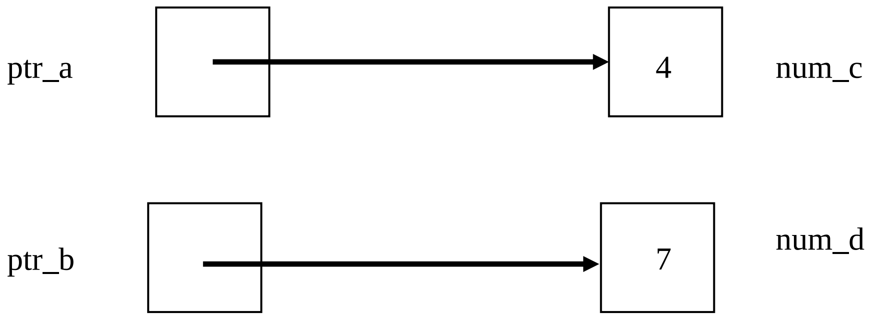
```
4 4
4 7
7 7
7 7
```

Diagrammatically, the state of the program after the assignments at lines 11 and 12 is:

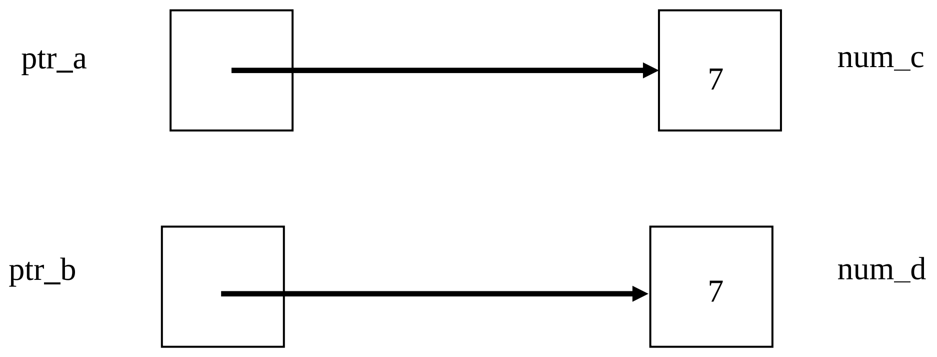




after the assignment at line 16 this changes to:



and after the assignment at line 20 it becomes:

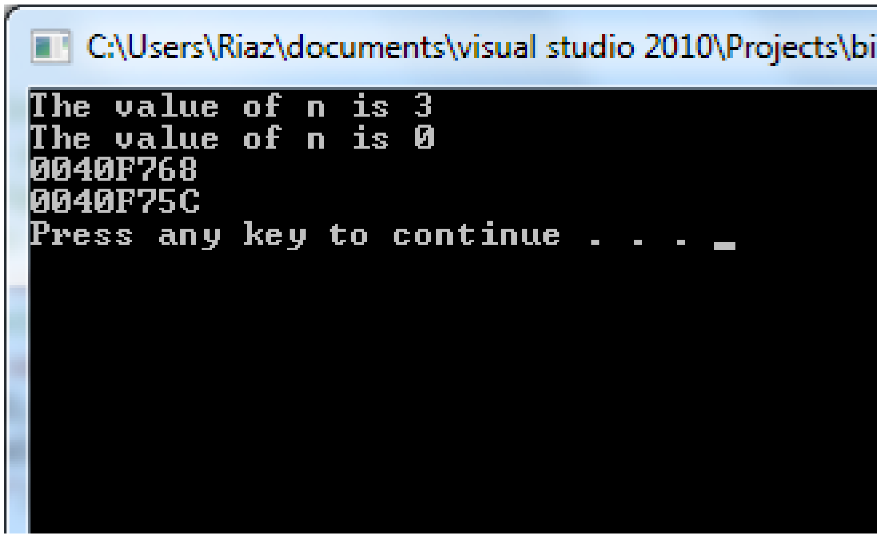


Note that "`*`" and "`&`" are in a certain sense complementary operations; "`*&*&*&num_c`" is simply "`num_c`".

Exercise: What is the output from the following program?

```
#include <iostream>
using namespace std;
int main() {
       int n;
       int *np;
       n = 3;
       np = &n;
       cout << "The value of n is " << *np << endl;
       *np = 0;
       cout << "The value of n is " << n << endl;
       cout<<np<<endl;
       cout<<&np<<endl;
       system("pause");
       return 0;
}
```

The output is explained on the next page.


C:\Users\Riaz\documents\visual studio 2010\Projects\bi

```
The value of n is 3
The value of n is 0
0040F768
0040F75C
Press any key to continue . . .
```

```cpp
#include <iostream>
using namespace std;
int main() {
	int n; // declare an integer n
	int *np; // declare a pointer to an integer (not pointing anywhere)
	n = 3; // assignment to n, the value of 3
	np = &n; // pointer np assigned the address of integer n – points to n=3
	cout << "The value of n is " << *np << endl; // outputs '3'
	*np = 0; // write 0 to n
	cout << "The value of n is " << n << endl; // outputs '0'
	cout<<np<<endl; // prints out address of variable n
	cout<<&np<<endl; // prints out address of pointer to n, i.e. np
	system("pause");
	return 0;
}
```

The "`new`" and "`delete`" operators, and the constant "`NULL`".

In program 5.1.1, the assignment statement

```
ptr_a = &num_c;
```

(in line 11) effectively gives an alternative name to the variable "`num_c`", which can now also be referred to as "`*ptr_a`". As we shall see, it is often convenient (in terms of memory management) to use dynamic variables in programs. These variables have no independent identifiers, and so can only be referred to by de-referenced pointer variables such as "`*ptr_a`" and "`*ptr_b`".

Dynamic variables are "created" using the reserved word "`new`", and "destroyed" (thus freeing-up memory for other uses) using the reserved word "`delete`". Below is a program analogous to 5.1.1, which illustrates the use of these operations:

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

      typedef int* IntPtr;

      int main()
      {
            IntPtr ptr_a, ptr_b;    /* LINE 8 */

            ptr_a = new int;            /* LINE 10 */
            *ptr_a = 4;
            ptr_b = ptr_a;              /* LINE 12 */

            cout << *ptr_a << " " << *ptr_b << "\n";

            ptr_b = new int;            /* LINE 16 */
            *ptr_b = 7;                 /* LINE 17 */

            cout << *ptr_a << " " << *ptr_b << "\n";

            delete ptr_a;
            ptr_a = ptr_b;              /* LINE 22 */

            cout << *ptr_a << " " << *ptr_b << "\n";

            delete ptr_a;               /* LINE 26 */
            return 0;
      }
```

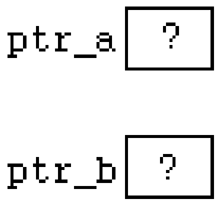
Program 5.1.2

The output of this program is:

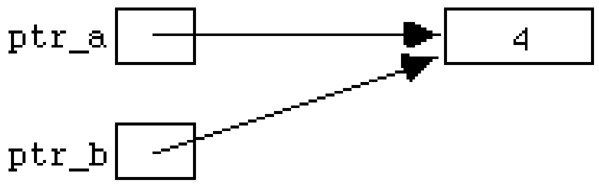
```
4 4
4 7
7 7
```

What is happening here (diagrammatically)?

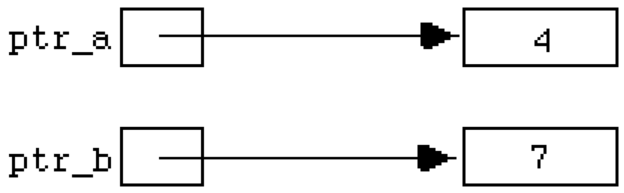
The state of the program after the declarations in line 8 is:



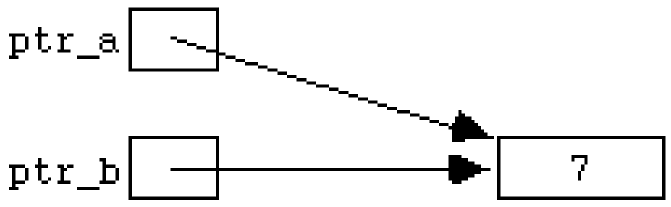
after the assignments in lines 10, 11 and 12 this changes to:



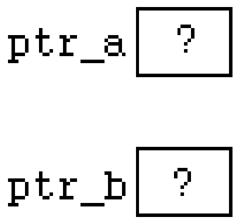
after the assignments at lines 16 and 17 the state is:



and after the assignment at line 22 it becomes:



Finally, after the "delete" statement in lines 26, the program state returns to:



In the first and last diagrams above, the pointers "ptr_a" and "ptr_b" are said to be dangling. Note that "ptr_b" is dangling at the end of the program even though it has not been explicitly included in a "delete" statement. **Note:** If *n* pointers point to a dynamic variable which is deleted, the result is *n* dangling pointers.

If "ptr" is a dangling pointer, use of the corresponding dereferenced expression "*ptr" produces unpredictable (and sometimes disastrous) results; most commonly the programmer has failed to place a valid address in a pointer variable. This results in the pointer potentially being directed to anywhere in memory, i.e. pointing to the code or to the operating system.

Unfortunately, C++ does not provide any inbuilt mechanisms to check for dangling pointers. However, safeguards can be added to a program using the special symbolic memory address "NULL".
Any pointer of any data type can be set to "NULL". For example, if we planned to extend Program 5.1.2 and wanted to safeguard against inappropriate use of the dereferenced pointer identifiers "*ptr_a" and "*ptr_b", we could add code as follows:

