Introduction to C++

Steve Miller (s.miller@bbk.ac.uk),
Mathematics and Statistics, Birkbeck College.

July 2012

Contents

1 Introduction

In this short course we will first introduce the basics of C++ language syntax, variable types and control flow. Next will come the important object oriented programming concepts of encapsulation and polymorphism. The purpose of these programming techniques is to conveniently package data storage and computational functionality, and to make code easily reusable. Most of the time spent on pure programming language issues will therefore be focussed on classes and their design.

However, programming techniques are only a means to an end: we are interested in applications of numerical methods in finance, and will be spending the remainder of the time on linear algebra methods, and random number generation.

We will also be drawing on code from other sources; although methods used for solving linear equations are interesting, you might not want to write a class to solve this problem before starting on anything else. In particular, there are good libraries available in the public domain for all common linear algebra tasks (*e.g.* Cholesky factorisation, solving linear equations), so we will not be dealing with how to solve these problems in this course. (If you look at some of the code we discuss, you will see why!)

It is important to note that this course cannot be an exhaustive survey of the C++ language or of object oriented programming techniques; the language is a large and complex one! However, we will give sufficient information about the background issues to enable you to get started on the numerical/financial side of things which is after all the focus of this course.

Compilers

Some possible compilers and development environments are:

1. Windows: Visual C++ 2010 Express is a very good compiler with a visually oriented debugger. This can be downloaded for free from Microsoft, along with SQL server.

2. Linux: GNU compiler. Free, but has a liberal attitude to language standards. (Called g++ on Linux.)

Useful Books

There are many books on C++, but not very many good ones. We recommend the following as being about the best available:

1. *C++ Primer*, Lippman and Lajoie, 4th Edition, Addison Wesley, 2005. This is probably about the best beginner/intermediate book on C++ and is fine for self-teaching.

2. *The C++ Programming Language*, Stroustrup, 3rd Edition, Addison Wesley, 1997. Written by the inventor of C++, this book contains everything you will ever want to know about C++ and quite a lot more besides. It is heavy going, though, until you are comfortable with the basics.

3. *C++: The Core Language*, by G. Satir and D. Brown, published by O'Reilly. This is a particularly good book if you already have some C experience.

4. *Introducing C++ for Scientists, Engineers and Mathematicians*, second edition, by D. M. Capper. Published in paperback by Springer. this book is oriented almost entirely towards scientific computing methods and is not about general language issues.

5. *Numerical Recipes in C++: The Art of Scientific Programming*, by W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, published by Cambridge University Press. This book is a source of implementations of many useful numerical methods, but any recommendation has to be strongly qualified. The design of the code in the book could at best be described as average, contains many bugs and in no way should be taken as examples of good programming practice. However, there are still things worth "borrowing."

When you have a good grasp of language basics, and are confident with the principle and practice of using classes, the following book will be of great interest:

Design patterns: elements of reusable object-oriented software by E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Addison Wesley (1995).

Part I
The C++ Language

2 Language Basics

Note that there is far more to the language than is covered in this brief section; this part of the course is just a "getting started" survey of the most important principles.

All C++ programs must contain one and only one function called `main`, and it is best to put this into a file called `main.cpp`. This is the function which is called by the operating system when a binary executable is run. The form of `main()` should be as follows

```
int main()
{
  // code goes in here
  return 0;
}
```

any text after the `//` characters on a line is a comment (*i.e.* is ignored). The `main` function must have an `int` return value, which can be used to indicate exit status from the program; the example shown here returns the value `0`. A semi-colon is required at the end of each line of code. Code in C++ is divided up into so-called blocks; these are deliniated by curly brackets. All of the code in `main()` has to be enclosed in curly brackets.

In addition, you will notice that there are other commands which are generally needed to main `main()` work. Firstly, there are `#include` statements which tell the compiler to use the interfaces of various parts of the language standard library. In addition, the statement `using namespace std;` is required to make the standard library available. It might seem that something here is redundant, but both are necessary for historical reasons (namespaces are a relatively recent addition to the language).

2.1 Variable Types and IO

2.1.1 Numbers

For the examples in this section (and many further on) the statements required before the declaration of the `main()` function should be added as follows:

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

int main()
{
  // ...
  return 0;
}
```

In numerical analysis we are mostly interested in numbers and there are various ways in which they can be represented. The most commonly used are integers (signed and unsigned) and double precision floating point variables (called `double` for short). The following code fragments declare some of these variables and then show how basic input and output is performed in C++.

```
int i = 1, j;
cout << "int variable types " << i << " " << j << endl;
unsigned int m, n = 0;
cout << "unsigned int types " << m << " " << n << endl;
double x, y = 234.432;
cout << "double types " << x << " " << y << endl;
```

An important issue that one can see from running this example is that all variables must have values set (*i.e.* be initialised) before being used, otherwise the values they contain will be unpredictable.

It is also extremely important to note that the computer does arithmetic with finite precision. Integer variables are represented by 4 bytes (so the maximum value of a variable of type `unsigned int` is $2^{32}-1$), and a variable of type `double` is represented using 8 bytes of memory. There is another type of floating point variable called `float` which uses only 4 bytes of memory, which is nowhere near enough for numerical work; `float` should never be used for numerical analysis.

In the previous bits of code, the `cout` represents standard output for the system (the `'c'` stands for "console"), and the `<<` characters send whatever is

on their right to output. This output appears on the command line interface (in the CBX environment this is the window at the bottom). Similarly, `cin` can be used as standard input to get data from the command line interface, as follows:

```
double x;
cout << "Enter a number : ";
cin >> x;
cout << endl << "You typed: " << x << endl;
```

2.1.2 Strings

Another important variable type which is part of the standard library of C++ is the string class. We have been using this already, since anything enclosed in double quotes is a string, but string variables can be declared in the exactly the same way:

```
string oString;
oString = "a value for the string";
cout << oString << endl;
```

The notation in naming the variable (`oString`) should be read as "object of type string." You will need to use the statement

```
#include <string>
```

in order to use the string class.

2.1.3 Boolean variables

A variable of type `bool` can be used to represent the values `true` or `false`, and is generally used in logical decision making (see the section on logical operations below). For example

```
bool bVar01 = false;
bool bVar02 = true;
```

although one can output these, they are only really useful when combined with the logical operations and control structures outlined below.

2.2 Operations On Variables

2.2.1 Arithmetic operations

All of the ususal arithmetic operations apply to the variable types just described. For example,

```
int i1 = 10, i2 = 20;
double x1 = 1.234, x2 = 3.432;

cout << "x1 * x2 = " << x1 * x2 << endl;
cout << "x1 / x2 = " << x1 / x2 << endl;
cout << "i1 - i2 = " << i1 - i2 << endl;
```

Now some operations with variables of type `unsigned int`

```
unsigned int j1 = 1, j2 = 100;
cout << "j1 - j2 = " << j1 - j2 << endl;
```

Does the output here make sense? Here is another example where the reason for the output might not be entirely obvious:

```
unsigned int k1 = 3, k2 = 2;
cout << "k1 / k2 = " << k1 / k2 << endl;
```

We would expect the output here to be a decimal number; however, we have been given the *integer* part of the expected result. This is of course what is supposed to happen, but remember that integer variables can only represent integer values so using them in numerical analysis is to be done with care!

Now suppose that we wanted to evaluate an expression involving variables of type `double` and `int` and get not what the compiler thinks we should get, but the mathematically correct answer instead. There is of course a way to do this:

```
int m1 = 3, m2 = 7;
cout << "m1 / m2 = "
  << static_cast<double>(m1) / static_cast<double>(m2)
  << endl ;
```

what we have done here is to use what is called a *cast*. This is an operation which tells the compiler to take the value of a variable but use it as if it were

actually using another type. In this case, we have got integer variables, but we are telling the compiler to perform the arithmetic as if the values were stored in variables of type `double`.

We can go in the other direction as well, and cast a `double` to type `int`, for example

```
double w = 3.321;
cout << "value = " << static_cast<int>(w) << endl;
```

where the fractional part of the `double` variable has been ignored. Note that casting a variable does not affect the value which that variable stores; it only affects how it is *used*.

The style of casting just described is a recent addition to C++, and in a good deal of older code, you will often see a different syntax:

```
double g = 123.321;
cout << "(int) cast of g = " << (int)g << endl;
```

where the variable type to be cast to is written in round brackets. This approach is still valid, but the newer method described above should always be used.

Important Point: Whenever you see arithmetic operations which incorporate variables of different types, be *very* careful that it means what you think it means.

2.2.2 Operator shorthand

The language contains a couple of shorthand idioms which are commonly used. Firstly,

```
unsigned int iValue = 10;
iValue++;
cout << "iValue = " << iValue << endl;
iValue--;
cout << "iValue = " << iValue << endl;
```

where the value of the variable has been incremented (using `++`) and then decremented (using `--`). That is, these operations are the same as the statements

```
iValue = iValue + 1;
iValue = iValue - 1;
```

These operators can also be put in front of the variable upon which they are to operate; try this and see what the difference is. Secondly, we have a shorthand using other arithmetic operators

```
double dX = 123.456, dY = 10.0;
dX *= dY;
dY += 3.5;
cout << "dX = " << dX << "; dY = " << dY << endl;
```

where the operation `*=` should be read as "multiply and assign", and `+=` should be read as "add and assign", and could be written out in full as

```
dX = dX * dY;
dY = dY + 3.5;
```

This shorthand also works with other operators.

2.2.3 Logical operations

We also have a set of logical operations in order to compare values of variables. For example,

```
double x = 1.0, y = 2.0;
bool b01, b02, b03, b04;
b01 = (x == y);         // test for equality
b02 = (x != y);         // test for inequality
b03 = (x <= y);         // test for x leq y
b04 = (x > y);          // test x gt y
```

The result of one of these logical operations (in this case upon two `double` variables) is a Boolean value; in this case, the value of `b01` will be `false` and the value of `b02` will be `true`. The following examples perform logical operations upon Boolean values as arguments,

```
b03 = (b01 && b02);     // logical AND
b04 = (b02 || b04);     // logical OR
```

and we can of course test Boolean variables for (in)equality as well. The `!` operator is logical NOT, that is changes `true` to `false` and *visa versa*, for example we might write

```
b01 = !b02;
```

and if `b02` is `true` then `b01` will be assigned to `false` in this example.

One final point is that we can also apply the logical AND/OR operations to number variables such as `unsigned int` or `double`. These are interpreted as meaning `true` if they are non-zero, and `false` if they are zero.

2.3 The Mathematics Library

There are many mathematical functions built into the language, which can be accessed in any code provided that the statements

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

are incorporated into the header of the file in question. A few examples are `sqrt()` (square root), `abs()` (absolute value), various trigonometric functions, and many more.

For some reason, the value of π is not built in, but values of many constants can be easily obtained

```
double dPI = acos(-1.0);
double dE  = exp(1.0);
double dR2 = sqrt(2.0);
```

2.4 Control Structures

2.4.1 Looping

The most commonly used looping structure has the following form

```
unsigned int nLoop = 10;
for (unsigned int iLoop=0; iLoop<nLoop; iLoop++)
{
  // code to be executed in the loop
}
```

the `for` keyword is followed by three expressions in round brackets seperated by semi-colons. The first of these initialises the loop, the second must evaluate to `true` for an iteration of the loop to take place, and the third is

executed at the end of every iteration. Each iteration of the loop executes the code in the block immediately following the for statement. If brackets are not used to deliniate a block of code, then only the next line of code will be executed at each iteration.

In this example, the loop is initialised by setting `iLoop=0`, and if `iLoop<nLoop` is `true` then an iteration will take place. At the end of each iteration, the expression `iLoop++` is executed (meaning `iLoop=iLoop+1`), incrementing the value of the looping variable.

In general the `for` statement has the structure

```
for ( <exp1>  ;  <exp2>  ;  <exp3> )  {    }
```

consisting of three expressions separated by semi-colons. The first initialises the loop, the second has to evaluate to `true` in order for an iteration to be executed, and the third is executed at the *end* of each iteration. Note that the test of `<exp2>` is made *before* an iteration, therefore the looping code may never be executed.

One can exit from any loop using the `break` keyword, for example

```
if (iLoop >= 2)
  break;
```

could be used in the first example using the `for` loop given above.

There are other looping structures which are less commonly used;

```
while ( <exp> ) {    }

do {      }
while ( <exp> );
```

in both cases, the code enclosed in curly brackets is executed if the `<exp>` evaluates to `true`. However, the difference between the two approaches is apparent if one considers when the expression `<exp>` is tested. In the first case, the expression is tested at the beginning of the loop, and so the code may never be executed. In the second case, the expression is tested at the *end* of the loop, so the looping code will always be executed *at least once*.

2.4.2 Decision making using `if`

The `if` statement is the most commonly used decision making mechanism, and has the following form

```
if ( <exp>  )
{
  // code executed if <exp> is true
}
```

where if the `<exp>` enclosed in round brackets evaluates to `true` then the following block of code is executed.

Sets of decisions can be put together, and we might also use Boolean data types as well, for example

```
bool bVal;
double x = 123.123, y = 345.432;
bVal = (x >= y);

if (bVal)
{
  // perform task A
}
else
{
  // perform task B
}
```

or using `else if` we can have more than two possibilities, as follows:

```
if (x <= 2.0)      {      }
else if (x <= 4.0) {      }
else               {      }
```

2.4.3 Decision making using `switch`

This comes under "possible, but not recommended."

The `switch` statement can be used to select one of several tasks from a list of possibilities which are selected according to the value of some specified variable

```
unsigned int iTestValue;
// do something with iTestvalue here
switch (iTestValue)
{
```

