

C++ for Scientific Computing (2021)



Joe Pitt-Francis

Department of Computer Science

from material by Joe Pitt-Francis & Jonathan Whiteley

Administration

- Course material is at www.cs.ox.ac.uk/people/joe.pitt-francis/C++ScientificComputing/
- New for this year: prerecorded video lectures (released weekly)
- There will be lots of time to do practical exercises
 - More than enough material for course
 - Suggested pace on web-page (don't do them all)
- New this year: live Teams Q&A and office hours (Thurs @10)
 - Announcements
 - Discussion topics from lectures
 - Model answers (please ask in advance)
- Special topic assignment details on the web-page
- Teams/Email: Joe.Pitt-Francis@cs.ox.ac.uk

A few introductory remarks

- C++ is a common programming language for scientific computing applications, but it is not the only choice—other languages also have their merits
- Learning C++ allows you to learn other languages—e.g. FORTRAN, Java, C— very easily
- There are lots of resources for C++. I don't advocate you spending extra on books, but these are helpful
 - Books: Pitt-Francis & Whiteley: Guide to Scientific Computing in C++ and Meyers: Effective C++
 - Web: <http://www.cplusplus.com/>
- “Integrated Development Environments (IDEs)” exist—e.g. Eclipse and Visual Studio. These are very useful for managing large code development

C++ is ‘object-oriented’

- **Modularity:** All the data and methods of a particular object are held in one or two files – so can be worked on independently.
- **Abstraction:** The essential features and functionality are put in one place.
- **Encapsulation:** The implementation of an object is kept hidden from the caller (can be changed without breaking the interface).
- **Extensibility:** Code is easily re-used and extended.
- **Polymorphism:** The same code can be used for a variety of objects (or for different types of data).
- **Inheritance:** A powerful way to extend functionality.

Many lecture courses begin with these concepts. We'll cover the basics of programming without object orientation – and mention this on the way.

Lecture 1 — The basics

General structure of a basic C++ program

```

1 #include <header1>
2 #include <header2>
3 int main(int argc, char* argv[])
4 {
5     line of code;
6     // this is a comment, ignored by the compiler
7     more code; // this is a comment as well
8     /* Multi-line
9      * comment
10     */
11     return 0;
12 }
```

see: later

Things to note

1. Header files are listed first. These are files that contain the functions needed for operations such as input, output and mathematical calculations
2. There is a section of code that starts “`int main(int argc, char* argv[])`” (or “`int main()`” for brevity when command-line input is unimportant)
3. This section of code is followed by more code enclosed between curly brackets, `{` and `}`
4. Comments may easily be inserted into the code
5. Lines of code that “do something” end with a semicolon ;
6. Just before the closing curly bracket at the end of the code is a statement `return 0;`

A first C++ program

```

1 #include <iostream>
2
3 int main(int argc, char* argv[])
4 {
5     std::cout << "Hello World\n";
6
7     return 0;
8 }
```

Save this code as `hello_world.cpp`

This program prints the text “Hello World” to the screen

- `iostream` is a header file that is needed when using input and output
- `std::cout` is a command that sends output to the console, i.e. the screen
- `\n` is a formatting command that starts a new line
- All statements (lines of the program) inside the curly brackets end with a semicolon ;

Compiling the code

A key difference between MATLAB and C++—before the code can be executed it must be **compiled**

When using the Gnu compiler of Unix/Cygwin this code can be compiled by saving the code and typing

```
g++ -Wall -O -o hello_world hello_world.cpp
```

followed by return. This produces an executable called `hello_world` that can be executed by typing

```
./hello_world
```

Visual Studio can also be forced to compile on the command line (if you set the correct environment first):

```
cl /EHsc hello_world.cpp
```

Numerical variables

Before a variable is used the type of variable must be declared. For example if the variables `i` and `j` are integers and `a` is a double precision floating point number the statements

```
int i, j;
double a;
```

must be included in the program before these variables are used.