```
#include <iostream>
...
...
        delete ptr_a;
        ptr_a = NULL;
        ptr_b = NULL;
        ...
        ...
        if (ptr_a != NULL)
        {
                *ptr_a = ...
                ...
                ...
```

Fragment of Program 5.1.3

In the case that there is not sufficient memory to create a dynamic variable of the appropriate data type after a call to "new", C++ automatically sets the corresponding pointer to "NULL". Hence the following code typifies the kind of safety measure that might be included in a program using dynamic variables:

```
#include <iostream>
#include <cstdlib>   /* ("exit()" is defined in <cstdlib>) */
...
...
        ptr_a = new int;
        if (ptr_a == NULL)
        {
                cout << "Sorry, ran out of memory";
                exit(1);
        }
        ...
        ...
```

Fragment of Program 5.1.4

Pointers can be used in the standard way as function parameters, so it would be even better to package up this code in a function:

```
        void assign_new_int(IntPtrType &ptr)
        {
                ptr = new int;
                if (ptr == NULL)
                {
                        cout << "Sorry, ran out of memory";
                        exit(1);
                }
        }
                        Fragment of Program 5.1.5
```

5.2 Array Variables and Pointer Arithmetic

In a previous lecture we saw how to declare groups of variables called arrays. By adding the statement

```
        Int T[6];
```

we could then use the identifiers

```
        T[0]   T[1]   T[2]   T[3]   T[4]   T[5]
```

as though each referred to a separate variable.

Experienced C++ programmers use pointers for many purposes. A powerful use of pointers is an alternative to an array name. Recall that any array name is actually a memory address, and that pointers hold memory addresses. Therefore, a pointer also can hold an array name, provided the pointer type and array type are the same.

In fact, C++ implements arrays simply by regarding array identifiers such as "T" as pointers. Thus if we add the integer pointer declaration

```
        int* ptr;
```

to the same program, it is now perfectly legal to follow this by the assignment

```
        ptr = T;
```

After the execution of this statement, both "ptr" and "T" point to the integer variable referred to as "T[0]". Thus "T[0]", "*T", and "*ptr" are now all different names for the same variable. The variables "T[1]", "T[2]", etc. now also have alternative names. We can refer to them either as

```
        *(T + 1)   *(T + 2)   ...
```

or as

```
*(ptr + 1)   *(ptr + 2)   ...
```

In this case, the "`+ 2`" is shorthand for "plus enough memory to store 2 integer values". We refer to the addition and subtraction of numerical values to and from pointer variables in this manner as pointer arithmetic. Multiplication and division cannot be used in pointer arithmetic, but the increment and decrement operators "`++`" and "`--`" can be used, and one pointer can be subtracted from another of the same type.
Pointer arithmetic gives an alternative and sometimes more succinct method of manipulating arrays.

Exercise: Run the following code

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

int main ()
{

       // an array with 5 elements.
       double T[5] = {1.3, 6.9, 8.4, 16.3, 50.0};
       double *p;

       p = T;

       // output each array element's value
       cout << "Array values using pointer " << endl;
       for ( int i = 0; i < 5; i++ )
       {
              cout << "*(p + " << i << ") : ";
              cout << *(p + i) << endl;
       }

       cout << "Array values using T as address " << endl;
       for ( int i = 0; i < 5; i++ )
       {
              cout << "*(T + " << i << ") : ";
              cout << *(T + i) << endl;
       }


       system("pause");
       return 0;
}
```

5.3 Passing by Pointer

Pointers provide us with a mode for passing variables to functions – it is similar to passing by reference, so the function can change the variable value outside its body. Whilst passing by pointer is a concept from C, the idea of passing by reference is "essentially" part of C++.

When using passing by pointer, it is important to note that the addresses of the variables must be given to the function, rather than the variables themselves.

The function is unable to alter the value of the pointer itself, i.e. the address (of the variable) remains fixed in memory. However, the function will now be able to change the value in memory that the pointer points to. Within the function, to access the variable (i.e. the value held at the address), the pointer must be de-referenced. Consider the following code consisting of two functions, which pass by value and by pointer. The value of the variable passed is changed by the latter. Note the difference in syntax when passing a standard variable by pointer, the address must be provided to the function using the address operator &.

```
include <iostream>
using namespace std;

void add_to_byvalue(double x);
void add_to_bypointer(double *x);

int main() {
double z = 4.5;
add_to_byvalue(z); // pass z
cout << z << endl; // '4.5'
add_to_bypointer(&z); // pass &z, ie the address
cout << z << endl; // '5.5'
return 0;
}

void add_to_byvalue(double x)
{ x++; } // does not change x outside

void add_to_bypointer(double *x)
{(*x)++; } // does change x outside
```

A subtle but important point is in the use of the brackets in the expression (*x)++, because of the operator precedence, without which interpretation of the statement would have been *(x++), which will produce garbage and will probably cause your program to terminate.

5.4 Dynamic Arrays – Dynamic memory allocation

A major application of pointers is in the dynamic allocation of memory at run time. We have seen that when declaring an array, its size must be stated in order to avoid a compile time error. Very often we are uncertain about the precise size of an array prior to the running of the program. For this reason C/C++ supports the invaluable property of dynamic memory allocation, i.e. the ability to allocate memory at run time.

The mechanisms described above to create and destroy dynamic variables of type "`int`", "`char`", "`float`", etc. can also be applied to create and destroy dynamic arrays. This can be especially useful since arrays sometimes require large amounts of memory.

A dynamic array of 10 integers can be declared as follows:

```
  int *number_ptr;
number_ptr = new int[10];
```

As we have seen, array variables are really pointer variables, so we can now refer to the 10 integer variables in the array either as

```
number_ptr[0]   number_ptr[1]   ....            number_ptr[9]
```

or as

```
*number_ptr   *(number_ptr + 1)   ....  *(number_ptr + 9)
```

To destroy the dynamic array, we write

```
delete [] number_ptr;
```

The "`[]`" brackets are important. They signal the program to destroy all 10 variables, not just the first. To illustrate the use of dynamic arrays, here is a program fragment that prompts the user for a list of integers, and then prints the average on the screen:

```
...
...
int no_of_integers, *number_ptr;

cout << "Enter number of integers in the list: ";
cin >> no_of_integers;

number_ptr = new int[no_of_integers];
if (number_ptr == NULL)
{
        cout << "Sorry, ran out of memory.\n";
        exit(1);
}

cout << "type in " << no_of_integers;
cout << " integers separated by spaces:\n";
```

```
for (int count = 0 ; count < no_of_integers ; count++)
        cin >> number_ptr[count];
cout << "Average: " << average(number_ptr,no_of_integers);

delete [] number_ptr;
...
...
```

Fragment of Program 5.4.1

Dynamic arrays can be passed as function parameters just like ordinary arrays, so we can simply use the definition of the function "average()" from an earlier lecture with this program.

5.5 Automatic and Dynamic Variables

Although dynamic variables can sometimes be a useful device, the need to use them can often be minimised by designing a well-structured program, and by the use of functional abstraction. Most of the variables we have been using in the previous lectures have been *automatic* variables. That is to say, they are automatically created in the block or function in which they are declared, and automatically destroyed at the end of the block, or when the call to the function terminates. So, for a well structured program, much of the time we do not even have to think about adding code to create and destroy variables.

N.B. It is also possible to declare variables as being *static*, i.e. remaining in existence throughout the subsequent execution of the program, but in a well designed, non-object based program it should not be necessary to use any static variables other than the constants declared at the beginning.

A static variable has the visibility of a local variable (that is inside the function containing it) and its lifetime is similar to that of an external variable, except it doesn’t come into existence until the first call to the function containing it. Thereafter it remains in existence for the life of the program.

Exercise: Predict the output of the following program. Then run the program to see if your prediction is correct.