```
case 1:
  cout << "Task 1 here" << endl;
  break;
case 2:
  cout << "Task 2 here" << endl;
  break;
case 3:
  cout << "Task 3 here" << endl;
  break;
default:
  cout << "Default task here" << endl;
  break;
}
```

The name of the variable upon which to switch is specified in round brackets (and it must be of integer type), and the special values specified by the programmer are given in each of the `case` statements, all of which must be enclosed in curly brackets to make a block of code as usual. Note that each of the `case` statements is followed by a colon, not a semi-colon.

The `break` statements cause the program to exit from the `switch` block of code when a particular case has completed, but do not have to be present. However, if the `break` statements are not used, then every bit of code from the first case which is found to be true until the end of the `switch` will be executed. Consider the following example

```
unsigned int iP = 1;
switch (iP)
{
  case 1:
    cout << "task 1" << endl;
  case 2:
    cout << "task 2" << endl;
}
```

Finally, note that the `default` case does exactly what it says; if a `break` is not encountered beforehand, then the default task will be performed, irrespective of the value of the `switch` variable. It does not have to be included, but is usually a good idea to have some kind of default behaviour for cases which are not specified.

2.5 Pointers and Memory

From the point of view of the programmer, the machine's memory (RAM) is a linear list of bytes, each of which is numbered, starting at zero and going up to some maximum value

$$0, 1, 2, 3, \ldots, n-2, n-1,$$

The number referring to a byte is called its address. On a 32-bit machine, the variable type representing memory addresses is a 32-bit `unsigned int` and has a maximum value of $2^{32} - 1 = 4294967295$. Thus the maximum amount of memory you can physically have on a 32-bit machine is about 4Gb. (On a 64-bit machine, the theoretical maximum would be $\approx 1.8 \times 10^{19}$, not far off the number of atoms in the computer.)

You can access what's at byte m by using a variable name, or what is called a "pointer."

A pointer is a special kind of variable which contains the value of the memory address of another variable, in units of bytes. The type of the variable to which the pointer refers also has to be specified:

```
double x = 1.0;   // variable of type double
double *pD;       // pointer to double
```

the `&` operator means "take the address of", and the operator `*` means "value at a pointer address." For example the following code sets the pointer equal to the address of the variable, and then gives some output:

```
pD = &x;
cout << "value of pD = " << pD << endl;
cout << "value at pD = " << *pD << endl;
```

There are many uses for pointers, but we are mainly concerned with memory allocation and management. Suppose that instead of creating a variable of type `double` we wish to have an array of variables (which can be thought of as a vector). The standard way of doing this in C++ is as follows:

```
double *pA = NULL;           // initialise a pointer to NULL
unsigned int nArray = 100;   // length of the array
pA = new double[nArray];     // allocate memory to pointer
  // do something
  // with the array
delete [] pA;                // delete the memory
pA = NULL;                   // reset the pointer to NULL
```

The `new` operator allocates memory and returns a pointer to the location used (remember that a pointer is a memory address). When `new` is called with a number in square brackets `[nArray]` as its argument, it allocates that number of variables and returns the pointer address of the *first* of these. These variables are located in a contiguous piece of memory and, and with the array allocated in the previous example may be used as follows:

```
for (unsigned int iLoop=0; iLoop<nArray; iLoop++)
{
  pA[iLoop] = 0.0;    // set some values
  cout << iLoop << " " << pA[iLoop] << endl;
}
```

The operators `new` and `delete` can also be called without the square brackets in which case only one variable of the specified type will be created. We will be using this approach later on.

Although not required, it is good programming practice to initialise all pointer values to `NULL` (which means 0) and to always test pointer values (to make sure they are not equal to `NULL`) before they are used.

Very Important Point (A): When memory is allocated using the `new` operator, it must always be released from the program again using `delete`. If this is not done then repeated calls to `new` will allocate the computer's memory to the program without it being released, resulting in what is called a memory leak.

Very Important Point (B): If the memory was allocated using `new []` (to allocate an array) then it must be released using `delete []` (to release an array); similarly, if `new` was called without the square brackets argument (to allocate a single variable of the specified type), so too must `delete`. Calls to `new` and `delete` on a particular pointer must match.

2.5.1 Arrays and pointer arithmetic

Suppose that we have an array created using a pointer and the `new` operator in the way that was just described. The elements of this array were referred to by using the square brackets operator, and for an array that consists of `nArray` elements, the index should go from `0` to `nArray-1`. This seems like a peculiar way to index array elements, but these is a very good reason for it.

Consider the creation of the array in the way just described:

```
unsigned int nArray = 100;
```

```
double *pA = new double [nArray];
```

The pointer is of course a memory address, and after the call to the `new` operator, it contains the address of the *first* element of the array. Therefore, this element can be accessed by the statement

```
*pA = 10.0;          // set the first element of the array
```

How can we now make the pointer refer to the next element of the array? The answer is to increment the pointer

```
pA++;                // increment the pointer
*pA = 20.0;          // set the second element
```

So the operation `pA++` is the same as saying `pA = pA + 1`, but if the actual pointer values were to be printed out, then we would see something interesting. Try the following, using the `double` pointer as defined above

```
cout << "pointer value before " << pA << endl;
pA++;
cout << "pointer value after  " << pA << endl;
```

and the two numbers which are output here should differ by 8. The reason for this is that in incrementing the pointer we have stepped from one array element to the next; these array elements are variables of type `double`, each of which occupies 8 bytes of memory; thus in stepping from one element to the next, the memory address will have increased by 8.

Note that although the pointer value is given in bytes, when we increment or decrement a pointer, the units which are used correspond to the *size* of the variables to which the pointer refers. This is why the type of the pointer is important, and why it has to be specified when the pointer is declared.

We can also increment the pointer value by more than unity, and write something like

```
pA = pA + n;       // n is some integer variable
```

and again the number by which we increment the pointer will be interpreted as the number of variables of the specified type to step over, not a number of bytes. Recalling that the value which is stored at a given memory address can be read by using the `*` operator, we can use the statements

```
double x;
x = *pA;
x = *(pA+1);
```

where the first of these means "read the value stored at `pA`." The second of these means "read the value stored at `pA+1`" where this is the next variable of type `double` stored in memory (8 bytes along from the first).

Similarly, we can access any double variable by referring to its location with respect to `pA` in the following way

```
x = *(pA+n);
```

In the code examples above, the `n`-th array element can be accessed using the square brackets operator where we type `pA[n]` where the variable `n` traverses the range `0` to `nArray-1`. The meaning of the array index `n` and the reason why it has this range will now become clear.

The index `n` is the *offset* from the beginning of the array, and so the first element of the array will be found at `pA[0]`, the next at `pA[1]` and so on. The equivalent statements in terms of the `*` operator are

```
pA[0]    same as     *pA or *(pA+0)       first array element
pA[1]    same as     *(pA+1)              second array element
```

and in general then, for the `n`-th array element,

```
pA[n]    same as     *(pA+n)
```

where the index `n` goes from `0` to `nArray-1`. The statements `*(pA+n)` and `pA[n]` are referring to the same array element.

It also makes sense that we can decrement a pointer, and the meaning is the obvious one; for example

```
unsigned int iArray  = 10;
unsigned int *pI = new unsigned int [iArray];
pI = pI + iArray;
pI--;
// now use *pI
```

Furthermore, we can add or subtract pointers remembering at all times that the arithmetic is done in units of the size of the variables to which the pointers refer.

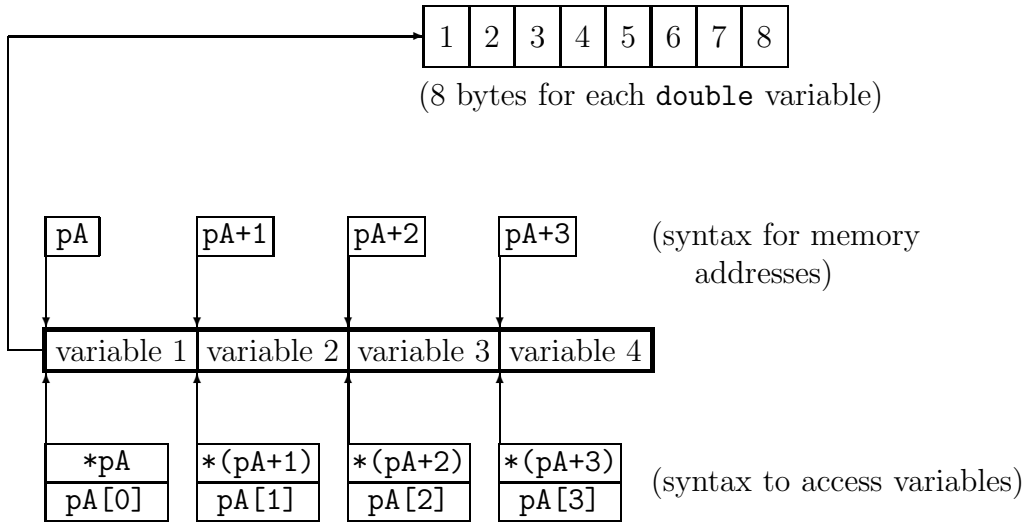

Figure 1: An array of `double` variables, created by the statement "`double *pA = new double [4];`". Each of the variables is stored in 8 bytes of memory (illustrated for the first element of the array only). The memory address of the first byte of storage associated with each of the variables are `pA+n` where the index `n` runs through $[0, 1, 2, 3]$. To read (or set) the values we use the syntax `*(pA+n)` or `pA[n]`.

2.5.2 More on arrays

(A) You can define a 2-dimensional array if you like, but this involves some extra fiddling. An easy way to manage a 2-d array using a 1-d array is by storing the 2-d array in rows one after the other. For example,

```
unsigned int iRows = 10, iCols = 20;
double *pA = new double [iRows*iCols];
```

and just indexing along the 1-d array. Thus the (i, j)-th element of the matrix is obtained by writing

```
x = pA[i*iCols + j];
```

and we might loop over the whole array in the following fashion

```
for (unsigned int i=0; i<iRows; i++)
  for (unsigned int j=0; j<iCols; j++)
    pA[i*iCols + j] = 0.0;
```

(B) A genuine 2-d array can be allocated and used as follows

```
double **pArray = new double* [iRows];
for (unsigned int i=0; i<iRows; i++)
   pArray[i] = new double [iCols];
```

and element access is performed in the intuitive way

```
for (unsigned int i=0; i<iRows; i++)
  for (unsigned int j=0; j<iCols; j++)
    pArray[i][j] = 0.0;
```

But you have to make sure that it's all deleted again in the correct order

```
for (unsigned int i=0; i<iRows; i++)
  delete [] pArray[i];
delete [] pArray;
```

I prefer the first technique because it is easier to follow and involves less typing, both of which are assets in this game.

2.6 Functions

A function is a way of breaking up bits of a program into smaller more convenient parts, or packaging some task which must be performed repeatedly to avoid duplicating material in source code. Here is a simple example

```
double TestFunction01(double x, double y)
{
  double z = sqrt(x*x + y*y);
  return z;
}
```

The function *header* consists of the type of the return value (`double` in this case), the function name (`TestFunction01`), and the list of arguments of the function enclosed in round brackets (two `double` variables). The function *body* is enclosed in curly brackets (like any other block of code), and contains the code to be executed when the function is called. The return value of the function is just the norm of a vector in two dimensions. Here is a code fragment showing how to call the function

```
double a = 1.0, b = 1.23, c;
c = TestFunction01(a, b);
```

So return values and arguments of functions must be either variable types supported by the language or predefined class types. One thing that it is very important to understand about function calls is demonstrated by the following function which appears to change the values of its arguments

```
void TestFunction02(double a, double b)
{
  a = 2.0 * a;
  b = 2.0 * b;
}
```

and if we were to call this function from `main()` as follows,

```
double x = 1.0, y = 2.0;
TestFunction02(x, y);
cout << "x = " << x
     << "; y = " << y << endl;
```

the output might not be quite what one expects. The reason for this is simple: when a function is called, new variables are created which are copies of the argument list. It is these temporary copies which are used during the function call, and only exist for the duration of the function call. Thus any changes which are made to the arguments of a function will not be noticed from the point of view of the calling program.

Therefore in the example using `TestFunction02`, even though the function alters the values of the arguments, when we output the values of `x` and `y` after the function call, there is no change. As a matter of nomenclature, this process is referred to as "calling by value," that is, the *values* of arguments are copied.

What if we want a function which *will* alter the values of its arguments in such a way that the values will be changed in the calling program? Even if we didn't want to change the argument, suppose that it was an object containing a 50Mb image, or six months' worth of tick data; we wouldn't want to have temporary copies created since this would be inefficient in memory usage (allocating memory and creating copies of objects can also slow things down dramatically).

Important Point: By default, function calls in C++ are call-by-value.

2.6.1 Pointers and function calls

The first way of dealing with this is to use a pointer to a variable instead of the variable itself as an argument. A modifed version of the previous example would look like

```
void TestFunction02_Mod01(double *a, double *b)
{
  *a = 2.0 * (*a);
  *b = 2.0 * (*b);
}
```

When this version of the function is called, we have to give the addresses of the variables we wish to alter as follows

```
TestFunction02_Mod01(&x, &y);
cout << "x = " << x
     << "; y = " << y << endl;
```

2.6.2 Call by reference

C++ provides a better way of doing the same thing, which is to call by reference. This is almost the same as using pointers, but the syntax is easier, and our next version of the function will be

```
void TestFunction02_Mod02(double &x, double &y)
{
  x = 2.0 * x;
  y = 2.0 * y;
}
```

and when this version of the function is called, we do not have to use the `&` (address of) operator, since it is done automatically

```
TestFunction02_Mod02(x, y);
cout << "x = " << x
     << "; y = " << y << endl;
```

Important Point: If you want to change the argument of a function, use call by reference.

2.6.3 The `const` keyword

In the previous section, we described how calling by reference (or using a pointer) allows us to change the argument of a function. The second advantage of this is that temporary copies of arguments to the function call will not be created, so duplication of large amounts of memory can be avoided. However, suppose we have an argument to a function which uses a lot of memory (thus we wish to avoid copying it) but which the function call must not change? There is a way of ensuring that a function call cannot change an argument which is given as a reference.

Suppose we have function which requires a variable of type `double` as argument; consider the following two definitions of the function prototype (interface):

```
void Test(double dValue)
{
  // function body
}
```

```
void Test(const double &dValue)
{
  // function body
}
```

Superficially they do the same thing. They each accept a double argument and cannot change that argument with respect to the calling program. However, when the one of the left is called, the argument will be copied, an 8-byte `double` variable. When the version on the right is called, the argument of the function is actually a memory address, which on a 32-bit computer will be 4-byte number (probably an `unsigned int`). So although these two versions of the function prototype appear to do the same thing, the version on the right is to be preferred, because it is more efficient.

Thus the `const` keyword should be used when we have an argument to a function which is large (and we wish to avoid temporary copies), and we do not want it to be changed by the function call. This ensures that the argument *cannot* be changed, even accidentally.

3 Classes

3.1 First Example

One of the most important concepts in modern programming is that of encapsulation, which means taking data which we are processing or performing computations with, and insulating it where possible from the rest of the pro-

gramme. We wish to prevent anything being done with the data which we do not explicitly allow.

The way in which this is done in C++ is using a class, which is a sort of container into which we can place anything. The class also can contain methods (another name for functions) which allow us to access the class data members and perform operations upon them. A class is used in a program in much the same way as we use other simple data types such as `double`, but since we can use class methods as well, they are much more versatile.