It is advisable to use `double` rather than `float` in scientific computing applications

Some names, such as `int`, `for`, `return` may not be used as variable names because they are used by the language. These words are known as **reserved words** or **keywords**

see: [example structure](#)

Some example lines of code that add two integers and print the answer to screen are

```
1 int integer1, integer2, answer;
2
3 integer1 = 5;
4 integer2 = 10;
5
6 answer = integer1 + integer2;
7
8 std::cout << "The sum of " << integer1 << " and "
9           << integer2 << " is " << answer << "\n";
```

The details of the `std::cout` statement will be explained later

More on numerical variables

A variable may be **initialised** when defining the variable type, for example

```
int i = 5;
```

The line above is equivalent to declaring `i` to be of type `int` and **assigning** it the value 5, in one step.

```
int i;
i = 5;
```

When assigning values to floating point variables it is good programming practice to write numbers with decimal points, i.e.

```
double a = 5.0;
```

rather than

```
double a = 5;
```

If a quantity is a constant throughout the program it may be declared as such

```
const double density = 45.621;
```

Some variable types and ranges in most 64-bit operating system are given below. Note the ranges are operating system dependent

Variable type	C++ name	Range in 64-bit
integer	<code>short int</code>	-2^{15} to $2^{15} - 1$ ($\approx 10^4$)
integer	<code>int</code>	-2^{31} to $2^{31} - 1$ ($\approx 10^9$)
integer	<code>long int</code>	-2^{63} to $2^{63} - 1$ ($\approx 10^{19}$)
unsigned integer	<code>unsigned int</code>	0 to $2^{32} - 1$
floating point	<code>float</code>	-3.4×10^{38} to 3.4×10^{38}
floating point	<code>double</code>	-1.7×10^{308} to 1.7×10^{308}
floating point	<code>long double</code>	-1.2×10^{4932} to 1.2×10^{4932}

Note that `long double` is identical to `double` on an older 32-bit architecture

Note that the size of `long int` is identical to `int` 32-bit architectures

There is a shorthand for some mathematical operations

Longhand	Shorthand
<code>a = a + b;</code>	<code>a += b;</code>
<code>a = a - b;</code>	<code>a -= b;</code>
<code>a = a * b;</code>	<code>a *= b;</code>
<code>a = a / b;</code>	<code>a /= b;</code>
<code>a = a + 1;</code>	<code>a++;</code> if <code>a</code> is an integer
<code>a = a - 1;</code>	<code>a--;</code> if <code>a</code> is an integer

Some example lines of code

```
1 float a, b;
2 double d, e;
3 a = 3.0;
4 b = ( a * pow(a, 7.5) ) / 2.0;
5 d = 4.0;
6 e = 2.0 * sqrt(d);
```

`pow(x,y)` gives the value of x^y .

`sqrt(d)` gives the square root of the variable `d`

When using mathematical functions such as `pow`, you should use the additional header file `cmath` and so you need the line of code

```
#include <cmath>
```

Division of an integer by another integer will almost certainly cause problems

An example is given in the following piece of code

```
int i = 5, j = 2, k;  
  
k = i / j;  
  
std::cout << k << "\n";
```

The variable `k` is an integer and so cannot store the true value, 2.5
Instead, it will store the value 2

Suppose an integer is divided by a variable of type `double` — or vice versa — and that the result returned is stored in a variable of type `double`, as shown in the code below.

The variable `k` in this code will contain the mathematically correct answer

```
double i = 5.0, k;  
int j = 2;  
  
k = i / j;  
  
std::cout << k << "\n";
```

However, this is bad programming practice and should be avoided

Suppose an integer is divided by a variable of type `double` — or vice versa — and that the result returned is stored in a variable of type `int`, as shown in the code below.

The variable `k` in this code is unable to store the mathematically correct answer

```
double i = 5.0;  
int j = 2, k;  
  
k = i / j;  
  
std::cout << k << "\n";
```

Only construct mathematical operations that are on elements of the same type

An integer can be converted to a different data type – see the next slide

Variables can be converted from one type to another for example

```
double i = 5.0, k;  
int j = 2;  
  
k = i / ((double) (j));  
  
std::cout << k << "\n";
```

In this example, `((double) (j))` allows the variable `j` to behave as if it were a double variable.