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

      typedef int *IntPtrType;

      int main()
      {
            IntPtrType ptr_a, ptr_b, *ptr_c;

            ptr_a = new int;
            *ptr_a = 3;
            ptr_b = ptr_a;
            cout << *ptr_a << " " << *ptr_b << "\n";

            ptr_b = new int;
            *ptr_b = 9;
            cout << *ptr_a << " " << *ptr_b << "\n";

            *ptr_b = *ptr_a;
            cout << *ptr_a << " " << *ptr_b << "\n";

            delete ptr_a;
            ptr_a = ptr_b;
            cout << *ptr_a << " " << *&*&*&*&*ptr_b << "\n";

            ptr_c = &ptr_a;
            cout << *ptr_c << " " << **ptr_c << "\n";

            delete ptr_a;
            ptr_a = NULL;

            return 0;
      }
```

5.6 A two-dimensional Dynamic Array

```
#include <iostream>
#include<cstdlib>

using namespace std;

typedef int **intptr;


int main()

{
        int size,i,j;
        cout << "enter size"<<endl;
        cin>> size;

        intptr matrix_a ;

        matrix_a=new int* [size];

        if (matrix_a==NULL)
        {
                cout << "Error" <<endl;
                exit(1);  // belongs to stdlib
        }

        for ( i=0; i<size; i++)
        {
                matrix_a[i]=new int [size];

        }

        cout<<"Enter components of matrix in row column format"<<endl;

        for ( i=0; i<size; i++)
                for (j=0; j<size; j++)
                {

                        cin >> matrix_a[i][j];

                }

                for ( i=1; i<=size; i++)
                        for (j=1; j<=size; j++)
                        {

                                cout << matrix_a[i-1][j-1]<< '\t'<< i << j<<endl;

                        }
                        for ( i=0; i<size; i++)
                        {
                                delete [] matrix_a[i];

                        }

                        delete [] matrix_a;
```

```
                    system("pause");
                    return 0;
}
```

For higher dimensional arrays, pointers of the type `***double` `****double` are used to allocate the necessary memory.

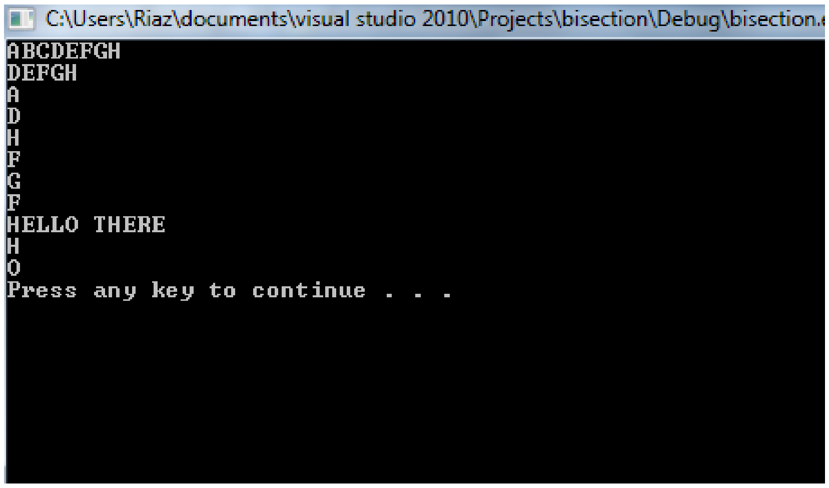
5.7 Pointer Operations

Here we look at manipulating arrays, which becomes beautifully simplified when using pointers.
We begin by considering a string:

```
#include <iostream>


using namespace std;


int main()
{
       char st[]="ABCDEFGH";


       char* p=&st[0]; // p pts to st[0] A
       char* q=&st[3]; // p pts to st[3] D
       cout<<p<<endl; // prints out array from [0] onwards
       cout<<q<<endl; //prints out from [3] onwards


       cout<<*p<<endl; // deferences st[0]
       cout<<*q<<endl; //deferences st[3]


       q+=4; // pts to [7] H
       cout<<*q<<endl;
       q-=2; // pts to [5] F
       cout<<*q<<endl;
       q++;
       cout<< *q<<endl; //pts to [6] G
       --q;
       cout<< *q<<endl; //pts to [5] F


       p+=3; // pts to [3] D



       char* chptr= new char; // create new ptr to a char
```

```
        chptr= "HELLO THERE";
        cout<<chptr<<endl; // gives complete array
        cout<< *chptr<<endl; // gives [0] H
        cout <<*(chptr+4)<<endl; // gives [4] O
        system("pause");
        return 0;
}
```

The output is



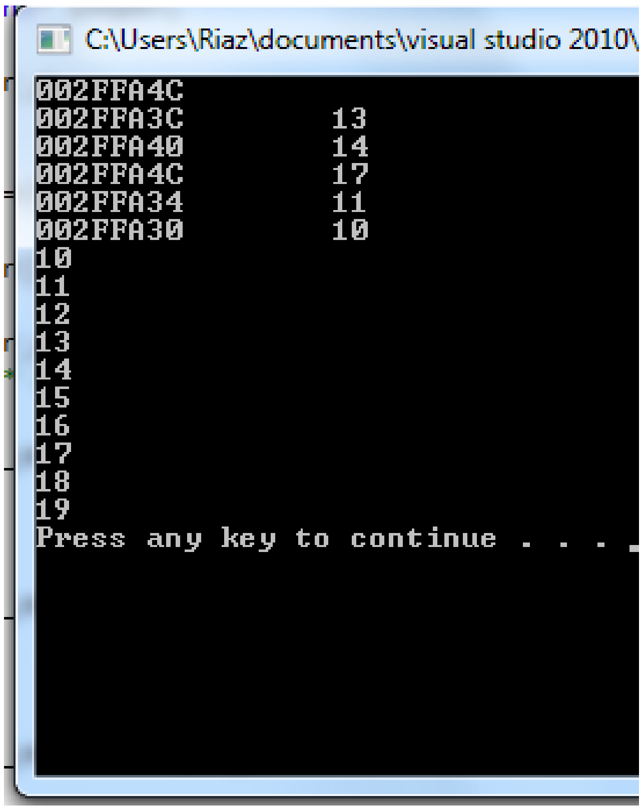
C:\Users\Riaz\documents\visual studio 2010\Projects\bisection\Debug\bisection.

```
ABCDEFGH
DEFGH
A
D
H
F
G
F
HELLO THERE
H
O
Press any key to continue . . .
```

We now illustrate pointer arithmetic for integer arrays:

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

int main()
{

        typedef int* intptr;

        intptr ptr_a;


        int arr[]={10,11,12,13,14,15,16,17,18,19};

        ptr_a=&arr[3];

        cout<<(ptr_a+4)<<endl; // refers to address of arr[3+4]
        // using * above then gives contents of arr[7]=17

        cout<<ptr_a<<'\t'<<*ptr_a<<endl; // gives address and arr[3]

        ++ptr_a;
        cout<<ptr_a<<'\t'<<*ptr_a<<endl; // gives address and arr[4]

        ptr_a+=3;

        cout<<ptr_a<<'\t'<<*ptr_a<<endl; // gives address and arr[7]

        ptr_a-=6;

        cout<<ptr_a<<'\t'<<*ptr_a<<endl; // gives address and arr[1]

        --ptr_a;

        cout<<ptr_a<<'\t'<<*ptr_a<<endl; // gives address and arr[0]

        for(int i=0; i<=9; ++i){
                cout<<ptr_a[i]<<endl;  // prints out entire array.
        }
        system("pause");
        return 0;
}
```

with the following output



```
C:\Users\Riaz\documents\visual studio 2010\
002FFA4C
002FFA3C        13
002FFA40        14
002FFA4C        17
002FFA34        11
002FFA30        10
10
11
12
13
14
15
16
17
18
19
Press any key to continue . . .
```

5.8 Sample Code

The following two programs for Gaussian elimination and the Gauss Seidel method (in turn) illustrate the use of passing a pointer to an array to a function. This is the most effective way to pass an array between functions.