```
class Test01
{
private:
  double dVal;                          // a class data member
public:
  void SetVal(double x)                 // a class method
  {
    dVal = x;
  };
  double GetVal()                       // another class method
  {
    return dVal;
  };
  void PerformOperation()               // another class method
  {
    dVal = sqrt(dVal);
  };
};
```

and here are some code examples of how to use this class, which should be put into `main()`,

```
double x = 1.5, y = 2.5;
// create an instance of Test01; call class methods
Test01 oTest01_01;
oTest01_01.SetVal(x);
oTest01_01.PerformOperation();

// now create a second instance of Test01; call class methods
Test01 oTest01_02;
oTest01_02.SetVal(y);
oTest01_02.PerformOperation();
```

```
// finally, let's look at some output
cout << "First object value = " << oTest01_01.GetVal() << endl
  << "Second object value = " << oTest01_02.GetVal() << endl;
```

So what is going on? The line of code `Test01 oTest01_01;` creates an instance of the class `Test01`.

The next two lines of code call the class methods for the object `oTest01_01`. The operator `.` (*i.e.* the dot) indicates membership of an object, thus when we say `oTest01_01.SetVal()` we are calling the method `SetVal()` for the object `oTest01_01`.

Next, we create another instance of `Test01`, called `oTest01_02`, and then call a couple of the class methods. The final line of code outputs the values that are stored by each object.

Each instance of the class has its own copy of the data member `dVal`. The keywords `private` and `public` determine how data members and methods can be accessed from outside of the class object. If the data member `dVal` were in the `public` storage class, then we would be able to access it using code like

```
oTest01_01.dVal = 12.34;
```

instead of calling the methods `SetVal()` and `GetVal()`. Although this seems simpler, it would violate the idea of encapsulation, whereby the class interface controls access to the class data members. In a simple case like `Test01` this doesn't matter, but for more complex classes it definitely does.

3.2 Nomenclature

There is specific terminology associated with creating variables (or objects). For example, where a variable or object is created by something like

```
Test01 oTest01_01;
```

we say that an "instance" of type `Test01` has been created, and the object creation process is referred to as "instantiation." The object `oTest01_01` is referred to as an instance of the class `Test01`, or alternatively an object of type `Test01`.

Note that instantiation can also be performed through pointers, using the `new` operator as discussed in the previous section, and we will be using this approach later on.

3.3 The Constructor and Destructor

Although not mentioned until now, when an object is instantiated, a class method called the constructor is called. This method has the same name as the class itself, but does not have a return value. It could be used for example to set default values for class data members, or perform some operations necessary for initialisation. In the case of `Test01` above we did not explicitly define the constructor, in which case the compiler creates a default implementation.

Similarly, when an object is destroyed, another class method called the destructor is called. This has the same name as the class but prepended with the ~ symbol. In the examples we have looked at so far, none of the objects have been explicitly destroyed by code we have written, but this is happening nonetheless when the end of the `main()` function is reached.

Now we will define a class which creates and stores an arbitrary number of double variables as an array. It will contain data members to deal with memory storage, a default constructor to initialise them, and a second version of the constructor which allocates some memory. We will also need a destructor which will release the memory allocated when the object is destroyed.

```
class Test02
{
private:
  double *pData;                    // class data member
  unsigned int iSize;               // class data member
public:
  Test02()                          // the default constructor
  {
    pData = NULL;
    iSize = 0;
  };
  Test02(unsigned int iS)           // an overloaded constructor
  {
    pData = new double [iS];
    iSize = iS;
  };
  ~Test02()                         // the destructor
  {
    if (iSize != 0)
        delete [] pData;
  };
```

```
  unsigned int GetSize()           // class method
  {
    return iSize;
  };
};
```

Note that in C++ we can have two or more methods with the same name, such as the default and overloaded constructors from this example, and the compiler decides which one you want by the list of arguments. Here is some code to go in `main()` to illustrate the `Test02` class, and some comments as to what is happening.

```
  unsigned int iMemory = 100;

  // create an instance of Test02; call the default constructor
  Test02 oTest02_01;

  // create a couple more; call the overloaded constructor
  Test02 oTest02_02(iMemory), oTest02_03(2*iMemory);

  // get some output
  cout << "oTest02_01.GetSize() = " << oTest02_01.GetSize() << endl
    << "oTest02_02.GetSize() = " << oTest02_02.GetSize() << endl
    << "oTest02_03.GetSize() = " << oTest02_03.GetSize() << endl;
```

We create three instances of `Test02`, one calling the default constructor and two calling the overloaded constructor which takes an `unsigned int` as its single argument. In the output line, each of these instances returns the size of their memory storage, and we can see that the one created using the default constructor of course has size zero. The objects will be destroyed and their destructors called when the end of `main()` is reached.

This class has a couple of limitations, in that if the default constructor is called the memory size is stuck at zero, and in any case we cannot access the memory stored in the objects. But that's not the point of this example, and we will return to these issues shortly.

Now we shall create instances of `Test02` through use of pointers and the `new` operator

```
  // declare a couple of pointers
  Test02 *pT02_01, *pT02_02;
```

```
// create instances of Test02
pT02_01 = new Test02;
pT02_02 = new Test02(iMemory);

// get some output
cout << "pT02_01->GetSize() = " << pT02_01->GetSize() << endl
  << "pT02_02->GetSize() = " << pT02_02->GetSize() << endl;

// now we must destroy the objects
delete pT02_01;
delete pT02_02;
```

The way in which the instances of `Test02` have been created has changed, and we are using the `new` operator explicitly to create objects which are now accessed through the pointers `pT02_01` and `pT02_02`. When objects are created in this way, we must call the `delete` operator when we have finished with them.

Important point: Note the syntax for calling class methods when the object is accessed through a pointer, using the `->` operator. This refers to a member of an object (method *or* data), accessed though a pointer to that object.

3.4 The Assignment Operator and Copy Constructor

Given two objects of the same type, we can assign one to the other. Suppose that we do that with the two instances of a class type `Object`, which contains a pointer data member to which the constructor allocates some memory. Appropriately, when the destructor is called for this class, the `delete` operator is called on this pointer.

```
Object Object01, Object02;  // create two objects
Object01 = Object02;        // assignment
Object01 = Object01;        // self-assignment
```

By default, this assignment (or copying) process will copy exactly bit-for-bit the object on the right hand side to the one on the left. The value of the pointer data member of the class will copied and will now have the same value in both of these instances. See Figure 2 for an illustration of this process.

Suppose that the program reaches the end of `main()`, and the destructors are called for all instances of this class. The `delete` operator will be called when these instances are destroyed. However, remember that the pointer

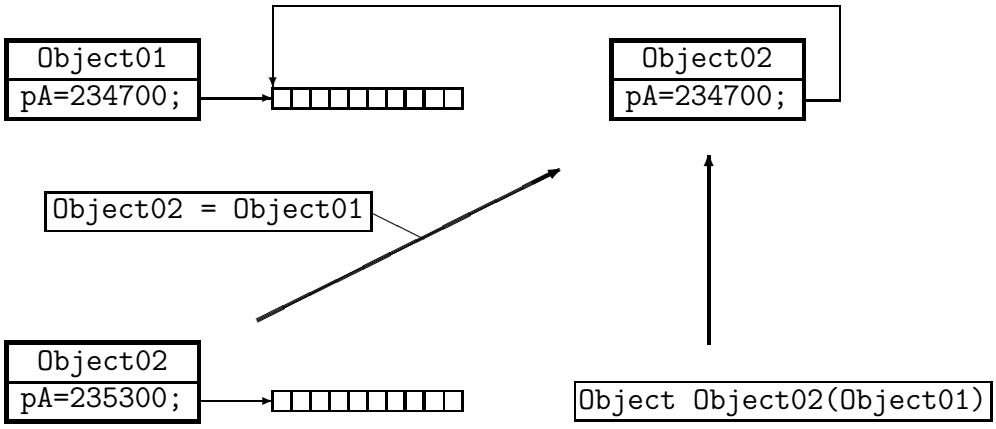

Figure 2: Incorrect assignment and copy construction. See text for explanation.

data member of the two instances now have the same value (that is, point to the same chunk of memory) because of the assignment.

Therefore what will happen is that the `delete` operator will be called twice with the same pointer value as argument. This will result in what the language standards rather euphemistically call "undefined behaviour." Whenever you see this term used, it means something bad will happen.

In addition, we also need to think about what has happened to the memory allocated to `Object02`; is has not disappeared, and remains allocated to the program. However, any reference to it has been lost, and there is now no way it can be accessed; more ominously, it cannot be deleted.

Similarly, when an object is created by copying another instance of the same class (so-called copy construction), the default behaviour is that an exact bitwise copy of the object is made.

```
Object Object01;              // create Object01
Object Object02(Object01);    // copy construction
```

Again, we end up with two objects which contain pointers to the same memory. (This is also illustrated in Figure 2.)

This default behaviour for copying of objects or copy construction is clearly not sensible if pointers and allocation of memory are involved. The next two

sections describe the appropriate solution to this problem in C++.

3.4.1 The assignment operator

When an assignment between two class objects takes place, we are actually calling a class method, represented by the = symbol, which is called the assignment operator. In the above example, the default behaviour for assignment is not appropriate. This problem is solved by creating a new version of the class method to perform the assignment operation.

We now return the the class `Test02`; the appropriate implementation of the assignment operator is as follows:

```
Test02& operator=(const Test02& oRHS)
{
  if (this != &oRHS)
  {
    delete [] pData;

    iSize = oRHS.iSize;
    pData = new double [iSize];
    for (unsigned int iData=0; iData<iSize; iData++)
      pData[iData] = oRHS.pData[iData];
  }
  return *this;
};
```

The first line of the method consists of three parts: the return value which is `Test02&`, that is, a reference to type `Test02`; the name of the operator which has to be preceded by the appropriate keyword, `operator=`; and the argument of the operator, the object on the right hand side. Note the use of a `const` reference here!

The body of the function consists of three parts: firstly, we delete the memory which is currently pointed to by `pData`, the data member of the LHS. We then copy `iSize` from the RHS, allocate new memory to `pData` to hold the number of values given by the new value of `iSize`, and then copy the values over. Finally, the `return *this;` statement returns the object assigned. Every object in C++ has data member called `this` which is a pointer to itself. Figure 3 illustrates the correct assignment proceedure.

Note that the code which does the work in the assignment operator is enclosed inside the following block

```
if (this != &oRHS) {     }
```

which is a test to guard against self-assignment. Suppose that we were to write a line of code which assigns an object to itself

```
oTest02_01 = oTest02_01;
```

which is perfectly OK (although nothing should happen). But what would happen if the guard against self-assignment was not present? Firstly, the memory storage of the object on the left-hand-side would be deleted; then new memory would be allocated for the object on the left-hand-side; then we would attempt to copy the data stored in the object on the right-hand-side... but the data stored in the object on the right-hand-side has already been deleted. So an assignment operator must always test against self-assignment.

Finally, notice that if the sizes of the memory storage on the left and right hand sides of the assignment are the same, then the memory in the object on the left hand side does not have to be deleted and then created again with a different size. The code for assignment can therefore have the form

```
Test02& operator=(const Test02& oRHS)
{
  if (this != &oRHS)
  {
    if (iSize != oRHS.iSize)
    {
      delete [] pData;
      iSize = oRHS.iSize;
      pData = new double [iSize];
    }
    for (unsigned int iData=0; iData<iSize; iData++)
      pData[iData] = oRHS.pData[iData];
  }
  return *this;
};
```

which gives us a slightly more efficient version of the operation.

Important Point: Every object contains a data member `this` which is a pointer to itself.

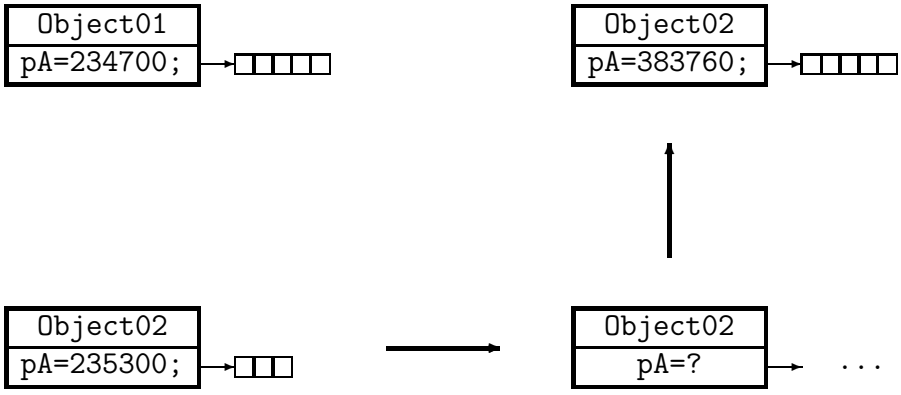

Figure 3: Assignment of two objects (`Object02 = Object01;`). Firstly the storage assocated with the LHS object must be deleted; then storage of the appropriate size is allocated to the LHS; then the data from the RHS are copied to the LHS object.

3.4.2 The copy constructor

Another way of instantiating an object is by copying another one, and there is a special kind of constructor called the *copy constructor* to do this. It is invoked in the following ways:

```
// instantiation using the copy constructor
Test02 oTest02_04(oTest02_01);
Test02 oTest02_05 = oTest02_01;
```

and for the class `Test02` would have the following form

```
Test02(const Test02& oRHS)
{
    iSize = oRHS.iSize;
    pData = new double [iSize];
    for (unsigned int iData=0; iData<iSize; iData++)
        pData[iData] = oRHS.pData[iData];
};
```

Once again, `oRHS` is the object being copied from. We don't need to `delete` anything here, since the object is being constructed, and since the method is a constructor there is no return value.

3.5 Operator Overloading

This refers to the idea of taking one of the language operators and giving it a new meaning for a particular class. This is done by creating a class method which is called through the operator symbol instead of a name. We have already seen one example of this, which is the assignment operator for `Test02`.

There are plenty of others which can be used, for example, `+, -, *, /` which normally have an arithmetic interpretation, or `==` and `!=` which have a logical interpretation. One of the most useful is `[]`, which is commonly used for providing access to data in an array-like object such as `Test02`. Insert the following methods into `Test02`

```
double& operator[](unsigned int iData)
{
  return pData[iData];
};
```

```
double operator[](unsigned int iData) const
{
  return pData[iData];
};
```

The first of these gives data access as a LHS, that is, the element of the array can be assigned (i.e. changed). The second does not allow the data to be changed (hence the keyword `const`), and is called when data are required as a RHS.

Here are are some code examples of data access for `Test02`:

```
iMemory = 10;
// create instance using overloaded constructor
Test02 oTest02(iMemory);

// set some data values
for (unsigned int iData=0; iData<oTest02.GetSize(); iData++)
  oTest02[iData] = sqrt(static_cast<double>(iData));

// output the values
for (unsigned int iData=0; iData<oTest02.GetSize(); iData++)
  cout << oTest02[iData] << " ";
cout << endl;
```

Firstly, we create an instance of `Test02` using the overloaded constructor which contains enough memory to store 10 `double` variables. Then we assign values to the 10 variables by calling the first version of the overloaded `[]` operator which allows the data values to be changed. Then we output the values to the screen by calling the second version of the `[]` operator, where the data values are not changed.

An overloaded operator can be made to mean anything, but it is a good idea to keep the meaning of the overloaded version analogous to the original meaning of the operator if possible.

3.6 Static Data and Methods

So far we have seen class data and methods which are associated with an instance of the class, that is, a particular object. For example, in class `Test02`, each object `oTest02_01` and `oTest02_02` has its own copy of the

data members `pData` and `iSize` which can have different values depending on how their constructors were called.

It is possible to have data members and methods which are associated with the class itself, and are in common to all instances of the class rather than specific to a given instance. These are referred to as `static` data and methods. Static data might be used if a class needs certain numerical constants.

An interesting example of static data and methods might be if we have a very complex program and wish to track the number of instances of a particular class. Suppose we add a static data member to `Test02` to count instances, as follows:

```
class Test02
{
private:
  // ...
  static unsigned int Instances;
  // ...

  public:
  // ...
  static unsigned int GetInstances()
  {
    return Instances;
  }
  // ...
};
unsigned int Test02::Instances = 0;
```

The keyword `static` is used in the class declaration. Note also that the variable has to be declared seperately, since it is not associated with a particular instance of the class. It is initialised with the value zero.

There is also a public method to access the variable `Instances`, which has been called `GetInstances()` and this is also labelled with the `static` keyword.

Now in order to count the number of instances of `Test02` which exist, all we have to do is increase the counter by 1 every time an instance is created and decrease by the counter by 1 every time an instance is destroyed. An instance can be created by the default constructor, any overloaded constructor, or by the copy constructor, so the following line of code

```
Instances++;
```

must be added to each of these. When an instance is destroyed, the destructor is called, and so it must contain the following

```
Instances--;
```

Add these static data and methods and the extra lines of code to your version of `Test02` and check that they do what is expected. The static method to return the counter may be called as follows:

```
cout << "Number of instances of Test02 = "
    << Test02::GetInstances() << endl;
```

Note that access to static methods and data is through the `::` membership operator. In contrast the dot operator `'.'` is used to access instance data and methods, as we have already been using.

3.7 Interface and Implementation

Generally, in large software projects, the code is arranged differently than in the examples we've looked at so far – it wouldn't be a good idea to try to put everything in one file. Code is divided up into two sections, referred to as the interface, and the implementation. The interface is placed in a header file, with the extension `.h` and the implementation goes in a `.cpp` file. Generally one has one pair of files per class and use the same name for the files as the class they contain, so for the `Test02` example we would have `Test02.h` and `Test02.cpp`. In our file main.cpp, instead of the code we've already used, there would be the statement `#include "Test02.h"` and as long as the header and implementation files are added to the project, everything will be fine.

There are a few points of syntax with dividing up the code in this way. The contents of `Test02.h` would be as follows:

```
class Test02
{
private:
  static unsigned int Instances();
  double *pData;
  unsigned int iSize;
public:
  Test02();
```

```
  Test02(unsigned int iS);
  Test02(const Test02& oRHS);
  Test02& operator=(const Test02& oRHS);
  ~Test02();
  unsigned int GetSize();
  double& operator[](unsigned int iData);
  double operator[](unsigned int iData) const;
  static unsigned int GetInstances();
};
unsigned int Test02::Instances = 0;
```

This is the interface of the class, and is the same as before but with the function bodies removed. Note that the assignment operator and the copy constructor are now included. The function bodies will be placed in the implementation file, and the only difference is that the code must state when a method is defined which class it is a member of. The implementation file must contain the statement `#include "Test02.h"`. Class membership is again indicated by the operator `::`, and here is how this would look for the method `GetSize()`

```
unsigned int Test02::GetSize()
{
  return iSize;
}
```

and other than the `Test02::` bit, this looks like a normal function, with the usual return value, name and body. Here is how the copy constructor would look,

```
Test02::Test02(const Test02& oRHS)
{
  Instances++;
  iSize = oRHS.iSize;
  pData = new double [iSize];
  for (unsigned int iData=0; iData<iSize; iData++)
    pData[iData] = oRHS.pData[iData];
}
```

and here is the assignment operator

```
Test02& Test02::operator=(const Test02& oRHS)
```

```
{
  // everything else the same here
}
```

and the rest of the methods would be done in an analagous fashion.

3.7.1 Header Guard

There is an important issue to consider when dividing up projects into interface and implementation, and when many different source code files are involved. What would happen if, for example, we had the header file for a class `Test02.h` and this is then used in two other classes, `ClassAA.h` and `ClassBB.h`, with the usual statement `#include "Test02.h"` in both of these?

Suppose that we now wish to use the latter two classes in `main.c`, and use the statements

```
#include "ClassAA.h"
#include "ClassBB.h"
```

As far as the compiler is concerned, the header for the class `Test02` is appearing twice in `main.c`, which will result in an error. The compiler will not accept any class or variable defined twice, a sensible safety feature.

So how can we stop the interface of a class (or anything else for that matter) appearing twice? The answer is the *header guard*. Compiler directives should be placed around the header of a class, as follows:

```
#ifndef HEADER_Test02_
#define HEADER_Test02_

class Test02
{
  // class interface as usual
}

#endif
```

One should ensure that the symbol which is used is uniquely associated with the interface it protects. For example, in the case of `Test02`, I have used `HEADER_Test02_`. Similarly, we might use `HEADER_ClassAA_`, and this will

work all the time as long as there's no possibility of confusion with other classes or namespaces.

Important point: Whenever you create a class you should place a header guard around its interface, and make sure that it is unique to that class.

4 Templates

4.1 Template Functions

In C++ we can have two or more functions with the same name but which can have different types of arguments or different numbers of arguments, as in the case of the constructors for `Test02`. The types of the return values of the functions (other than constructors) may also differ. Suppose we use the name `TestFunction` as an example, as follows:

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

int TestFunction(int i)
{
  return i*i;
};
```

and here is some code to demonstrate calling these functions,

```
double dVal = 1234.5678;
cout << "TestFunction(dVal) = "
  << TestFunction(dVal) << endl;
int iVal = 10;
cout << "TestFunction(iVal) = "
  << TestFunction(iVal) << endl;
```

Suppose that we want several versions of TestFunction that can accept `int`, `float`, `double`, `unsigned int`, and return the square of the argument as the same type. This can be done without explicitly writing out the code for each one (although this would of course work). We can replace the two versions of `TestFunction` above with the following code,

```
template <typename D>
D TestFunction(D dVal)
{
  return dVal*dVal;
};
```

The keyword `template` requires an argument list enclosed in `<>`. This contains a list of one or more typenames to be used along with the symbol to represent them in the function code. In this case, the symbol `D` is used. Again this is called function overloading. This can be done both for functions which are standalone or class methods.

4.2 Template Classes

The class `Test02` allocates and controls access to some memory, a specified number of double variables. What if we wanted several different versions of the `Test02` class able to allocate memory for different variable types, for example, `unsigned int`, `double`, `float`, *etc*?

In an analogous fashion to our template function from the first example, we can create a template class, the interface of which would look as follows,

```
#include <cstddef>       // this is so we can use
using namespace std;     // the standard library

template <typename D>
class Test03
{
private:
  static unsigned int Instances;
  D* pData;
  unsigned int iSize;
public:
  Test03();
  Test03(unsigned int iS);
  Test03(const Test03<D>& oRHS);
  Test03<D>& operator=(const Test03<D>& oRHS);
  ~Test03();
  unsigned int GetSize();
  D& operator[](unsigned int iData);
  D operator[](unsigned int iData) const;
```

```
  static unsigned int GetInstances();
};
```

Differences between this and `Test02` are few: there is the `template <typename D>` statement which is the same as used for `TestFunction` above; when the class name itself is used as a return type from a function or as an argument type, we have to include the template parameter and specify `Test03<D>` instead.

There are some syntactical differences in the implementation, where the template parameter(s) have to specified, and here is the implementation of the `GetSize()` method which would appear in the file `Test03.cpp`,

```
template <typename D>
unsigned int Test03<D>::GetSize()
{
  return iSize;
}
```

and here is the implementation of the assignment operator

```
template<typename D>
Test03<D>& Test03<D>::operator=(const Test03<D>& oRHS)
{
  if (this != &oRHS)
  {
    delete [] pData;

    iSize = oRHS.iSize;
    pData = new D [iSize];
    for (unsigned int iData=0; iData<iSize; iData++)
      pData[iData] = oRHS.pData[iData];
  }
  return *this;
}
```

When we use a template class in code, we have to specify the template parameters, so that the compiler knows which type is required:

```
// instantiate an object of type Test03<double>
unsigned int iMem = 1000;
Test03<double> oTest03_01(iMem);
```

```
// and now create one of type Test03<unsigned int>
// but access it through a pointer
Test03<unsigned int>* pT03_01 =
  new Test03<unsigned int>(iMem);

// now call one of the class methods
cout << "Size of storage is "
  << pT03_01->GetSize() << endl;

// also note the results from the following
cout << "Test03<double> instances = "
  << Test03<double>::GetInstances() << endl;
cout << "Test03<unsigned int> instances = "
  << Test03<unsigned int>::GetInstances() << endl;

// and make sure that the object is destroyed
delete pT03_01;
```

and then proceed as before.

Use of both template functions and classes can depend on what build environment and compiler one is using. Sometimes the compiler has to be told which template types to generate. In the case of template function example, if we wanted `TestFunction` to work for `unsigned int` and `double`, the implementation file should contain the instructions

```
template double TestFunction<double>(double dVal);
template unsigned int
    TestFunction<unsigned int>(unsigned int dVal);
```

after the function definition. In the case of template classes the implementation file should contain template class instantiation instructions as follows

```
template class Test03<unsigned int>;
template class Test03<double>;
```

in order that the right types are generated. Static data members are also affected by this issue, and the implementation file should also contain

```
unsigned int Test03<double>::Instances = 0;
unsigned int Test03<unsigned int>::Instances = 0;
```

in order that we also get the static data member `Instances` for each of the classes `Test03<double>` and `Test03<unsigned int>`.

Although this might seem like it defeats the object of using templates in the first place, it is necessary because most compiler implementations have not yet caught up with the published C++ ANSI/ISO standard.

Finally, as a exercise, complete the implementation of `Test03`.

5 Inheritance and Polymorphism

In this part of the course we will look at the second main feature of object oriented programming in C++, that of polymorphism. This means that one or more different classes can be accessed through a common interface defined in a *base class*. The so-called *derived classes* are created through the mechanism of inheritance.