Modern C++ aside: the `auto` type

Modern C++ allows the type to be inferred if it is initialised to a value when it's declared.

```
auto i = 5.0;
auto j = 2;
```

This is useful when types are long (see Lecture 13).

But it is dangerous when we are using simple types such as `int` and `double`.

```
auto x = 20; // Compiler infers x as int
x += 2.5;
// Programmer might assume x is double
std::cout << "x = "<<x<<"\n"; // x = 22
```



To use modern C++ features compile with `g++ -std=c++11`

Elements of the array are accessed by placing the indices in separate square brackets, for example

```
array1[0] = 1;
array2[1][2] = 3.0;
```

Arrays can be initialised when they are declared, for example

```
double array1[3] = {0.0, 1.0, 2.0};
int array2[2][3] = { {1, 6, -4}, {2, 2, 2} };
```

where the array `array2` represents the matrix

$$\begin{pmatrix} 1 & 6 & -4 \\ 2 & 2 & 2 \end{pmatrix}$$

Arrays (static allocation)

An array is known to mathematicians as a matrix or, in one dimension, a vector

If the size of the array is known in advance then it can be declared as follows

```
int i = 5, j = 4, array1[4];
double array2[3][3]; //3 by 3 array
```

In contrast to MATLAB and FORTRAN the indices of an array of length `n` start at 0 and end at `n-1`

Either get used to the new index numbering, or define arrays to have one extra element

Note that the values of arrays may only be set using the curly bracket notation when they are declared—for example the code

```
int array[3] = {0, 1, 2};
```

is correct, but the code

```
int array[3];
array[3] = {0, 1, 2};
```

is not correct.

Boolean variables

These variables take the values `true` or `false`, and are of use when using `if` conditionals and `while` loops

They are used as follows:

```
bool flag;
flag = true;
```

Strings

A character is one letter or number, a string is an ordered collection of characters

For example, “C++” is a string consisting of the ordered list of characters “C”, “+”, and “+”

To use strings in C++ requires an extra header file as shown below

```
1 #include <iostream>
2 #include <string>
3 int main()
4 {
5     std::string city; // note the std::
6     city = "Oxford"; // note the double quotation marks
7     std::cout << city << "\n";
8 }
```

ASCII characters

ASCII characters are numbers, uppercase letters, lowercase letters and some other symbols

These characters may be represented using the data type `char`

```
1 #include <iostream>
2
3 int main()
4 {
5     char letter;
6     letter = 'a'; // note the single quotation marks
7
8     std::cout << "The character is " << letter << "\n";
9
10    return 0;
11 }
```

Tip: automated builders and IDEs

- Possibly start with an editor and a command-line compiler
- As projects get larger you need something to keep track of the compilation for you
- **Make** is a good dependency-checker: time-stamps determine what is re-made (Lecture 7). **CMake** is useful for configuration.
- **SCons** is a more sophisticated tool (written in Python). MD5 hashes are used to decide what needs to be re-made.
- Integrated development environments: Visual Studio (Windows), Borland (Windows), Xcode (OSX), Kdevelop (KDE) and Eclipse (multi-platform) either use Make/SCons or their own dependency checkers
- An IDE will help you to write code by cross-referencing between your files, saving your work and offering a debugger

Lecture 2 — Flow of control

The if statement

Suppose you want to execute two statements if the condition $p > q$ is met

This is achieved using the following code

```
if (p > q)
{
    statement1;
    statement2;
}
```

Note the indentation of the block between { and }. This makes it clear which statements are executed in the block

Third example – more than one condition

```
if (p == 0)
{
    statement1;
}
else if (p < 0)
{
    statement2;
    statement3;
}
else
{
    // p > 0
    statement4;
}
```