Gaussian elimination

main.cpp

```
#include<iostream>
#include"gauss.h"

using namespace std;


int main()
{


      matrix A = {{10,-1,2,0}, {-1,11,-1,3}, {2,-1,10,-1},
      {0,3,-1,8}};
      vector b={6,25,-11,15};
      vector x={0.0,0.0,0.0,0.0};

      double* ptr = new double;

      ptr=gauss(A, x, b, dimen);


      gauss(ptr);


      delete ptr;
      ptr=NULL;

      return 0;

}
```

gauss.h

```
#ifndef GAUSS_H
#define GAUSS_H

typedef const unsigned short int ushort;
ushort dimen=4;
ushort max_=dimen-1;
typedef double matrix[dimen][dimen] ;
typedef double vector[dimen] ;


double* gauss (matrix , vector, vector, ushort);
void gauss (double*);

#endif
```

gauss.cpp

```
#include<iostream>
#include"gauss.h"

using namespace std;



double* gauss (matrix A, vector x, vector b, ushort dimen)

{
	double* pptr=new double[dimen];

	double pivot,sum;

//	int k,j,l;

	for (int k=0; k<=max_-1; k++)
	{
		for (int l=k+1; l<=max_; l++)
		{

			pivot=A[l][k]/A[k][k];


			for  (int j=k; j<=max_;j++)

			{

				A[l][j]-=A[k][j]*pivot;


			}

			b[l]-=b[k]*pivot;


		}

	}


	x[max_]=b[max_]/A[max_][max_];


	for (int k=max_-1; k>=0; k--)
	{
		sum=b[k];


		for (int j=k+1; j<=max_; j++)
		{

			sum-=x[j]*A[k][j];
		}
```

```
                x[k]=sum/A[k][k];


        }


        for (int k=0; k<=max_; k++)
        {

                pptr[k]=x[k];

        }

        return pptr;

}



void gauss (double* ptr){

for (int i=1; i<=dimen; i++)

        {
                cout<<"x"<<i<<" = "<<ptr[i-1]<<endl;
        }
}
```

Gauss Seidel Method

main.cpp

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


int main()
{
      matrix A = {{10,-1,2,0}, {-1,11,-1,3}, {2,-1,10,-1},
      {0,3,-1,8}};
      double b[4]={6,25,-11,15};

      double X_old[n]={1.0,1.0,1.0,1.0};
      double X[4];


      int Iter=7;



      for (int i=0; i<n; i++){
            X[i]=X_old[i];}

      double* ptr;



    for (int iteration=1; iteration<=Iter; iteration++){

      ptr=Gauss(A,b,X,X_old);



}
 for  (int i=0; i<n; i++)
      {

            cout<<X[i]<<endl;
      }

      delete [] ptr;


      return 0;

}
```

Gauss.h

```
#ifndef GAUSS_H
#define GAUSS_H

const unsigned short int n=4;
typedef double matrix[n][n] ;
double* Gauss(matrix, double b[4], double X[4], double X_old[4]);


#endif
```

Gauss.cpp

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


double* Gauss(matrix A, double b[4], double X[4], double X_old[4])
{

      double* pptr=new double[4];

for (int i=0; i<n; i++)
      {

            double sum1=0.0;
            double sum2=0.0;

            for (int j=0; j<=i-1; j++)

            {

                  sum1+=-A[i][j]*X[j];
            }

            for (j=i+1; j<n; j++)
            {

                  sum2+=-A[i][j]*X_old[j];

            }

X[i]=(sum1+sum2+b[i])/A[i][i];


      X_old[i]=X[i];


      pptr[i]=X[i];
}


return pptr;
      }
```

6. Recursion

6.0 The Basic Idea

We have already seen how, in a well designed C++ program, many function definitions include calls to other functions.

A function is *recursive* (or has a *recursive definition*) if the definition includes a call to itself. If care is not taken, the function may go on calling itself indefinitely.

Recursion is a familiar idea in mathematics and logic. For example, the natural numbers themselves are usually defined recursively. Very roughly speaking, the definition is:

- 0 is a natural number.
- if n is a natural number then S(n) (i.e. n+1) is a natural number, where S is the "successor function".

In this context, the notion of recursion is clearly related to the notion of *mathematical induction*. Notice also that the above definition includes a non-recursive part or *base case* (the statement that 0 is a natural number).

A Simple Example

The following program includes a call to the recursively defined function "`sum(n)`", which sums all integer numbers between zero and n:

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

int sum(int n);

int main(){

       int n;
       cout<<"Enter the integer value of n: "<<endl;
       cin>>n;
       cout<<"The sum is: " <<sum(n)<<endl;

              return 0;
       }

       int sum(int n)
{
              return n+sum(n-1);
}
```

Program 6.1

A typical input/output session is:

```
        Enter the integer value of n: 4
```

There is no output. In fact the function is calling itself "infinitely". Let us look at precisely what is happening, using the above case for $n = 4$.

Call 1: 4+`sum(3)`

Call 2: 4+3+`sum(2)`

Call 3: 4+3+2+`sum(1)`

Call 4: 4+3+2+1+`sum(0)`

The process continues

Call 5: 4+3+2+1+0+`sum(-1)`

and

Call 5: 4+3+2+1+0+`sum(-2)`.

Clearly the function has failed to terminate after $n = 0$. So although there appeared to be no output, the program is actually still executing, i.e. the function `sum` is still calling itself.

Had the function terminated after $n = 0$ the last number to be added (i.e. zero) into the summation then we would have the correct answer to the calculation.

Now the smallest number in the summation is always zero, and no matter where this calculation begins the count down, zero will eventually be reached. This is the point at which we want the function to stop calling itself.

Now consider the revised function `sum`

```
int sum(int n)
{
      if (n==0)
            return 0;
      else
      return n+sum(n-1);
}
```

which produces the (required) output :

```
Enter the integer value of n: 4
The sum is: 10
```

Another familiar mathematical example of a recursive function is the factorial function "!". Its definition is:

- 0! = 1
- for all n > 0, n! = n (n-1)!

Thus, by repeatedly using the definition, we can work out that

6! = 6.5! = 6.5.4! = 6.5.4.3! = 6.5.4.3.2! = 6.5.4.3.2.1! = 6.5.4.3.2.1.0! = 720

Again, notice that the definition of "!" includes both a base case (the definition of 0!) and a recursive part.

6.1 The components of a recursive function

There are two important ingredients for successful recursion:

Base Case: The base case of a recursive function is the value of the parameter(s) for which the function does not perform a recursive call. Without a base case the function will continue calling itself, and fail to terminate. The base case is usually a value of the parameter(s) for which the solution to the problem is trivial.

In the factorial example, the base case is the number zero, i.e. $0!=1$.

There may be more than one base case in a recursive function, but there must be at least one.

Recursive Case: In addition to the base case, a recursive function will have at least one recursive case, which is the value of the input parameter for which the function calls itself to form a part of the answer.

The recursive call must pass a parameter to the function, which represents some form of progress towards the value of one of the base cases. If we consider the earlier function `sum`, the recursive call passes $n-1$ as the actual parameter to the recursive call. This represents progress, because the value passed is one step closer to arriving at the base case, zero. So each time in calling “we go one step down”.

Suppose we did not pass on a smaller value in the recursive call to the function, we would find that (once) again termination would fail.

Now suppose, the function is defined, so that the recursive call makes no progress towards the base case, i.e.

```
int sum(int n)
{
      if (n==0)
            return 0;
      else
      return sum(n);
}
```

The function will only terminate if we pass it the value of the base case as the actual parameter, so call in the main body of `sum(0)`.

So in summary, every recursive definition must have two parts: a basis and a recurrence relation. The two components are often expressed together in a combination formula like this:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{if } n > 0 \end{cases}$$

This simply combines the basis formula

$n! = 1$, if $n = 0$

with the recurrence formula

$n! = n.(n-1)!$, if $n > 0$

into a single expression that completely defines the factorial function.

6.2 Some More Examples

Below are three more examples of recursive functions. We have already seen a function earlier to calculate the factorial of a positive integer.

Here is an alternative, recursive definition:

```
      int factorial(int number)
      {
            if (number < 0)
            {
                  cout << "\nError - negative argument to factorial\n";
                  exit(1);
            }
            else if (number == 0)
                  return 1;
            else
                  return (number * factorial(number - 1));
      }
```

Fragment of Program 6.2

As a second example, here is a function that raises its first argument (of type "`float`") to the power of its second (non-negative integer) argument:

```
  float raised_to_power(float number, int power)
  {
        if (power < 0)
        {
              cout << "\nError - can't raise to a negative power\n";
              exit(1);
        }
        else if (power == 0)
              return (1.0);
        else
              return (number * raised_to_power(number, power - 1));
    }
```

Fragment of Program 6.3

In both cases, care has been taken that a call to the function will not cause an "infinite loop" - i.e. that the arguments to the functions will either cause the program to exit with an error message, or are such that the series of recursive calls will eventually terminate with a base case.

The third example of recursive function sums the first n elements of an integer array "`a[]`".

```
  int sum_of(int a[], int n)

  {
        if (n < 1 || n > MAXIMUM)
        {
              cout << "\nError - can only sum 1 to ";
              cout << MAXIMUM << " elements\n";
              exit(1);
```

```
            }
            else if (n == 1)
                  return a[0];
            else
                  return (a[n-1] + sum_of(a,n-1));
        }
```

Fragment of Program 6.4

The test driver (i.e. the complete program whose sole purpose is to test a function) for Program 4 is:

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

const int MAXIMUM=10;
int   sum_of(int a[], int n);

int   main(){
int   n;
int   a[MAXIMUM]={1,2,3,4,5,6,7,8,9,10};
cout<<"Enter the number of integers to be summed: "<<endl;
cin>>n;
cout<<sum_of(a,n)<<endl;

return 0;

}
```

6.3 Recursion and Iteration

From a purely mechanical point of view, recursion is not absolutely necessary, since any function that can be defined recursively can also be defined iteratively, i.e. defined using "for", "while" and "do...while" loops. So whether a function is defined recursively or iteratively is to some extent a matter of taste.