5.1 Inheritance

The best way to start is with an example of a simple base class,

```
class Base01
{
protected:
  double dVal;
public:
  Base01()
  {
    dVal = 0.0;
  };
  void SetVal(const double &dValue)
  {
    dVal = dValue;
  };
  double GetVal()
  {
    return dVal;
  };
  virtual double PerformOperation() { return 0.0; };
};
```

Note there are two new things in the declaration of this class. Firstly, the use of the keyword `virtual` in the method `PerformOperation`, and the use of the `protected` storage class, in addition to `public`. (There are no private members of this class so the `private` declaration is not needed.) Here are a couple of derived classes

```
class Derived01: public Base01
{
public:
  double PerformOperation()
  {
    return 2.0 * dVal * sqrt(dVal);
  };
};

class Derived02: public Base01
{
public:
  double PerformOperation()
  {
    return 3.0 * log(dVal);
  };
};
```

In the class declaration, the syntax `class Derived01: public Base01` means that class `Derived01` is publically inherited from the class `Base01`. A class from which others are derived is referred to as a base class. Here is some code to demonstrate basic properties of these classes:

```
// create pointers to type Base
Base01 *pB01, *pB02;

// create objects of type Derived01 and Derived02
Derived01 oD01;
Derived02 oD02;

// set the pointers to object addresses
pB01 = &oD01;
pB02 = &oD02;

// set some data
```

```
pB01->SetVal(10.0);
pB02->SetVal(10.0);

// get some output
cout << "pB01->PerformOperation() = "
    << pB01->PerformOperation() << endl
  << "pB02->PerformOperation() = "
    << pB02->PerformOperation() << endl;
```

Firstly, this example declares two pointers to the base class type, `Base01`. Then one object of each of the two derived types is created by calling the default constructor. Then the first interesting thing happens: each of the base class pointers is set to the address of one of the derived class objects. It is not normally possible to set a pointer of one type to the address of a variable of another type. For example, the following

```
double dVal;
unsigned int* pI = &dVal;
```

would result in a compile time error. It is the relationship between `Base01` and `Derived01` and `Derived02` defined by the inheritance mecahanism which makes it possible in this case.

The kewyword `virtual` is used in the definition of the `PerformOperation()` method in the base class. This means that a derived class can override the version of the method provided in the base class. In this example, even if the `PerformOperation()` method is called through a base class pointer, it is the version of the method provided in the derived class which is executed.

If the keyword `virtual` was not used in the base class declaration, then the base class version of the method would be called through the base class pointer. If the method were called through the derived class interface commands

```
cout << "oD01.PerformOperation() = "
    << oD01.PerformOperation() << endl
  << "oD02.PerformOperation() = "
    << oD02.PerformOperation() << endl;
```

then it is the derived class version of the method would be called. The derived class can, but does not have to, override a virtual base class method; there is a technique for forcing this to happen, which will be discussed later.

One final point is that the data member `dVal` is in the `protected` storage class, rather than the `private` storage class. If `dVal` were `private` in the base class then the derived class methods would not be able to access it; if it is `protected` in the base class then it behaves as if `dVal` were a `private` member of the derived class.

The inheritance relationship means that the derived class acquires data members *and* methods of the base class, but their use and access depends upon the details of the storage class in which they are placed.

5.2 Construction and Destruction of Objects

When creating a derived class object, we are in effect creating an object of the base class type and one of the derived class type all in one, and accessing them through the same interface. This "composite" object can either be accessed through the derived class interface, or the base class pointer, as shown above.

However, this simple example did not deal with either the destructor of the base class or the constructor of the derived class. All of these should be called at some point when these objects are being created, used and then destroyed. The precise order in which these methods are called is important and there are some syntactical quirks involved with making sure that this happens correctly.

Imagine we have the following simple heirarchy of a base class and a single derived class where both the constructors and destructors print out a message when they are called.

```
class Base03
{
public:
  Base03()
  {
    cout << "Base03 ctor" << endl;
  };
  ~Base03()
  {
    cout << "Base03 dtor" << endl;
  };
  virtual void Operation() {};
};
```

```
class Derived03: public Base03
{
public:
  Derived03()
  {
    cout << "Derived03 ctor. " << endl;
  };
  ~Derived03()
  {
    cout << "Derived03 dtor" << endl;
  };
};
```

Now suppose that we construct a derived class object in the following way, and wait for `main()` to destroy it upon exiting

```
int main(void)
{
  Derived03 oD03;
}
```

Try this and see what the output is. This result makes some sense, and all seems to be OK. Note the order in which the base and derived class constructors and destructors are called. Suppose that we now wish to access the derived class properties and methods through the base class interface, as follows:

```
int main(void)
{
  Base03* pB03 = new Derived03;
  // do some things here
  delete pB03;
}
```

try this and note the difference in output with the previous case. There is a problem: the derived class destructor is not being called. The "composite" object is not being properly destroyed, resulting in a memory leak.

When the derived class is created in the first example, all is well, but because in this case the `delete` operator is being called on the base class pointer, only the base class destructor will be called. In order to make sure that destructors for the base class *and* derived class are called, the keyword `virtual` must be added to the declaration of the base class destructor as follows,

```
class Base03
{
  // ...
  virtual ~Base03()
  {
    cout << "Base03 dtor" << endl;
  };
  // ...
};
```

and all will be well.

5.3 Abstract Classes

In previous examples we have considered the case where a virtual method in the base class is overridden by a version of the method defined in the derived class. The derived class does not have to override the default behaviour, but there is a technique for forcing any class derived from a particular base class to provide an implementation of a method.

We add a virtual method to the class `Base03` as follows

```
class Base03
{
public:
  //  ...
  virtual void Operation()
  {
     cout << "Base03::Operation() called. " << endl << flush;
  };
  //  ...
};
```

and so as before it is optional for the derived class to override this method. Now suppose that instead the base class looked like:

```
class Base03
{
public:
  //  ...
  virtual void Operation() = 0;
```

```
  //  ...
};
```

This is called a *pure* virtual method and makes sure that any class derived from `Base03` must provide an implementation of this method, and if it does not, there will be a compile-time error. Furthermore, if the method `Operation()` is to be pure virtual then the base class cannot provide a default implementation. If we attempt to put a function body into this declaration, there will be a compile-time error. Since `Base03` cannot contain an implementation of one of its methods, it is no longer possible to create an instance of `Base03`.

Any class containing pure virtual methods is called *abstract*.

5.4 Some Details

5.4.1 Multiple Inheritance

Is is possible for a class to inherit from more than one base class, thus one might well see a declaration like

```
class Derived: public BaseClass01, public BaseClass02
{
  // ...
};
```

There is no particular problem with this, except if the two base classes have inherited from a common base class themselves, that is

```
class BaseClass00
{
public:
  void BaseClassOperation() {};
};

class BaseClass01: public BaseClass00 {};

class BaseClass02: public BaseClass00 {};
```

Suppose that `BaseClass00` has a method called `BaseClassOperation`, and then we try to call this method for the derived class, for example,

```
oDerived.BaseOperation();
```

there will be a compile-time error. The reason for this is that the class `Derived` has been given two copies of the method `BaseOperation()`, one from each of its base classes, `BaseClass01` and `BaseClass02`, resulting in an ambiguity.

One way to resolve this is to specify which method is desired in the call, that is, which of the base classes one is referring to,

```
oDerived.Base01::BaseClassOperation();
```

A more elegant way round the problem is to make sure it never happens in the first place. If we change the declarations of the classes `BaseClass01` and `BaseClass02` in the following way

```
class BaseClass01: virtual public BaseClass00 {};
```

this ensures that the class `BaseClass00` will only be added to the heirarchy if it not there already, and will therefore only be added once. `BaseClass00` is called a *virtual base class.*

5.4.2 Other Types of Inheritance

There are three storage classes for class data members and methods, `private`, `protected` and `public`. These determine how a class data member or method can be accessed from outside of the class, but also affect how class members are accessed with the inheritance heirarchy.

An example that we have already seen is the difference between `private` and `protected` class members in public inheritance. A `protected` base class member behaves as if it were `private` in the derived class. Although not explicitly stated, it should also be noted that `public` data members and methods in the base class are `public` in the derived class as well.

There are three types of inheritance, `private`, `protected` and `public`. The above examples used `public` inheritance. The difference between these techniques appears in how different storage classes are treated going up and down the inheritance heirarchy. However, `public` inheritance is by the the more commonly used technique, and we do not have time to look at `private` and `protected` inheritance. All of the details can be found in the reference material quoted for this course.

5.5 Practical Use Of Polymorphism

Now for a more realistic example. Suppse we have a class that performs a computation of an expectation via Monte Carlo simulation. In performing this computation, a random number generator is required, but the code which performs the averaging does not need to know how the random numbers are generated. Additionally, depending upon the precise situation being modelled, we might require random numbers drawn from different distributions (e.g. Gaussian, or jump processes).

The first design decision we therefore make is to put the Monte Carlo computation and the random number generator in different classes. The Monte Carlo object will be passed a pointer to a random number generator object which it will use to perform its computation.

In order to make different random number generators interchangeable from the point of view of the object which is using them (the client) they will inherit their interface from a common base class. Furthermore, the base class methods which the client will use will be pure virtual in the base class; this will ensure that every random number generator class must implement them. Remember also that since we are using `public` inheritance, the base class destructor should be virtual.

Here is a sketch of the random number generator abstract class which defines the interface

```
class Random
{
protected:
  // data and methods
public:
  Random() {};
  virtual ~Random() {};

  virtual double GetVariate() = 0;
};
```

The `GetVariate()` method is the one which the client will call to get a draw from the appropriate distribution.

The Monte Carlo class interface might look a bit like this

```
class MonteCarlo
{
```

```
  private:
    // data and methods
    Random* pRandom;
  public:
    MonteCarlo();
    ~MonteCarlo() {};

    bool Register(Random* pR);
    void Unregister();

    bool PerformComputation();
  };
```

The `MonteCarlo` constructor should set `pRandom` to `NULL` to make sure that it isn't used without being set.

```
  MonteCarlo::MonteCarlo()
  {
    pRandom = NULL;
  }
```

The two methods `Register()` and `Unregister()` are used to set (and remove if necessary) a random number generator for the object, and their implementations are

```
  bool MonteCarlo::Register(Random* pR)
  {
    if (pR == NULL)
      return false;
    if (pRandom != NULL)
    {
      Unregister();
    }
    pRandom = pR;
    return true;
 }

  void MonteCarlo::Unregister()
  {
    pRandom = NULL;
  }
```

the `Register()` method returns `true` or `false` depending on whether the operation works or not. The `PerformComputation()` method is the one that does the work, and will be calling the random number generator. Of course this will not be possible if a valid pointer has not already been registered, so

```
bool MonteCarlo::PerformComputation()
{
  if (pRandom == NULL)
    return false;

  // perform computation here, calling
  // pRandom->GetVariate(), etc

  return true;
}
```

but if `pRandom` has been set, then we may proceed.

A class which provides an implementation of an interface (such as `Random`) is referred to as *concrete*. So here is a sketch of a concrete class for a random number generator

```
class Gaussian: public Random
{
public:
  Gaussian() {};
  ~Gaussian() {};

  double GetVariate()
  {
    double dVal;
    // perform an iteration
    // of Box-Muller
    return dVal;
  };
};
```

The `Gaussian` class provides an implementation of the `GetVariate` method which performs an iteration of the Box-Muller algorithm for Gaussian pseudo-random variates, and returns the appropriate value. We can have any number of these concrete classes which provide different distributions.

And finally, this is how these classes might be used,

```
// create random number generator
Random* pR = new Gaussian;

// create MonteCarlo object
MonteCarlo* pMC = new MonteCarlo;

// register pR, and make sure to test
// for possible error
if (!pMC->Register(pR))
{
  cout << "Attempt to register Random failed. "
    << endl;
  // maybe return from here if appropriate
}

// do something
pMC->PerformComputation();

// and finish
pMC->Unregister();
delete pR;
delete pMC;
```

To reiterate the main points thus far:

Encapsulation: The programming tasks have been broken up into more manageable sections. Complex behaviour and numerical computations are more readily tackled (and debugged!) if parts which are mathematically or otherwise independent can be separated from each other.

Polymorphism: The `MonteCarlo` class will accept any object derived from the `Random` abstract class. Since the `GetVariate()` method is pure virtual, any object registered with `MonteCarlo` is certain to provide an implementation of the methods which will be called. Any concrete classes derived from `Random` can be used interchangeably.

6 Exception Handling

Consider the previous example where we create objects of type `Gaussian` and `MonteCarlo` and then test for an error before carrying on with the work. If the error test had shown something to be amiss and the code had returned

at that point, then the objects referred to be `pR` and `pMC` would not have been deleted, leading to a memory leak. The only way to avoid this situation is to make sure that the when the error is noticed, then all other temporary objects are cleaned up before returning the program to a higher level.

The syntax for the basic mechanism for dealing with errors (exceptions) in C++ on the client side is as follows

```
SomeClass oClass;

try
{
  oClass.PerformOperation();
}
catch (Exception oException)
{
  // handle exception
}
```

and in the declaration of `SomeClass`

```
class SomeClass
{
public:
  void PerformOperation() throw(Exception)
  {
    // ...
    if (Error)
    {
       Exception oException;
       throw oException;
    }
    // ...
  };
};
```

Code which might lead to an error is given the ability to "`throw`" an object of some kind, which should be designed to contain data about the state of the object which threw it and what error conditions were encountered.

The call to this perilous operation should be enclosed in a `try/catch` block. The code enclosed in the `try` block is executed as normal if nothing goes wrong, and the code in the `catch` block is ignored. If an exception is thrown

during its execution, however, the code in the `catch` block is then executed. Furthermore, the object which was thrown by the exception is copied so that its contents can be inspected.