If only one statement is to be executed curly brackets aren't strictly necessary

For example, the following code will execute **statement1** if the condition $p > q$ is met

```
if (p > q)
    statement1;
```

but this is poor software engineering practice

Instead, write this code as

```
if (p > q)
{
    statement1;
}
```

The use of curly brackets makes it clear which statement(s) are to be executed

Fourth example – nested if statements

```
if (p < q)
{
    if (x >= y)
    {
        statement1;
    }
}
```

Fifth example – more than one condition

```
if (p < q || x < y)
{
    statement1;
}
```

statement1 is executed if and only if one or both of $p < q$ and $x < y$ is true - i.e. **||** is the logical **OR** operator

Relational and logical operators

relation	operator	
equal to	==	(note that it isn't "=")
not equal to	!=	
greater than	>	
less than	<	
greater than or equal to	>=	
less than or equal to	<=	

Boolean variables may be used in `if` statements as follows

```
1 bool flag1 = true, flag2 = false;
2 if (flag1)
3 {
4     std::cout << "Does print something" << "\n";
5 }
6 if (flag2)
7 {
8     std::cout << "Doesn't print anything" << "\n";
9 }
10 if (!flag2)
11 {
12     std::cout << "Does print something" << "\n";
13 }
```

logical condition	operator
AND	&&
OR	
NOT	!

The while statement

Similar syntax to `if` statements

```
while ( x < 100.0 && i < 10 )
{
    x += x;
    i++;
}
```

The condition `x < 100.0 && i < 10` is tested only at the beginning of the statements in the loop, and not after every statement.

For example if the loop is entered when `x = 99.0` and `i = 1`, the loop will be executed completely once.

Loop won't be entered when `x ≥ 100`: `x` and `i` will be unchanged.

If you need a loop to execute at least once, with a test at the end, use `do { ... } while (condition)`

for loops

The following loop executes the statements inside the loop 10 times.
Note that `i` must have been declared as an integer earlier in the code

```
for (i=0; i<10; i++)
{
    statement1;
    statement2;
}
```

Alternatively, the integer `i` may be declared inside the loop:

```
1 for (int i=0; i<10; i++)
2 {
3     statement1;
4     statement2;
5 }
```

The output from the section of code on the previous slide is

```
i = 0    j = 5
i = 0    j = 4
i = 0    j = 3
i = 0    j = 2
i = 0    j = 1
i = 1    j = 5
i = 1    j = 4
i = 1    j = 3
i = 1    j = 2
i = 2    j = 5
i = 2    j = 4
i = 2    j = 3
i = 3    j = 5
i = 3    j = 4
i = 4    j = 5
```

`for` loops can be nested and run over variable indices. The output from the following section of code is given on the next slide

```
1 for (int i=0; i<5; i++)
2 {
3     for (int j=5; j>i; j--)
4     {
5         std::cout << "i = " << i << "    j = " << j << "\n";
6     }
7 }
```

Use of `assert` statements for debugging

`assert` statements can be used in code to confirm something you expect to be true

For example, you may wish to confirm that a number you are about to take the square root of is non-negative

If the condition is not met, the code aborts giving an error message that explains what went wrong

To use `assert` statements you must use the header file `cassert`

An example of the use of `assert` statements is given on the next slide

```

1  #include <iostream>
2  #include <cassert>
3  #include <cmath>
4
5  int main()
6  {
7      double a;
8      std::cout << "Enter a non-negative number\n";
9      std::cin >> a;
10     assert(a >= 0.0);
11     std::cout << "The square root of a is " << sqrt(a) << "\n";
12     return 0;
13 }

```

If the code on the previous slide is compiled and run, and the user enters “-5” at the prompt, the code will be terminated and the following error message given:

```
msqrt:: msqrt.cpp:10: int main(): Assertion 'a >= 0.0' failed
```

This error message tells us to look at line 10 in `program.cpp`, where the assertion in quotes failed the test

Lecture 3 — Input and output

Console output

Console output may be achieved by using `std::cout`

We have already seen that the statement

```
std::cout << "Hello World\n";
```

prints the text “Hello World” to the screen, followed by a newline

The statements

```
int x = 1, y = 2;
std::cout << "x = " << x << " and y = " << y << "\n";
```

give output to the screen

```
x = 1 and y = 2
```

Note that any spaces required in the output must be included within quotation marks