Here is a recursive function from earlier, re-defined iteratively:

```
int factorial(int number)
{
      int product = 1;

      if (number < 0)
      {
            cout << "\nError - negative argument to factorial\n";
            exit(1);
      }
      else if (number == 0)
            return 1;
      else
      {
            for ( ; number > 0 ; number--)

                         product *= number;


                  return product;

         }

  }
```

Fragment of Program 6.5

It is a matter of debate whether, for a particular function, a recursive definition is clearer than an iterative one. Usually, an iterative definition will include more local variable declarations. Due to extra stack (see Savitch page 762) manipulation, recursive versions of functions often run slower and use more memory than their iterative counterparts. But this is not always the case, and recursion can sometimes make code easier to understand.

6.4 Two Classic Number Theory Examples

The Fibonacci Sequence

A function may include more than one recursive call. The best example of this is the well-known ***Fibonacci sequence***, defined by the recurrence relation

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n > 1 \end{cases}$$

So

$f(0)=0$	$f(1)=1$	$f(2)=f(1)+f(0)$ $=0+1=1$	$f(3)=f(2)+f(1)$ $=1+1=2$	$f(4)=f(3)+f(2)$ $=2+1=3$	$f(5)=f(4)+f(3)$ $=3+2=5$
$f(6)=8$	$f(7)=13$	$f(8)=21$	$f(9)=34$	$f(10)=55$	$f(11)=89$

The sequence does not converge to a limit, and it can be shown that this is a divergent sequence that grows exponentially.

Once given in this nice mathematical form, the sequence easily translates to a recursive function in C++:

```
int fibonacci(int n)
{
      if(n<2) return n;
      else
            return fibonacci(n-1)+fibonacci(n-2);
}
```

The Euclidean Algorithm

A positive integer d is a ***divisor*** of a larger integer n if n is some multiple of d , i.e. $n=k\times d$ for some integer k. We then say d divides n.

For example, 7 divides 812 because $812=116\times 7$.

Now d (as above) is a ***common divisor*** of two larger integers m and n if it divides both these integers. So again 7 is a divisor of both 812 and 924 because $812=116\times 7$ and $924=132\times 7$.

Let d be a positive integer. Then d is said to be the ***greatest common divisor*** or gcd of two larger integers m and n if it is a common divisor of m and n and no other common divisor of these two integers is greater than d. So 28 is the gcd of 812 and 924.

The greatest common divisor of two integers m and n is usually denoted gcd(m, n), hence gcd(812,924)=28. The Greeks discovered a clever algorithm for finding the gcd(m, n) without having to perform the laborious task of computing all the divisors of m and n. The method is called the Euclidean Algorithm.

The recurrence relation is gcd(m, n)= gcd(n-m, n). Consider the following numerical example which uses Euclid's algorithm to compute the gcd when m=528 and n=936. The algorithm says to repeatedly subtract the smaller integer from the larger until the subtraction yields zero. Then the last positive integer in the sequence must be the greatest common divisor.

$936 - 528 = 408 \longrightarrow \quad 528 - 408 = 120 \longrightarrow \quad 408 - 120 = 288 \longrightarrow \quad 288 - 120 = 168 \longrightarrow$
$168 - 120 = 48 \longrightarrow$
$120 - 48 = 72 \longrightarrow \quad 72 - 48 = 24 \longrightarrow \quad 48 - 24 = 24 \longrightarrow \quad 24 - 24 = 0$

Therefore gcd(528, 936) = 24. The recursive function for this is

```
int gcd(int m, int n)
{
      if (m>n) return gcd(n, m);
      if (m==0) return n;
      return gcd(n%m, m);

}
```

The use of modulus/remainder operator `%` is valid because division is equivalent to repeated subtraction

7. Introduction to Object-Oriented (OO) Programming

7.0 Introduction

Object oriented programming (OOP) has replaced languages such as Pascal or C – the so called procedural or structured programming. The essence of structured programming is to reduce a program into smaller parts and then code these elements more or less independently from each other.

Although structured programming has yielded excellent results when applied to moderately complex programs, it fails when a program reaches a certain size. To allow for more complex programs to be written, the new approach of OOP was invented. OOP, while allowing us to use the best ideas from structured programming, encourages the decomposition of a problem into related subgroups, where each subgroup becomes a self-contained object that contains its own instructions and data that relate to the particular object. In this way complexity is reduced, reusability is increased and the programmer can manage larger programs with increasing ease.

There are three strands (or components) of OOP:

- **Encapsulation** - data hiding.
  We have already seen in earlier sections how functions/modules behave like independent mini-programs, not only containing their own sets of instructions, but with their own variables as well. The variables and instructions within a module are hidden and contained—that is, **encapsulated**—which helps to make the module independent of all other modules and therefore reusable. Good programming centres on the principle of reusable software as it greatly saves time and money and enhances reliability. If the `CDF()` function in section 3.3 has been tested before, one can be confident that it will produce correct values for $N(x)$. If another programmer creates a new and improved `CDF()` function, we do not care how it works as long as it returns the result to the correct precision. With procedural programs, you must know the names of the modules to call, and cannot reuse those names for other modules within the same program. If you need a similar but slightly different procedure, you must create a new module with a different name, and use the new name when you call the similar module. Object-Oriented Programming greatly reduces these limitations.

- **Inheritance** - creating new types that inherit properties from existing ones.
  A basic concept of using classes provides a powerful object organisational structure; it is especially useful because classes are reusable. That is, you can extend them—they are extensible. You can create new classes that extend or are descendents of existing classes. The descendent classes can inherit all the attributes of the original (or parent) class, or they can override inappropriate attributes. Employing inheritance can save a lot of work; when you create a class that inherits from another, you only need to create the new features. When an automobile company designs a new car model, it does not build

every component from scratch. The car might include a new feature—for example, some model contained the first air bag—but many of a new car's features are simply modifications of existing features. The manufacturer might create a larger gas tank or a more comfortable seat, but these new features still possess many of the properties of their predecessors from older models. Most features of new car models are not even modified; instead, existing components, such as air filters and windshield wipers, are included on the new model without any changes. Similarly, you can create powerful computer programs more easily if many of their components are used either "as is" or with slight modifications. Inheritance does not enable you to write any programs that you could not write if inheritance did not exist; you could create every part of a program from scratch, but reusing existing classes and interfaces makes your job easier. In geometry, a Cube is a descendent of a Square. A Cube has all of a Square's attributes, plus one additional characteristic: depth. A Cube, however, has a different method of calculating totalArea (or volume) than does a Square. A DisposableDish class has all the characteristics of a Dish, plus some special ones. In business, a PartTimeEmployee contains all the attributes of an Employee, plus more specialized attributes. Because object-oriented programming languages allow inheritance, you can build classes that are extensions of existing classes; you don't have to start fresh each time you want to create a class.

- **Polymorphism** - using one name for many related but technically different functions/purposes (we can think of *polymorphism* to mean "many forms"). Depending on the context, programming modules may need to adapt the way they operate. OO programs use the science of polymorphism to carry out similar operations in a manner suited to the object; such differentiation is never allowed in languages that are non- object oriented. Without polymorphism, separate module or method names for a function that multiplies two numbers and one that multiplies three numbers would have to be given. Without polymorphism, you would have to create separate module names for a method that cleans a Dish object, one that cleans a Car object, and one that cleans a clothing object. Just as a blender can produce juice regardless of whether two fruit items or three vegetables are inserted, using a polymorphic, object-oriented multiplication function call will produce a correct product whether you send it two integers or three floatingpoint numbers.