Here is a more practical example. We will create a class called `E01` which will perform an operation using polymorphic classes from earlier.

```
class E01
{
public:
  E01()
  {
    cout << "E01 ctor. " << endl;
  };
  ~E01()
  {
    cout << "E01 dtor. " << endl;
  };

  void PerformComputation() throw (string);
};
```

The method `PerformOperation` has been given the ability to `throw` a `string` object if necessary, and here is a first implementation

```
void E01::PerformComputation() throw (string)
{
  bool bError = false;
  Derived03 oD;

  bError = true; // force error condition
  if (bError)
  {
    string oString("error happened");
    throw oString;
  }

  // perform the rest of the task
  oD.Operation();

  return;
};
```

And the client code will look like

```
E01* pE = new E01;
try
{
  pE->PerformComputation();
}
catch (string oString)
{
  cout << "error caught: " << oString << endl;
}
delete pE;
```

Compile and run this example. All is OK here; note that all of the appropriate constructors and destructors are being called.

Suppose now we wish to use the class `Derived03` but in a polymorphic fashion through the base class pointer. Here is a second version of the method

```
void E01::PerformComputation() throw (string)
{
  bool bError = false;
  Base03* pD = new Derived03;

  bError = true;
  if (bError)
  {
    string oString("error happened");
    throw oString;
  }

  // perform the rest of the task
  pD->Operation();

  delete pD; // paired with new Derived03;
  return;
};
```

What happens now? There is a problem because the destructors for `Derived03` and `Base03` are not being called. Looking at the code for this version of `PerformOperation` the reason is fairly obvious: the exception is being thrown *before* the `delete` operator is called on `pD`. One way of dealing with this issue

is as described above; we have to write code explicitly to `delete` all temporary objects before leaving the method. Fortunately, there is a slightly more palatable solution.

There is a template class in C++ called `auto_ptr<>` (pronounced "auto pointer") which behaves as a pointer to another object, but takes care of disposing of the object to which it points when it goes out of scope. We no longer have to call `delete` on `pD`. So here is `PerformOperation` in its final form,

```
void E01::PerformComputation() throw (string)
{
  bool bError = false;
  auto_ptr<Base03> pD(new Derived03);

  bError = true;
  if (bError)
  {
    string oString("error happened");
    throw oString;
  }

  // perform the rest of the task
  pD->Operation();

  return;
};
```

and all is now well.

7 The Standard Template Library

The standard template library (STL) consists of two main parts. Firstly, there is a set of container classes which are designed for managing storage and access of arbitrary data. These data can be any other classes, provided that their design conforms to certain basic criteria. The second part of the STL is a set of algorithms which operate upon the container classes. Some of these perform typical computing operations such as sorting, and a few are oriented towards numerical issues. In order that an algorithm can traverse a container to access data or perform some operation, we also require the important concept of iterators.

7.1 Container classess

7.1.1 The `pair` container

The simplest container class in the STL is the `pair`, which is used to hold two variables (or class objects). It has two template parameters, one for each of the types which it can hold, and is used as follows

```
pair<unsigned int, double> oPair01;
```

which creates an object storing two variables, an `unsigned int` and a `double`. This calls the default construtor for the `pair`. The `pair` has two data members to store each of the variables, and these are (unusually) in the public storage class. They can be accessed as follows

```
oPair01.first = 1;
oPair02.second = 123.234;
```

where the data member `first` accesses the `unsigned int` data member, and `second` accesses the `double` data member. The types are in the same order as the template parameters in the original declaration of the object. We can also assign values when the object is created, by calling an overloaded constructor

```
pair<unsigned int, double> oPair02(2, 234.3);
```

Any data type or class can put into a `pair`, so we could use an example from earlier in the course

```
pair<unsigned int, Test02> oPair03;
```

where the `first` data member is still and `unsigned int` and `second` is now type `Test02`. The methods of the `Test02` member can be accessed through the usual syntax for instance data and methods

```
oPair03.first = 1;
cout << oPair03.second.GetSize() << endl;
```

One use of `pair` objects might be if we want a function or method to return two variables or objects instead of one, for example

```
pair<double, double> PairFunction(double x, double y)
{
  pair<double, double> oResult;
  oResult.first = x+y;
  oResult.second = x*y;
  return oResult;
}
```

and this would be called as follows

```
  double x = 10.0, y = 12.0;
  pair<double, double> oPair04 = PairFunction(x, y);
  cout << oPair04.first << " " << oPair04.second << endl;
```

7.1.2 The `vector` container

We now come to a template class that can store only one type of variable or object, but can deal with an arbitrary number of them. When we use a `vector` the code `#include <vector>` must be inserted in the program header. To create a `vector` we have to supply a single template parameter, for example

```
  vector<double> oVector01;
  vector<Test02> oVector02;
```

These two lines have created `vector` objects, one to store `double` types and the other to store objects of type `Test02`. These two examples have called the default constructor for `vector` and thus have no elements; however, an overloaded constructor could have been called instead, or the `resize()` method could be called as follows

```
  unsigned int iMem = 432;
  oVector01.resize(iMem);

  vector<double> oVector03(iMem);
```

We can also call another overloaded constructor the arguments of which are the size parameter for the vector and an object of the type being stored which will be copied on initialisation,

```
  iMem = 10;
```

```
vector<double> oVector04(iMem, 23.2);

Test02 oTest02(3);
vector<Test02> oVector05(iMem, oTest02);
```

where the object `oVector04` will have all of its `iMem` elements initialised to the value 23.2. The object `oVector05` will store `iMem` `Test02` objects, each of which was initialised by copying the object `oTest02`; the copy constructor for `Test02` was called to do this. If the argument `oTest02` were ommitted here, the default constructor for `Test02` would have been called to initialise the elements of the vector.

The `[]` operator is used to access `vector` elements, and the length of a `vector` object is returned by the `size()` method,

```
for (unsigned int iVec=0; iVec<oVector04.size(); iVec++)
  cout << oVector04[iVec] << " ";
cout << endl;

for (unsigned int iVec=0; iVec<oVector05.size(); iVec++)
  cout << oVector05[iVec].GetSize() << " ";
cout << endl;
```

In the second example here, we have called the `GetSize()` method for all elements of the vector `oVector05`, each of which is a `Test02` object.

When using the `[]` operator to access elements of a vector, it can of course be used as LHS as well as RHS of an expression,

```
oVector04[0] = 4.123;
```

There are many other methods for the `vector` class for inserting elements at arbitrary positions, erasing part or all of the container, and changing its size. However, many of these other operations are not very efficient and are not the best use of the `vector` container.

7.1.3 The `queue` container

A `queue` can again store only one type, and is used to store objects which must be processed in a particular order. The queue is first-come-first-served (i.e. first-in-first-out), that is when we add an object to the container, it goes on the back, and when we remove an object from the container, it comes off the front. In order to use the `queue` class, the program must contain `#include <queue>` at the appropriate point. Consider the following code

```
queue<Test02> oQueue;

// add some objects to the queue
unsigned int nObject = 10;
for (unsigned int iObject=0; iObject<nObject; iObject++)
{
  Test02 oTest02;
  oQueue.push(oTest02);
}
```

The size of the object is given by `oQueue.size()`. Now we can process the objects in the order that they were added

```
while (!oQueue.empty())
{
  Test02 oTest02 = oQueue.front();
  oQueue.pop();

  // processing of oTest02
  cout << oTest02.GetSize() << endl;
}
```

The `pop()` method removes the front element of the queue, but does not return a value; the `front()` method must be called first. If `front()` is called for an object which is empty, then there will be an error; the size of the object must always be tested first. The `push` and `pop()` methods are the only ways in which the `queue` elements can be added or removed. Data access is only through the `front()` and `back()` methods; the `[]` operator is not supported.

7.1.4 The `map` container

The `map` and `multimap` containers are used to store so-called key-value pairs. This means that we can store arbitrary data objects and access them not through an array index variable (like we used for the `vector`) but through some other data type such as a `string`. We must use the directive `#include <map>` to use these classes. For example, if we wish to have string objects in the map which are accessed through numbers as keys, then we might use

```
map<unsigned int, string> oMap01;

map<unsigned int, string>::value_type
```

```
    oValue01(23, "Firstname Lastname 01");
  pair<unsigned int, string>
    oValue02(34, "Firstname Lastname 02");

  oMap01.insert(oValue01);
  oMap01.insert(oValue02);

  cout << "map size = " << oMap01.size() << endl;
```

Firstly, we instantiate a `map` object with template parameters which describe a key type of `unsigned int` and a value type of `string`. Then we create two objects, a `map::value_type` and a `pair`; these are the types which describe key-value pairs for this map. These two forms are equivalent. If we attempt to insert a key-value pair where the key is already present, the attempt will fail.

This is how to search a `map` for the data associate with a particular key

```
  unsigned int iKey = 10;

  map<unsigned int, string>::iterator itMap = oMap01.find(iKey);
  if (itMap == oMap01.end())
    cout << "key " << iKey << " not found " << endl;
  else
    cout << "key " << iKey << " " << itMap->second << endl;
```

the `find()` method for the map returns what is called an iterator; this is basically a pointer to the result (more about iterators in the next section). If the key is not present in the map, then `find()` returns a pointer to the end of the map. We may also remove an element from the `map` which is referred to by the iterator `itMap`

```
  oMap01.erase(itMap);
```

A `multimap` is a map which can have more than one value associated with each key. The basic ideas here are the same as for `map`, but the results of the `find()` method are different. If a key is present, then the results are defined by a *range*; that is, all elements found between two iterators.

7.1.5 The `complex` container

A class for complex arithmetic comes ready made, using `#include <complex>`, and objects may be instantiated as follows

```
complex<double> oComplex01,
                oComplex02(2.3),
                oComplex03(0.2, -3.32);

cout << oComplex01 << endl
     << oComplex02 << endl
     << oComplex03 << endl;
```

we have instantiated three `complex` objects where both the real and imaginary parts are both `double` valued. Other template parameters may be used, but the real and imaginary parts must be the same type. The first instantiation calls the default constructor, and sets the number to zero; the second sets the real part only, and the third sets the real and imaginary parts respectively.

All of the appropriate numerical operators are provided for this class, as well as a few special functions which are meaningful for complex numbers only. For example,

```
cout << norm(oComplex03) << " "
     << abs(oComplex03) << endl;
```

7.1.6 Some other containers

There are more container classes in the standard library, such as `list`, `hash`, `stack`, and others. For example, the `stack` is a last-come-first-served (i.e. last-in-first-out) queue. Where functionality for different containers is analogous, method names have been made as consistent as possible, but inevitably there are many differences between them.

It should be noted that the containers described in previous sections have many more methods and properties than the most important ones which were described. Details of these and the rest of the standard containers can be found the sources previously given.

7.2 Iterators

As mentioned previously, a method of accessing data and traversing the elements of the standard containers is also provided. An iterator for a `vector<double>` can be declared and used as follows

```
vector<double> oVector(20);
```

```
vector<double>::iterator itVec;
for (itVec=oVector.begin(); itVec<oVector.end(); itVec++)
  *itVec = 10.0;
```

as can be seen from this example, the iterator behaves like a pointer to array elements. The range over which it is incremented is defined by the values returned by `vector<>::begin()` and `vector<>::end()`. The `begin()` method returns the address of the first element of the container, and the `end()` method returns the address one element beyond the last element of the container. That is, the elements of the vector are in the half-open iterator range $[\text{begin}, \text{end})$.

The reason for this is clear if one considers the standard way of accessing array elements in C/C++ which we have previously seen for a `vector` container

```
for (unsigned int iVec=0; iVec<oVector.size(); iVec++)
  oVector[iVec] = 10.0;
```

which performs the same task as the iterator example above. In this case we use a looping variable in the range $[0, \text{size})$ which is the analogous half-open interval.

A more compact (and thus less readable) version of the iterator traversal which is often used is

```
vector<D> oVec(iSize);
for (vector<D>::iterator itVec=oVec.begin();
  itVec<oVec.begin(); itVec++)
      {
        // do something with *itVec here
      }
```

So if there are two equivalent versions of the container traversal, one using the iterator and the other using the convetional array indexing, why do we bother with iterators in the first place?

One reason is that some of the containers don't support element access through the array indexing operator, e.g. `list`, and for technical reasons pointer-like access would be inappropriate. Secondly, although iterators behave like pointers they are not, and can be given a variety of different behaviours. They provide a common mechanism for data access across all of the standard containers.

Compare the following two methods of traversing a vector

```
// create a vector and insert some things
vector<string> oVec;
oVec.push_back("string 1");
oVec.push_back("string 2");
oVec.push_back("string 3");

// forward iterator
for (vector<string>::iterator itStr=oVec.begin();
     itStr!=oVec.end();
     itStr++)
  cout << *itStr << " ";
cout << endl;

// reverse iterator
for (vector<string>::reverse_iterator itStr=oVec.rbegin();
     itStr!=oVec.rend();
     itStr++)
  cout << *itStr << " ";
cout << endl;
```

The iterator `vector<>::iterator` goes forwards through the container over the interval [begin, end). The second example is using a `reverse_iterator`, which traverses the interval [rbegin, rend). In the latter case, `rbegin` gives the last element of the container and `rend` points *beyond* the first element of the container, in the same sense that `end` points beyond the last element. Also note that incrementing a `reverse_iterator` gives a pointer to the *previous* element of the container.

7.3 Standard Algorithms

To go along with the standard containers are a set of algorithms for performing a variety of generic tasks. Generally one should use the statements `#include <numeric>` and `#include<algorithm>`. A description of some of the commonly used ones follows.

7.3.1 `max_element` and `min_element`

These algorithms return an iterator of the appropriate type pointing to the maximum or minimum element defined in an iterator range given by the arguments to the function. Consider the following example using a `vector`

```
vector<double> oVec(1000);
for (vector<double>::iterator itVec=oVec.begin();
     itVec<oVec.end(); itVec++)
  *itVec = sqrt(static_cast<double>(rand()));

cout << *min_element(oVec.begin(), oVec.end()) << " "
  << *max_element(oVec.begin(), oVec.end()) << endl;
```

Suppose that we wanted to know the index in the vector where the value is found instead of the value itself

```
unsigned int iVal;
iVal = static_cast<unsigned int>(
       max_element(oVec.begin(), oVec.end()) - oVec.begin()  );
cout << "index of maximum = " << iVal << endl;
```

where we have used the iterator analogy of pointer arithmetic.