The basic underlying principle of object-oriented languages is the idea of an *object*. This is the combination into a single unit, of both data and associated functions which are allowed to operate on that data. The existence of objects lies at the very heart of object-oriented programming.
We have been using objects and classes (implicitly) without defining them formally. Examples of classes and objects included

Class	**object**
`ostream`	`cout`
`istream`	`cin`
`ofstream`	`print`
`fstream`	`input`

Using the broadest definition of the word, the world is made up of individual objects. Looking around we are surrounded by objects: tables, chairs, paper, pens. An object is simply a self-contained entity whose existence is independent of others. An obvious consequence is the need to classify objects into groups. For example:

• **Tower**	belongs to the *class* of	**bridges in London**
• **Buckingham Palace**	belongs to the *class* of	**Royal buildings**
• **Margaret Thatcher**	belongs to the *class* of	**ex-Prime Ministers of Britain**

That is, *Tower* is the object belonging to a class called *bridges in London*.
For the last example if the class was ex-Prime Ministers of Britain not to have received an honorary degree from Oxford University then Margaret Thatcher would be the sole object!
A class can be regarded as a group to which objects belong in terms of their properties. For example we can describe the class Human, as having the following properties:

2 legs; 2 arms; omnivorous; warm blooded; gestation period of 9 months; laughs/smiles when happy

This set of properties describes all humans and if it was complete (difficult to achieve for real-life objects), would define what a human is. So a class can be thought of as an abstract description of an object, whereas an object is an actual entity that is a member of a class.

Suppose there is a person called Gerry . Gerry can be regarded as an object that is a member of the class Human, as he has all the properties of the class Human, in addition, he has an identity (i.e. his name).

Now consider the class Derivative with the following objects

- Future
- Forward
- Option

- Swap

Or Class Option with objects

- American
- Bermudan
- Passport (a call option on the profits of a trading account)
- Hawaiian (Asian option which allows for early exercise)
- Cliquet (Option settles periodically and resets the strike at the then spot level)

C++ the Object Based Paradigm
More formally we can consider the following definitions.

Object: An object has **state**, exhibits some well-defined **behaviour**, and has a unique **identity**.

Class: A class describes a **set** of objects that share a common structure, and a common behaviour.

A single object is an **instance** of a class.

Classes are what makes C++ object-oriented – they are used to specify objects.

An object is to a class, what a variable is to a data type. So we can think of the data type `int` as the class and the variable `x` as the object. Recall, integers are generally defined as simply being the class of positive and negative whole/natural numbers.

In the class Human we have mentioned two types of property:

1. Things which describe the object, i.e. warm blooded
2. Things which the object does, e.g. shouts when angry

OO languages allow us to define these two types of property, i.e. description of the object and action, which it performs. In OO terminology these properties and actions are called (in turn)

- ***attributes*** – data only, descriptions of an object
- ***methods* (or *operations* or *member functions*)** – behavioural descriptions of what an object can do (or can be done to it)

Example: Class European_Option

Attributes	**Operations**
Stock Price *S*	Call Price *C*
Exercise Price *E*	Put Price *P*
Interest rate *r*	delta Δ
Volatility σ	gamma Γ
Expiry *T*	theta θ
Cost of carry *q*	

The attributes are simply data which help define an option and the operations are functions, which are appropriate to objects of that class. So in the table above we can think of the attributes as variables/parameters in the paranthesis of $V(S,t,r,\sigma,E,T)$, i.e. the physical structure of an object – in this case a Vanilla Option. Things we can do with any particular object are calculate calls/puts and greeks which are functions/operations.

Once a class has been defined, it can potentially be reused in other programmes by including the class definition in the new program. However, it is not necessary for the programmer who uses a class to know how that works, they should simply need to know how to use it.

We will extend this example to consider code later.

All OO languages, support this sort of use by allowing the implementation of an object to be hidden from the outside. Only functions and data that are specifically declared as being available to external programs are, in fact, available.

Classes in C++

"A class is a user- defined type. A class declaration specifies the representation of objects of the class and the set of operations that can be applied to such objects."

A *class* comprises:

- *data member*s: (or fields) each object of the class has its own copy of the data members (local state)
- *member function*s: applicable to objects of the class

Data members describe the state of the objects. They have a type, and are declared as:

```
type dataMemberId
```

Member functions denote a service that objects offer to their clients. The *interface* to such a service is specified by its return type and formal parameter(s):

```
returnType memberFunctId(formalParams)
```

Member functions are functions that are included in the class.
In particular, a function with **void** return type usually indicates a function that modifies the state of the receiver.

Here is what a class declaration looks like:

```
class Name {
public:
	// public members go here

private:
	// private members go here

};
```

Note the class ends in `;` .

Finance Based Example
Consider the exercise of pricing a European Option *V* (can be a call or a put). Start with a simple example.

When we model this as a class we must identify its attributes and the messages to which instances (objects) of the class respond to. The attributes are as defined previously are r, $\boldsymbol{\sigma}$, E, T (in years), q.

These attributes are just names and when we create instances of the class we must give values to them, for example (Haug 1998):

$\boldsymbol{\sigma} = 0.15$

$K = 490$

$T = 0.25$ (3 months)

$r = 0.08$

$q = 0.03$

We thus see that the object is concrete while its corresponding class is abstract. Having defined the object's data we may speculate on the kinds of information we wish to extract from the object. This will be specific to the group requiring the information, i.e. Traders, Quantitative analysts, Risk managers, Technologists.

Each group has its own requirements and features that they would like to have. For example, a common set of requirements might be:

Calculate the option price

Calculate an option’s sensitivities (for hedging applications) - greeks

The ability to support constant, time-dependent and stochastic volatility models

Export option-related information to a spreadsheet, for example Excel

These features will be implemented by one or more so-called member functions. In order to reduce the scope we concentrate on the pricing and hedging. For example, the price for a palin vanilla in terms of the Black-Scholes price is

```
double CallPrice()
{
	double tmp = sig * sqrt(T);
	double d1 = ( log(S/E) + (q+ (sig*sig)*0.5 ) * T )/ tmp;
	double d2 = d1 - tmp;

	return (S * exp((q-r)*T) * N(d1)) - (E * exp(-r * T)* N(d2));
}
```

and

```
double PutPrice()
{
	double tmp = sig * sqrt(T);

	double d1 = ( log(S/E) + (q+ (sig*sig)*0.5 ) * T )/ tmp;
	double d2 = d1 - tmp;
	return (E * exp(-r * T)* N(-d2)) - (S * exp(qb-r)*T) * N(-d1));
}
```

A Simple Class

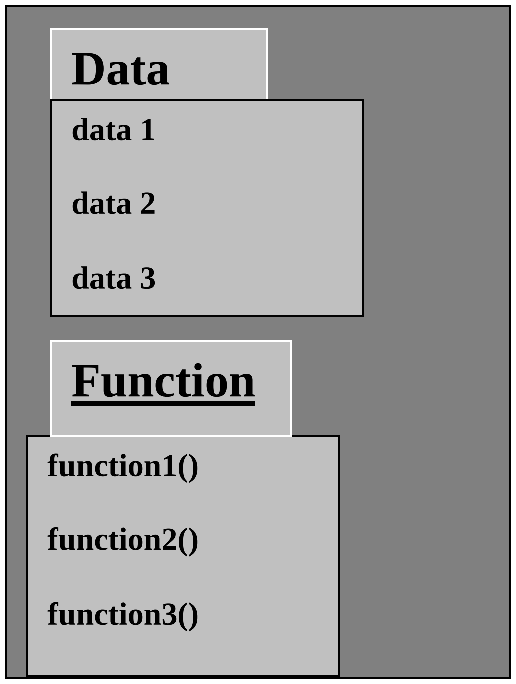
As a first example we consider the following simple program, which consists of a class and two objects of that class. Although basic in nature, the piece of code however demonstrates the rudimentary features of classes in C++.

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


class small_object{

private:
      int _data;    // Class data

public:
      small_object()  // zero argument constructor
      {_data=0;}

      small_object(int data)   // one argument constructor
      {_data=data;}


      ~small_object() {}   // destructor

      void show_data() // member function to display data

      {cout<<"data is "<<_data<<endl;} // 1 lined function
}; // end of class definition


int main(){

      small_object s1=1066;
      small_object s2=1666;
      /* 2 objects created and initialised*/

      s1.show_data();  // next 2 lines call member function to display
data
      s2.show_data();

      return 0;

}
```

The output is

`Data is 1066` $\longleftarrow$ object s1 displayed this

`Data is 1666` $\longleftarrow$ object s2 displayed this



Placing data and functions in this way, into a single entity is the central idea of OO-programming.

private and public

What is the purpose of these two (unfamiliar) keywords? A key feature of OO is ***data hiding***

- Data concealed within class
- Cannot be accidentally accessed by functions outside of class
- Primary mechanism for this being placed in class and making it private
- Private data/functions only accessible from within class
- Public data/functions on the other hand can be accessed from outside the class

Data Hiding not to be confused with security techniques. This is simply to hide data from parts of the program that do not need to access it. Data from one class is hidden from other classes. Data hiding is designed to protect well-intentioned programmers from honest mistakes.

Class Data

- `small_object` class contains one data item: `data` of type `int`
- data items are called *member data* (or *data members*) – any number can be in a class

- the member data: `data` follows keyword `private` – so only accessible inside class

Member Functions

- *Member functions* (or *methods*) are included within class
- `small_object` class contains 2 methods `set_data()` and `show_data()`
- both member functions follow keyword `public`, so can be accessed outside class

Functions – public, Data – private

Generally, data within class is private, and functions made public – to make data safe from accidental manipulation. Functions are public so that they can be accessed from outside the class.

However, there is no rule that enforces this. You may find circumstances where you will need private functions and public data.

Using the Class

Having declared the class, the main body of the program, `main()` is provided to make use of it.

Defining Objects

The first statement in `int main()` is