7.3.2 `sort` and `reverse`

We can also sort container elements in an iterator range, and reverse their ordering

```
sort(oVec.begin(), oVec.end());
for (vector<double>::iterator itVec=oVec.begin();
     itVec<oVec.end(); itVec++)
  cout << *itVec << " ";
cout << endl;
reverse(oVec.begin(), oVec.end());
for (vector<double>::iterator itVec=oVec.begin();
     itVec<oVec.end(); itVec++)
  cout << *itVec << " ";
cout << endl;
```

and the `vector` elements have been printed out to show what has happened.

7.3.3 `accumulate`

The `accumulate` algorithm applies the `+` operator to elements in the iterator range given by the first two arguments and to the third argument, used to initialise the summation

```
double dSum = 0.0;
dSum = accumulate(oVec.begin(), oVec.end(), dSum);
cout << "Sum = " << dSum << endl;
```

and the result is returned by the function. what does the following code do?

```
cout << "answer = " <<
  accumulate(oVec.begin(), oVec.end(), 0.0) /
    static_cast<double>(oVec.size());
```

7.3.4 Others

There are many other algorithms in the standard library, details of which are given in the reference materials cited. They all use the general idea of operating upon container elements defined by an iterator range.

7.4 Function Objects

Although it appears that the algorithms described above are mostly arithmetic in nature, this is because the operators for the intrinsic types such as `double` are defined by the language to have an arithmetic meaning. So if we have a `vector<double>` object, then calling the sort algorithm reorders the vector according to comparisons made using the `<` operator.

But what if we have a vector of an arbitrary class type, then what will happen if we attempt to call `sort` with this as an argument? It will work provided that an implementation of the `<` operator is defined for this class. So the `sort` algorithm will run using the overloaded operator provided by the class designer.

Any of the standard algorithms can be made to work on an arbitrary class type provided that this class implements the operator which that particular algorithm uses to perform its task.

There is another way of giving operators associated with an arbitrary class type to a standard algorithm, and that is through the use of the concept of function objects. To create a class to perform an operation on a data type `double` which can be used as a function object, we overload the `()` operator as follows

```
class FunctionObject
{
public:
```

```
  double operator()(const double& dVal)
  {
    double dReturn;
    // compute some value
    return dReturn;
  };
};
```

and then in client code we can use an instance of `FunctionObject` as if it were a function

```
  FunctionObject oFunc;
  double dValue = 123.321;
  cout << oFunc(dValue) << endl;
```

The connection between function objects and the standard algorithms is an important one. The operator which the algorithm uses to perform its task is, unless specified, already determined. However, one can specify an operator other than the default one for an algorithm to use, as long as it is given in a particular way. It must be a function object.

There are many built-in function objects in the language with exactly this application in mind. Consider the following example, using the `sort()` algorithm

```
  vector<unsigned int> oVec(10);
  for (vector<unsigned int>::iterator itVec=oVec.begin();
    itVec<oVec.end(); itVec++)
      *itVec = rand();

  sort(oVec.begin(), oVec.end());

  for (vector<unsigned int>::iterator itVec=oVec.begin();
    itVec<oVec.end(); itVec++)
      cout << *itVec << " ";
  cout << endl;
```

So what we have done here is created a `vector` container and filled it up the random numbers from `rand()`, then called `sort()` and then printed out the results. When `sort()` is working here, it is comparing the elements of the container using the `<` operator, which is default behaviour. Now replace the above call to `sort()` with the following

```
sort(oVec.begin(), oVec.end(), less<unsigned int>());
```

in this case, we are calling an overloaded version of the `sort` function, and the third argument is creating one of the built-in function objects. This one is the less-than operator which acts upon `unsigned int` data; note that this is a template class.

Now try the following instead

```
sort(oVec.begin(), oVec.end(), greater<unsigned int>());
```

and see what happens. In these calls to `sort()`, where the function object is being passed to the algorithm `()` appears; this is not the fuction call operator itself, but to call the constructor for the class. The instance of the function object `less` or `greater` is created when the algorithm is called, and only exists for the duration of this call.

These objects can of course be created and used on their own

```
less<unsigned int> oLess;
cout << oLess(1, 2) << endl  // true
     << oLess(2, 1) << endl; // false
```

as with anything else.

As long as we conform to this interface, we can create a function object which can be called by an appropriate algorithm. Consider the `generate()` algorithm which accepts a container and sets each elements in the container equal to the result of a call to the function object passed in. Consider the following example

```
class FunctionObject
{
private:
  double dValue;
public:
  FunctionObject()
  {
    dValue = 1.0;
  };
  double operator()()
  {
    double dReturn = dValue;
```

```
    dValue += sqrt(dValue+1.0);
    return dReturn;
  };
};
```

and the following client code

```
vector<double> oVec(10);
generate(oVec.begin(), oVec.end(), FunctionObject());
for (vector<double>::iterator itVec=oVec.begin();
  itVec<oVec.end(); itVec++)
    cout << *itVec << " ";
cout << endl;
```

An instance of `FunctionObject` is created when the the call to `generate()` takes place. The algorithm then calls the function object `()` operator and assigns each element of the container to the result of that method call.

8 Input and Output

8.1 IO Streams

In the standard library, there is a series of related classes called streams. These are used in various ways for handling input and output to and from the console (as we have used already), files and classes such as strings. Streams can be used with other classes if they are appropriately customised. We require `#include<iostream>` to use these concepts.

We have already seen the use of the standard output stream to send data of all kinds to the console

```
cout << string("this is a string... ")
  << complex<double>(-12.32, 3.345) << endl;
```

In WinXP this means though the command line interface, `cmd.exe`, and in UNIX through the shell. The program may also receive input from the console, as follows

```
double dValue;
cout << "input a double value: ";
cin >> dValue;
cout << "answer = " << 2.0 * dValue << endl;
```

When the program execution reaches the line of code containing `cin`, it waits for input from the console, and when it is received, carries on. Data aquired by the program in this fashion must be carefully checked, however, to ensure that they are valid. The `cout` and `cin` objects are of type `ostream` (output stream) and `istream` (input stream) respectively.

We are calling the `<<` and `>>` operators. The first, `<<`, is called an inserter because it inserts a value *into* the stream object; the second, `>>` is called an extractor, because it extracts a value *from* the stream object. These are pre-defined in the language for built-in types such as `double` or `complex<>`, but for a custom class have to be defined in a particular way.

Consider the `pair<>` container. It happens that the `>>` and `<<` operators are members of the `istream` and `ostream` classes respectively. Therefore we can not create customised versions of these operators by overloading them in the client class (`pair` in this case). Instead we provide a version of `<<` and `>>` operator functions which can take as arguments `ostream` and `istream` on the LHS and `pair<>` as RHS:

```
ostream& operator<<(ostream& s, pair<double, double>& oPair)
{
  s << oPair.first << " " << oPair.second;
  return s;
};

istream& operator>>(istream& s, pair<double, double>& oPair)
{
  s >> oPair.first >> oPair.second;
  return s;
};
```

Of course these operator functions could be templates as well in order that we can use them on any type of `pair<>`. Now we can use the operator functions in client code as usual, and send `pair<>` objects to the console and read them back in again

```
  pair<double, double> oPair(1.0, 2.0);
  cout << oPair << endl;
  cin >> oPair;
  cout << oPair;
```

The `<<` or `>>` operators are not members of the client class, but might need to be able to access its private data. In this case, the operator function should be made a `friend` of the client class using the following syntax

```
class ClientClass
{
private:
  friend ostream& operator<<(ostream& s, ClientClass& oCC);

  double dVal01, dVal02;
public:
  // ...
};

ostream& operator<<(ostream& s, ClientClass& oCC)
{
  s << oCC.dVal01 << " " << oCC.dVal02;
  return s;
};
```

The operator `<<` is not a member of `ClientClass` but can access its private data because it is a friend.

8.2 File Streams

8.2.1 Basic usage

File input and output is done through file streams which are also part of the `istream` and `ostream` heirarchy (see Stroustrup for a painfully detailed discussion of these matters). We need the directive `#include<fstream>` to use these facilities.

Here is an example of reading a file (which already exists) using a `ifstream` (input file stream) object

```
  string sBuffer;
  ifstream oInFile("Filename.txt", ios::in);
  if (!oInFile.is_open())
  {
    cout << "File could not be opened. " << endl;
  }
  else // process data
  {
     while (!oInFile.eof())
     {
```

```
        getline(oInFile, sBuffer);
        cout << sBuffer << endl;
    }
    oInFile.close();
}
```

Firstly, we create an object of type `ifstream`. Its overloaded constructor is given two arguments, a `string` for the file name, and `ios::in`. The latter is an integer which tells the stream that the file is to be used for input and therefore must exist already. We then test the `is_open()` method to check if the file was sucessfully opened, and if not, give an error message. If all is OK at this stage, then we go on to loop while the `ifstream::eof()` method returns false (that is, while the end of the file has not beed reached), and at each iteration read in a line from the file. The line being read in is placed in the string `sBuffer`. This string is then output to the console in order to check what is going on. Other processing or parsing of the string would take place at this stage.

Note that in order for this to work, you will have to create a file with the appropriate name and place it in the same directory/folder as the executable for the project. Also put a few lines of text in that file.

Here is some code to write to a file, in the same style as the above

```
ofstream oOutFile("FileOut.txt", ios::out);
if (!oOutFile.is_open())
{
  cout << "File could not be opened. " << endl;
}
else
{
  oOutFile << "string 0" << endl;
  oOutFile << "string 1 " << endl;
  oOutFile << "string 234" << endl;
  oOutFile.close();
}
```

Firstly, an object of type `ofstream` is created, again with two arguments, a file name and a integer `ios::out`. The latter specifies that the file will be used for output only, and if it exists already, it will be truncated. The next line checks that the file is open, and if not, we then write out an error message. If the stream is open then we write a few lines and then call the `close()` method.

This is probably the easiest way to do file IO. There are other approaches, for example reading or writing single characters at a time, but this is much more complicated to implement and full of pitfalls.

8.2.2 More advanced usage

Suppose that you have a large amount of numerical data that needs to be written and subsequently read in from a file. Suppose for the moment that these data are `double` valued and in an array called `pData` of length `nData`, and we write them out in the following way

```
ofstream oOut("file.txt", ios::out);
if (!oOut.is_open())
{
  cout << "file could not be opened. " << endl;
}
else
{
  oOut << nData << " ";
  for (unsigned int i=0; i<nData; i++)
      oOut << pData[i] << " ";
}
```

which will first write out the length of the array and then write out the numbers in the array one at a time, separated by spaces. Now suppose we read them in as follows:

```
ifstream oIn("file.txt", ios::in);
if (!oIn.is_open())
{
  cout << "file could not be opened. " << endl;
}
else
{
  oIn >> nData;
  for (unsigned int i=0; i<nData; i++)
    oIn >> pData[i];
}
```

There is nothing wrong with this, but there are some things to consider.

1. The contents of this file will look something like

```
10 0.00125126 0.563585 0.193304 0.808741 0.585009 ...
```

so the values have been written with 6 significant figures (the default), considerably less than the precision with which the machine stores `double` variables (it is about 15-16). This can be changed by calling `cout.precision(10)` where the argument is the number of significant figures required.

2. Suppose now that we write out a number with a greater number of significant figures:

   ```
   1.4142135623731
   ```

   Each character of this number when in a text file occupies 1 byte, so a `double` variable takes 15 bytes when stored in the file. However, in the machine's memory it only uses 8. So clearly this is inefficient.

3. When you write or read numbers to text files (or to `cout` and `cin`) the computer has to convert the internal format used for storage into text and then back again. This takes time — not obvious for small amounts of data, but can make a big difference.

A solution to this is to write data to disk files in the exact form in which they are stored in the machine's memory; then the data on disk occupy the same amount of memory as they would when stored in RAM. In addition, there is no need to convert to and from text, thus making things faster. The only slight downside is that the data in the disk file will no longer be readable, but if you've got Gb of data, were you going to look at them anyway?

In any case, this is how we write data out:

```
ofstream oOut("file.dat", ios::out|ios::binary);
if (!oOut.is_open())
{
  cout << "file not opened. " << endl;
}
else
{
  oOut.write(reinterpret_cast<char const *>(&nData),
             sizeof(unsigned int));
  oOut.write(reinterpret_cast<char const *>(pData),
             nData*sizeof(double));
}
```

and now read the same thing back in again

```
ifstream oIn("file.dat", ios::in|ios::binary);
if (!oIn.is_open())
{
  cout << "file not opened.  " << endl;
}
else
{
  oOut.read(reinterpret_cast<char *>(&nData),
            sizeof(unsigned int));
  oOut.read(reinterpret_cast<char *>(pData),
            nData*sizeof(double));
}
```

Note that when doing these operations we do not write individual variables, but instead pass to the `write()` method a number of bytes to write out, and the place to start reading from (a pointer).

1. Arguments to `write()` are a `const` pointer and number of bytes; the pointer is `const` because we are reading from it. The pointer argument to `read()` is not const because we are writing *to* that memory location.

2. The `sizeof()` function returns the number of bytes occupied by a particular data type.

3. When we read data into the location `pData` this must already have had sufficient memory allocated to it to store whatever is going to be read in, just as if you were going to use it as an array.

Part II
Numerical Methods and Libraries

9 Random Numbers

One of the most important areas in financial engineering is Monte Carlo simulation, the most important element of which is generating random numbers. Or pseudo-random numbers at least, since we are going to use deterministic algorithms to do so. There is some code built into the STL for tasks such as these; in addition, there are many other libraries available which incorporate such functionality. In times of real desparation, we could even write some ourselves, and I will give one example to show that some algorithms can be very simple.

In the following I will show how to generate uniformly distributed variates (either floating point in the range $[0, 1]$ or integer valued in $[0, N)$); once you have these, you can use them to generate variates from many other densities.

9.1 A simple method

This is a very simple algorithm based on modulo arithmetic; we generate a sequence of the form

$$I_{j+1} = bI_j + c \mod m,$$

and with suitable choices of the constants b, c and m we get a sequence with a very long period. Some appropriate values are $b = 1664525$ and $c = 1013904223$, and the arithmetic is done modulo 2^{32}. So if we use a 32 bit representation of unsigned integer variables (which is what `unsigned int` is on a 32 bit machine), the code can be rather succinctly put as

```
unsigned int iVal = 0;
unsigned int b = 1664525,
             c = 1013904223;
for (unsigned int i=0; i<100; i++)
{
  iVal = b*iVal + c;
}
```

Note 1: This method is not recommended, but just given as an example of how simple (and fast) some methods can be. You will, from time to time, encounter code like this.

Note 2: This method is an example of what is called a *"linear congruential"* generator, if you want to look them up.

9.2 C++ Intrinsics

In the standard library (accessed using `#include <cstdlib>`) there are functions called `rand()` and `srand(int)`. The first of these returns an integer random number between 0 and `RAND_MAX` (a number defined by the compiler) and the second of these takes an integer argument and uses it to set the random number seed. These functions are provided by the creators of any particular compiler and so will vary between platforms.

The value of `RAND_MAX` that the Windows, Solaris and many other compilers use is 32767. If you want to use these functions to obtain double precision numbers uniformly distributed between 0 and 1, then simply divide the return value of `rand()` by `RAND_MAX`, making sure that appropriate casting is used.

```
cout << "RAND_MAX = " << RAND_MAX << endl;
for (unsigned int i=0; i<10; i++)
{
  double x = static_cast<double>(rand()) / RAND_MAX;
  cout << x << " " ;
}
cout << endl;
```

Suppose you now wish to now generate uniformly distributed numbers in the interval $[0, N)$, where N is a positive integer. The most obvious thing to do is to take the return value of `rand()` and take the remainder dividing by N. However (for reasons that we are not going into here) the lower order bits returned by `rand()` may not be as "random" as the rest of the quantity, so a better solution is to take the following

```
unsigned int N = 17;
for (unsigned int i=0; i<10; i++)
{
  double x = static_cast<double>(rand()) / RAND_MAX;
  unsigned int j = N*x;
  cout << j <<  " " ;
```

```
}
cout << endl;
```

Note 1: As with the previous example, using the system `rand()` is probably not the best option. If you are going to use it, then make sure to test it carefully.

9.3 Gaussian variates

In this section I will discuss how use uniformly distributed random numbers (floating point) on the interval $[0, 1]$ to obtain unit normal random variables. The following is the Box-Muller technique.

I am not going to discuss the theory of this method, except to note that unlike rejection based methods, two uniform variates in results in two normal variates out, so nothing is wasted. On the other hand, it requires evaluation of the functions sin, cos and log; there are other possibilities, but this version of the algorithm is the most commonly used.

So, taking uniformly distributed variates $u_1, u_2 \sim U[0, 1]$ i.i.d., the quantities

$$\begin{aligned} x_1 &= \mu + \sigma\sqrt{-2\ln u_1}\,\cos(2\pi u_2), \\ x_2 &= \mu + \sigma\sqrt{-2\ln u_1}\,\sin(2\pi u_2), \end{aligned}$$

are $x_1, x_2 \sim N(\mu, \sigma^2)$ i.i.d.

So here's the appropriate C++ code for doing the computation; this function returns $2N$ variates of the form described, although I've written it as a standalone function here, could be incorporated as a class method. I have used TNT; you could replace the TNT `Array1D` class with the STL `vector` class if you like, or anything else for that matter.

```
Array1D<double> BM(const double& mu,
                   const double& sigma,
                   unsigned int N)
{
  Array1D<double> oResults(2*N, 0.0);
  double u1, u2, dC1, dC2,
         dTP = 2.0*acos(-1.0);
  // first compute uniform variates
  for (unsigned int i=0; i<2*N; i++)
  {
    oResults[i] = Rand01();
```

```
    }
    // now transform them
    for (unsigned int i=0; i<N; i++)
    {
      u1 = oResults[2*i];
      u2 = oResults[2*i+1];
      dC1 = sqrt(-2.0*log(u1));
      dC2 = dTP*u2;
      oResults[2*i]   = mu + sigma*dC1*cos(dC2);
      oResults[2*i+1] = mu + sigma*dC1*sin(dC2);
    }
    return oResults;
  }
```

As you can see, something called `Rand01` is being called here; this is a placeholder for a function to compute uniform random numbers in the range $[0, 1]$. Filling in the details, as they say, is an exercise for the student.

10 The Boost Library

The boost library can be obtained from

```
http://www.boost.org
```

and contains many interesting and useful things. There are several classes which implement different algorithms for generating pseudo-random numbers, and a set of complementary classes which are used to represent different probability densities.

This course has neither the space or time to go into how to build and use the boost library, but I recommend that you consider it as a good quality and reliable resource.

11 Template Numerical Toolkit

The template numerical toolkit (TNT) is a collection of matrix and vector classes and associated algorithms for C++. It was created by Roldan Pozo at NIST, and is available from

```
http://math.nist.gov/tnt/
```

All of the code for the libraries is supplied in the header files, and the library does not have to be built to be used. All that one requires is the appropriate `#include` statements in the program.

The code is divided into two parts: TNT, which consists of the 1, 2 and 3 dimensional array classes, and JAMA which is a set of linear algebra functions. To use TNT in code, the following lines should be included before the `main()` function, or in any other class where TNT is required.

```
#include "tnt.h"
#include "jama_cholesky.h"
#include "jama_eig.h"
#include "jama_lu.h"
#include "jama_qr.h"
#include "jama_svd.h"
using namespace TNT;
using namespace JAMA;
```

11.1 Class `Array1D`

The `Array1D<>` template class is a one-dimensional array (i.e. vector) and can be instantiated in the usual ways

```
Array1D<double> oArray1D_01,
                oArray1D_02(10),
                oArray1D_03(4, 0.234);
Array1D<double> *pArray1D = new Array1D<double>(oArray1D_03);
```

where we are creating arrays which store `double` variables. The first instantiation calls the default constructor, and the size of the object is 0; the second has called an overloaded constructor and the object has length 10 elements, but they are *not* initialised; the third object calls another overloaded constructor to get length 4 with all elements initialised to 0.234; the final instantiation has called the copy constructor.

It happens that the inserter and extractor are already overloaded for the `Array1D` class, so they may be sent to console output

```
cout << oArray1D_01 << oArray1D_02 << oArray1D_03 << *pArray1D;
```

The `dim()` method returns the dimension of the object, and the addition, subtraction and assignment operators are all defined

```
oArray1D_02 = oArray1D_03;
oArray1D_01 = oArray1D_02 + oArray1D_03;
cout << oArray1D_01.dim() << endl;
cout << oArray1D_01 << endl;
```

note that when the object is inserted to `cout` the dimension of the vector is in the output anyway. The `*` and `/` operators are defined as well, but perform element-by-element operations on their two arguments, provided they are the same dimension,

```
cout << oArray1D_01 * oArray1D_02;
cout << oArray1D_01 / oArray1D_02;
```

which is the same scheme as used in Matlab. Element access is through the `[]` operator which can be used as both a LHS and RHS

```
oArray1D_01[0] = 3.0;
cout << oArray1D_01[0] << endl;
```

11.2 Class `Array2D`

The `Array2D<>` template class is a two-dimensional array or matrix, and can be instantiated as usual

```
Array2D<double> oArray2D_01(10, 10),
                oArray2D_02(2, 3, 1.0);
```

where again we are creating objects to store `double` types. The first object is created as a 10 by 10 matrix, but elements are not initialised, and the second is a 2 by 3 matrix, with all elements initialised to 1.0. The copy constructor can also be used. The numbers of rows and columns of an object are returned by the `dim1()` and `dim2()` methods respectively, and the array objects can be sent to `cout` in the same way as before

```
cout << oArray2D_02;
cout << oArray2D_02.dim1() << " "
     << oArray2D_02.dim2() << endl;
```

The assignment operator is defined, as well as `+`, `-`, `*`, `/` which again operate in an element-by-element fashion as for `Array1D<>`. One more operation which can be done is matrix multiplication, and a method called `matmult()` is used for this

```
Array2D<double> oArray2D_03(3, 2, 3.2);
Array2D<double> oArray2D_04 =
  matmult(oArray2D_02, oArray2D_03);
cout << oArray2D_04;
```

and the definition is as usual; the number of columns in the first argument have to match the number of rows in the second, or the empty container will be returned. Element access for the array object uses the `[]` operator as before, and can be used as LHS or RHS

```
oArray2D_04[0][0] = 1.0;
cout << oArray2D_04;
```

11.3 Algorithms

11.3.1 LU decomposition

Consider the set of linear equations

$$\mathbf{Ax} = \mathbf{b}$$

if the matrix $\mathbf{A}$ is written as the LU decomposition,

$$\mathbf{A} = \mathbf{LU}$$

where $\mathbf{L}$ and $\mathbf{U}$ are lower triangular and upper triangular respectively, then the equations can be solved by first evaluating $\mathbf{y}$ from

$$\mathbf{Ly} = \mathbf{b}$$

and then solving

$$\mathbf{Ux} = \mathbf{y}$$

for $\mathbf{y}$. Solving the triangular sets of equations is trivial, but how do we obtain the decomposition $\mathbf{A} = \mathbf{LU}$ in the first place?

There is no point in explaining here how or why the algorithm for this works, but the answer using TNT/JAMA is as follows

```
unsigned int iDim = 10;
Array2D<double> oA(iDim, iDim);
for (unsigned int i=0; i<iDim; i++)
  for (unsigned int j=0; j<iDim; j++)
```

```
    oA[i][j] = rand();
Array1D<double> oVec_b(iDim);
for (unsigned int i=0; i<iDim; i++)
  oVec_b[i] = rand();

LU<double> oLU(oA);
Array1D<double> oVec_x = oLU.solve(oVec_b);
```

The first thing that happens here is we create a matrix of `double` values and fill it up with random numbers, `oA`. Then we create a vector of right hand sides, `oVec_b`. Next we create an object of type `LU<double>`, which computes the LU decomposition of the matrix which it is passed as an argument on construction. We can then call the `solve()` method with `oVec_b` as an argument to find the vector $\mathbf{x}$ from the expression $\mathbf{Ax} = \mathbf{b}$.

As an exercise, check that this has been done correctly.

11.3.2 Eigenvalue solver

The right-eigenvectors and eigenvalues of a real-symmetric matrix $\mathbf{A}$ are defined by the columns of the orthogonal matrix $\mathbf{X}$

$$\mathbf{X}^T\mathbf{AX} = \lambda$$

where λ is diagonal. The matrices $\mathbf{X}$ and λ can be obtained using TNT by the following code

```
unsigned int iDim = 10;
Array2D<double> oG(iDim, iDim);
for (unsigned int i=0; i<iDim; i++)
  for (unsigned int j=i; j<iDim; j++)
  {
    oG[i][j] = rand();
    oG[j][i] = oG[i][j];
  }

Eigenvalue<double> oEIG(oG);

Array1D<double> oEigReal, oEigImag;
oEIG.getRealEigenvalues(oEigReal);
oEIG.getImagEigenvalues(oEigImag);
```

```
Array2D<double> oV;
oEIG.getV(oV);
```

The first thing that happens is we create a real symmetric (square) matrix, `oG`. We then create an object of type `Eigenvalue<double>`, and pass it the matrix of interest. The eigenvalues and eigenvectors of the matrix are computed on construction, and can then be returned by the subsequent calls, and sent to `cout` if desired.

As an exercise, convince yourself that this operation has been done correctly.

11.3.3 Cholesky factorisation

Given a symmetric positive definite matrix $\mathbf{A}$, the Cholesky factorisation of $\mathbf{A}$ is a lower triangular matrix $\mathbf{C}$ such that

$$\mathbf{A} = \mathbf{C}\mathbf{C}^T.$$

This can be done using another of the JAMA classes; suppose that you have an appropriate matrix `oP` already defined.

```
JAMA::Cholesky<double> oCholesky(oP);
if (oCholesky.is_spd())
{
  oChol = oCholesky.getL();
}
else
{
  cerr << "couldn't perform Cholesky factorisation. "
    << endl << flush;
  exit(1);
}
```

Note that the method `Cholesky::is_spd()` returns true if the matrix used was symmetric positive definite, meaning that the factorisation was completed, and the matrix $\mathbf{C}$ can be recovered. Otherwise, the factorisation in this form is not possible.

11.4 An example

So one final example of how we might use some of the computational technology described above. Suppose that we have a multivariate normal density

specified by a mean vector $\bar{\mathbf{x}}$ and covariance matrix $\mathbf{P}$, and we wish to generate samples from that density. Suppose that the number of dimensions is m. The following steps will do this job:

1. Form the lower triangular Cholesky factorisation $\mathbf{P} = \mathbf{C}\mathbf{C}^T$
2. Form a m-vector of unit normal samples, $\mathbf{z} \sim N(0, 1)$, i.i.d.
3. Form the vector $\mathbf{w} = \mathbf{C}\mathbf{z} + \bar{\mathbf{x}}$

then the vector $\mathbf{w}$ is distributed as required, $\mathbf{w} \sim N(\bar{\mathbf{x}}, \mathbf{P})$, i.i.d.