```
small_object s1, s2;
```

which defines 2 objects belonging to the class. Important to note that the class `small_object` does not create any objects, but merely describes how they will look when created. It is the definition above, which actually creates the objects to be used by the program. Defining an object is similar to declaring a variable type.

Defining objects in this way means creating them and is called *instantiating* them. This term arises because an *instance* of a class is created.

Calling Member Functions

The next two statements in `main()` call the member function `set_data()`:

```
        s1.set_data(1066);

        s2.set_data(1666);
```

Although unlike 'normal' function calls (due to the period), this is the syntax used in OO to call a member function, associated with a specific object. The reason is a member function of a class must always be called in connection with an object of the class – so it would not make sense to have

```
set_data(1066);
```

by itself, because a member function is always called to act on a specific object.

To use a member function, the dot operator connects the object name and the member function. This dot operator is also called the *class member access operator*.

The first call to `set_data()` executes this member function of the s1 object. This function sets the variable `data` in object s1 to the value 1066. Similarly the second call initializes the object s2 with the value 1666.

Finally the two calls to the `show_data()` function will cause s1 and s2 to display their value.

Constructors

In the `small_object` class the member function `set_data` was used to initialize the objects created. It is far better to have a scheme, which does this automatically upon creation, rather than making a separate call to a member function.

Automatic initialization is carried out using a special member function called a *constructor*.

A constructor is a member function that is executed automatically whenever an object is created.

Properties:

- the constructor has the same name as the class of which they are members
- no return type is used for constructors
- usual to have 2 such functions, one zero argument list and the other non-zero

A constructor that takes as its only argument an object of its own class is called *copy constructor*. It is also used whenever a parameter is passed by value and when an object is returned by a function. We do not have to create a special constructor for this; one is already built into all classes.

Destructors

There exists another special member function, which is called automatically, when an object is destroyed.

Such a function is called a *destructor* (the complement of a constructor).

A destructor has the same name as the constructor (i.e. the name of the class) but is preceded by a tilde (~).

Like constructors they have no return type and also take no parameter list.

The most common use of destructors is to de-allocate memory which was initially created by the constructor.

The `class small_object` would now be written as

```
class small_object{

      public:

      small_object()  // zero argument constructor
      {_data=0;}

      small_object(int data)   // one argument constructor

      {_data=data;}



      ~small_object()    // destructor

      {}

      private:

            int _data;    // Class data

};
```

together with

```
int main()

{

      small_object s1=1066;

      return 0;

}
```

<u>**How would we express this in terms of header and source files?**</u>

```
main.cpp

#include<iostream>
#include"small_object.h"
using namespace std;

int main()
{

small_object s1;

s1.show_data(1066);

	return 0;

}
```

small_object.h

```
#ifndef SMALL_OBJ_H


#define SMALL_OBJ_H
class small_object{
	public:

	small_object();


	small_object(int data);


	void show_data(int);

	~small_object();



	private:

		int _data;	// Class data


};

#endif
```

small_object.cpp

```
#include<iostream>


#include"small_object.h"
```

```
using namespace std;


small_object::small_object()
      {_data=0;}

small_object::small_object(int data)
            {_data=data;}

void small_object:: show_data(int data)

      {cout<<"data is "<<data<<endl;}

small_object::~small_object()
      {}
```

`friend` **Functions**

In C++ a friend function of a class is a non-member function that has the privileges of a member function of that class, namely access to the class's private members.

Making a function a `friend` of a class instead of an actual member eliminates the requirement of the implicit argument, making instead all arguments passed to the function explicit. This simplifies the syntax and makes the code more readable.

A function is declared to be a `friend` of a class simply by declaring the function within the class and preceding its declaration with the keyword `friend`.

Overloading Operators

We encountered the term *overloading* in the section on functions and saw how powerful the technique of using the same function name to perform different tasks is. Recall the function **average** which calculated this statistical value for different parameter lists. The compiler in each case knew which definition of the function to call based upon what was passed as arguments.

Up till now, we have been using both unary and binary operators on different data types, i.e. integers and floats. An *operator* is essentially a function – hence it can be overloaded. Consider the *addition operator* +, which is usually used as `c=a+b;` Depending on if the operands are type int or type float, the + operator used in each case is different – the C++ complier knows which one to call.

However it can also be expressed formally as `c=operator+(a,b);`

As a function its name is `operator+`.

Below is a list of what can be overloaded (not all covered in this course):

+	-	*	/	%	^	&	\	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	<<=	>>=	= =	!=
<=	>=	&&	||	++	--	,	->*	->
( )	[ ]	new	delete	new[]	delete[]			

The following operators cannot be overloaded

.	.*	::	?:

In addition the pre-processor symbols `#` and `##` cannot be overloaded. Nor can `sizeof` .

Example: Time Class

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

typedef unsigned short int ushort;


class Time
{


public:

      Time()   // default (zero argument) constructor
      {_hours=0;
       _mins=0;
       _secs=0;
      }


      Time(int hours, int mins, int secs) /* overloaded (3 arg)
constructor */
      {_hours=hours;
       _mins=mins;
       _secs=secs;
      }

      Time::~Time()    // default constructor
      {}


      void Time:: display() // function to display object to screen
      { cout << _hours << ":"<<_mins<<":"<<_secs;
        cout<<endl;
      }


      void Time:: add_time(Time t1, Time t2)
      {
            _secs=(t1._secs+t2._secs)%60;
            _mins=((t1._mins+t2._mins)+(t1._secs+t2._secs)/60)%60;

            _hours=((t1._hours+t2._hours)+
                  (t1._mins+t2._mins+(t1._secs+t2._secs)/60)/60)%24;


      }




      friend Time operator+(Time t1, Time t2)
{

      ushort secs=(t1._secs+t2._secs)%60;
      ushort mins=((t1._mins+t2._mins)+(t1._secs+t2._secs)/60)%60;

      ushort hours=((t1._hours+t2._hours)+
            (t1._mins+t2._mins+(t1._secs+t2._secs)/60)/60)%24;
      Time temp(hours,mins,secs);

      return temp;

}


friend ostream& operator << (std::ostream& cout, Time& t)
{

      cout << t._hours << ":"<<t._mins<<":"<<t._secs;
      return cout;

}


private:
int _hours,_mins,_secs;

};
```

```
      int main()
      {

            Time time1(8,45,44);
            Time time2(14,5,57);

            /* create 2 time objects time1 and time2 and initialise with
            times */

            Time time3(0,0,0); // create object time3
            time1.display();  // time1 calls display function

            cout << endl;

            time3.add_time(time1,time2);
            /* time3 calls add function and is assigned time1+time2 */

            time3.display();
            /*time3 calls display function to output to screen*/

            cout<<time1+time2;
            /* using the overloaded operator + to dircetly add times
            and then the overloaded console output statement to the
screen*/
            cout << endl;



return 0;
      }
```

Arrays of Objects

We have already seen how to declare and manipulate arrays of integer, real and character data types. In session 4 we also used arrays of pointers to construct two-dimensional dynamic arrays. To generalise this idea we consider the declaration of arrays of objects.

Consider the previous program consisting of the time class. To create an array that holds objects of type time we make the following declaration :

```
Time time4; /* note that this object upon creation will be set
automatically to (0:0:0)*/
Time array[4]={time4}; /* here we declare an array of type Time to hold 4
objects and we have initialized it to time3, i.e. (0:0:0)*/
```

Within this same code we can define a pointer to a time object and then, have the pointer point to an array which we created above.

```
Time* ptr; // declare a pointer of Time class
ptr=&array[0]; /* initialize pointer by pointing to base address of array
*/
for (int i=0; i<4; ++i){
cout<<ptr[i]<<endl;
}
/* the loop above now prints out the contents of the array given by ptr
*/
```

We can also create dynamic variables of type Time. Consider the following member function for printing an object that has a pointer as part of the parameter list.

```
Time* Time:: display(Time* ptr) /* function to display object to screen
*/
{
return ptr;
}
```

with the following code in the main body

```
Time* ptr=new Time; /* create a dynamic variable to which ptr points */
Time time1(8,45,44); // instantiate an object
*ptr=time1; // the dynamic variable is initialised
ptr->display(); /* this is how a pointer is connected to a member
```

```
function - in this case it prints the variable */
delete ptr; // delete ptr_time;
ptr=NULL; // return null address
```

We note a new form of syntax `->` used when the pointer acted on the function. This is how a pointer is connected to a member function, and is an example of *syntactic sugar*.
In module 2 we used the `fstream` library to access the class `ofstream` and hence create an object out which allowed us to output results to an Excel file (or .dat and .txt formats). This idea also extends to print out objects in various files, in precisely the same way.

Mathematical Example 1: "Vector" Class

The following program creates a class called Vector, in the mathematical sense. 3D Objects of type Vector are created, which can then be added or multiplied (dot and vector products).

Vector.h

```
#ifndef VECTOR_H
#define VECTOR_H
class Vector
{
friend Vector operator* (Vector, Vector);
friend Vector operator+ (Vector, Vector);
friend Vector operator- (Vector, Vector);
friend std::ostream& operator << (std::ostream&, Vector&);

public:
Vector();
Vector (double,double,double);
~Vector();
double abs (Vector);
void print();
double dot (Vector, Vector);
private:
double _x,_y,_z;
};

#endif
```

Vector.cpp

```
#include<iostream>
#include<cmath.>
#include "Vector.h"
using namespace std;
Vector::Vector() {
_x=0.0;
_y=0.0;
_z=0.0;
}
Vector::Vector(double x, double y, double z) {
_x=x;
_y=y;
_z=z;
}

Vector::~Vector() { }
double Vector::abs(Vector a)
{
double modulus = sqrt (a._x*a._x +a._y*a._y + a._z*a._z);
return modulus;
}
double Vector::dot(Vector a, Vector b)
{
double dot_product = (a._x*b._x +a._y*b._y + a._z*b._z);
return dot_product;
}

void Vector::print()
{
cout<<"("<<_x<<","<<_y<<","<<_z<<")"<<endl;
}
Vector operator+ (Vector a, Vector b)
{
double x = a._x + b._x;
double y = a._y + b._y;
double z = a._z + b._z;
Vector temp(x,y,z);
return temp;
}
Vector operator- (Vector a, Vector b)
{
double x = a._x - b._x;

double y = a._y - b._y;
double z = a._z - b._z;
Vector temp(x,y,z);
return temp;
}
Vector operator* (Vector a, Vector b)
{
double x = a._y*b._z - a._z*b._y;
double y = a._z*b._x - a._x*b._z;
double z = a._x*b._y - a._y*b._x;
Vector temp(x,y,z);
return temp;
}

ostream& operator << (ostream& cout, Vector& a)
{
cout << "(" << a._x <<","<< a._y <<"," << a._z<< ")";
cout <<endl;
return cout;
}
```

main.cpp

```
#include<cmath>
#include "Vector.h"
using namespace std;
int main()
{
```

```
Vector a(2,1,3), b(2,2,2), c(0,0,0);
/* create and initialise 3 objects of class vector*/
      double u=a.dot(a,b);
/* form the dot product of a and b, then assign the value to u*/
c=a+b;
c.print();
cout<<endl;
cout << u;
return 0;
}
```

Copying Objects

Assigning the value of one object to another can be performed in a straightforward manner. There are two ways of achieving this. The first is a simple extension of the non-object oriented method, which we used earlier with simple data types, i.e. assigning one object to another. Consider a class Object and two objects `z1` and `z2`, where we wish to set the value of `z1` to `z2` :

```
Object z1, z2;

z1=z2;
```

The other method makes use of a (built-in) default copy constructor. When instantiating an object, it can be declared at the same time by being assigned the value of another (pre-existing) object. If we continue with code above and create a third object z3, which is assigned the value z1, then the copy constructor allows us to make the following declaration

```
Object z3(z1);
```

Mathematical Example 2: "Complex" Class

We include a class of type `Complex`, which creates objects $z=x+iy$. The four arithmetical operators and insertion/extraction operators are overloaded. In addition functions are developed for complex number manipulation using modulus (length), argument (angle) and complex conjugate (reflection in the x–axis). Although `cout<<` has been overloaded, a function to print a complex number is also included. The header and source files are given. As an exercise create a test driver, i.e. the main body of the program to use this class.

comp.h

```
#ifndef COMP_H
#define COMP_H
class Complex
{

friend Complex operator+(Complex,Complex);
friend Complex operator-(Complex,Complex); // overload + and -
friend Complex operator*(Complex,Complex); // overload multiplication
friend Complex operator/(Complex, Complex); // overload division
friend std::ostream& operator << (std::ostream&,Complex&);
friend std::istream& operator >> (std::istream&,Complex&);
// overload cin and cout
public:
Complex (); //default constructor - 0 args
Complex(double ,double ); // constructor - 2 args
~Complex(); // destructor
void print(); // displays z (although we have overloaded cout<<)
void get_data();


double arg(Complex); // argument z= arctan(y/x)
Complex conjugate(Complex); // x+iy becomes x-iy
Complex add(Complex , Complex ); // add function z1+z2
Complex modulus(Complex); // Pythagoras – calculates length of z
private:
double _re, _im,r;
};
#endif
```

```
comp.cpp
#include<iostream>
#include<cmath>
#include "comp.h"
using namespace std;
Complex::Complex () // zero arg constructor
{_re=0;
_im=0;}
Complex :: Complex(double re,double im) // 2 arg constructor
{_re=re;
_im=im;}
Complex :: ~Complex() // destructor
{}
/*-----------------------------------------------------------------*/


void Complex::print()
{
cout<<"("<<_re<<","<<_im<<")"<<endl;
}
/*-----------------------------------------------------------------*/
double Complex::arg(Complex A)
{
double argument = atan (A._im/A._re);
return argument;
}
/*-----------------------------------------------------------------*/

Complex Complex::conjugate(Complex A)
{
double re= A._re;
double im= -A._im;
Complex conj(re,im);
return conj;
}
/*-----------------------------------------------------------------*/
Complex Complex::add(Complex A, Complex B)
{
// Complex c(10,-1), d(-10,1);
// Complex l=c+d;
double re= A._re+B._re;
double im= A._im+B._im;
Complex temp(re,im);
return temp;
// return l;
}
/*-----------------------------------------------------------------*/
Complex Complex::modulus(Complex A) // modulus of z
{
double re = sqrt(A._re*A._re+A._im*A._im);
double im = 0 ;
Complex temp(re,im);
return temp;
}


/*-----------------------------------------------------------------*/
Complex operator+(Complex A, Complex B)
{
double re= A._re+B._re;
double im= A._im+B._im;
Complex temp(re,im);
return temp; }
/*-----------------------------------------------------------------*/
Complex operator-(Complex A, Complex B)
{
double re= A._re-B._re;


double im= A._im-B._im;
Complex temp(re,im);
return temp; }
/*-----------------------------------------------------------------*/
Complex operator*(Complex A, Complex B)
{
double re= A._re*B._re-A._im*B._im;
double im= A._im*B._re+A._re*B._im;
Complex temp(re,im);
return temp; }
/*-----------------------------------------------------------------*/


Complex operator/(Complex A, Complex B)
```

```
{
double denom = (B._re*B._re+B._im*B._im);
double re= (A._re*B._re+A._im*B._im)/denom;
double im= (A._im*B._re-A._re*B._im)/denom;
Complex temp(re,im);
return temp; }
/*----------------------------------------------------------------*/

ostream& operator << (ostream& cout, Complex& frac)
{if (frac._re!=0.0 && frac._im!=0.0){
cout << frac._re <<"+(" <<frac._im<<"i)";}
else


if (frac._re==0.0 && frac._im!=0.0){
cout <<frac._im<<"i";}
else
if (frac._re!=0.0 && frac._im==0.0)
{ cout<<frac._re;}
else
{ cout<<"0";}
cout <<endl;
return cout;
}
```

Explain the following terms: *pointer*; *dangling pointer*; *null pointer*; *dynamic array*.

What does the following code printout and why?

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

void MyFunction(double& a){
a*=10;
}
int main()
{
double a=1;
MyFunction(a);
cout<<a<<"\n";
system("pause");
return 0;
}
```

Write the output for the following block of code

```
     int* c = new int;
     *c = 10;
     int* p = c;
     *p += 3;
     cout << *c << endl;
     cout<<*p<<endl;
```

What does the following code print to the screen?

```
typedef int* intptr;
intptr ptr_a;
int arr[]={0,1,2,3,4,5,6,7,8,9};
ptr_a=&arr[3];
cout<<*(ptr_a+4)<<endl;
cout<<*ptr_a<<endl;
++ptr_a;
cout<<*ptr_a<<endl;
ptr_a+=3;
cout<<*ptr_a<<endl;
ptr_a-=6;
cout<<*ptr_a<<endl;
--ptr_a;
cout<<*ptr_a<<endl;
```