Some useful formatting commands are

Command	Symbol
newline	<code>\n</code>
tab	<code>\t</code>
'	<code>\'</code>
”	<code>\”</code>
?	<code>\?</code>
(bell)	<code>\a</code>

Sometimes, for example if the computer is busy doing a large volume of computation, the program may not print the output to the screen immediately. If immediate output is desirable then use `std::cout.flush()` after any `std::cout` commands

```
std::cout << "Hello World\n";
std::cout.flush();
```

Keyboard input for strings is slightly different. An example is given below

```
1 #include <iostream>
2 #include <string>
3 int main()
4 {
5     std::string name;
6     std::cout << "Enter your name and then hit RETURN\n";
7     std::getline(std::cin, name);
8     std::cout << "Your name is " << name << "\n";
9     return 0;
10 }
```

Keyboard input

Keyboard input for numbers and characters is achieved using `std::cin`

The following statements prompts someone to enter their PIN

```
int pin;
std::cout << "Enter your PIN, then hit RETURN\n";
std::cin >> pin;
```

`cin` may be used to ask for more than one input at a time

```
int accountno, pin;
std::cout << "Enter your account number then hit RETURN,\n";
std::cout << "and then your PIN followed by RETURN\n";
std::cin >> accountno >> pin;
```

Redirecting output

Instead of printing the output to screen, you may want to write the output to file. Suppose the executable is called `my_prog`. The screen output may be redirected to the file `output` by the command

```
./my_prog > output
```

When output has been redirected, you may prefer to print errors to screen. This can be done using `std::cerr`

```
std::cerr << "Error - division by zero\n";
```

When output is redirected to the file `output`, the errors printed using `std::cerr` are not printed to file, only to the screen

Writing to file

When writing to file, an additional header function `fstream` is needed. It can be a good idea to include these header files whether or not they are needed:

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

The file `output.dat` may be opened using the statement

```
std::ofstream out("output.dat");
```

We can then write to this file in a similar manner as writing to the screen, with the exception that `cout` is replaced by `out`

There are a number of formatting options provided by C++

The following prints data in scientific format

```
1 #include <iostream>
2 #include <fstream>
3 int main()
4 {
5     double x = -1.0, y = 45.3275893627129, z = 0.00000001;
6     std::ofstream out("output.dat");
7     assert(out.is_open());
8     out.setf(std::ios::scientific|std::ios::showpos);
9     out << x << " " << y << " " << z << "\n";
10    out.close();
11    return 0;
12 }
```

Reading from file

Very similar to writing to file

Suppose the file `numbers.dat` (in my home directory) has 5 rows and 3 columns of numbers. This file can be read using the following code

```
1 double x, y, z;
2 std::ifstream input("/home/joe/numbers.dat");
3 assert(input.is_open());
4
5 for (int i=0; i<5; i++)
6 {
7     input >> x >> y >> z;
8     // ... do something with x, y, z ...
9 }
10 input.close();
```

It doesn't matter if we don't know how long the file is — we can read data until we reach the end of the file

Using the example on the previous slide where the file `numbers.dat` has an unknown number of rows and 3 columns we may use the following code

```
1 double x, y, z;
2 std::ifstream input("/home/joe/numbers.dat");
3 assert(input.is_open());
4
5 while (!input.eof())
6 {
7     input >> x >> y >> z;
8 }
9 input.close();
```

`input.eof()` is a Boolean variable that contains “true” if the end of file has been reached and “false” otherwise. (See exercises.)

Reading from command-line arguments

Command-line arguments (or flags) are read from the user of your program via the “argument count” and “argument values” parameters of `main`.

Here’s some example code that reads from the command-line

```
#include <iostream>
int main(int argc, char* argv[])
{
    for (int i=0; i<argc; i++)
    {
        std::cout<<"argv "<<i<<" is "<<argv[i]<<"\n";
    }
}
```

Here’s that code in action

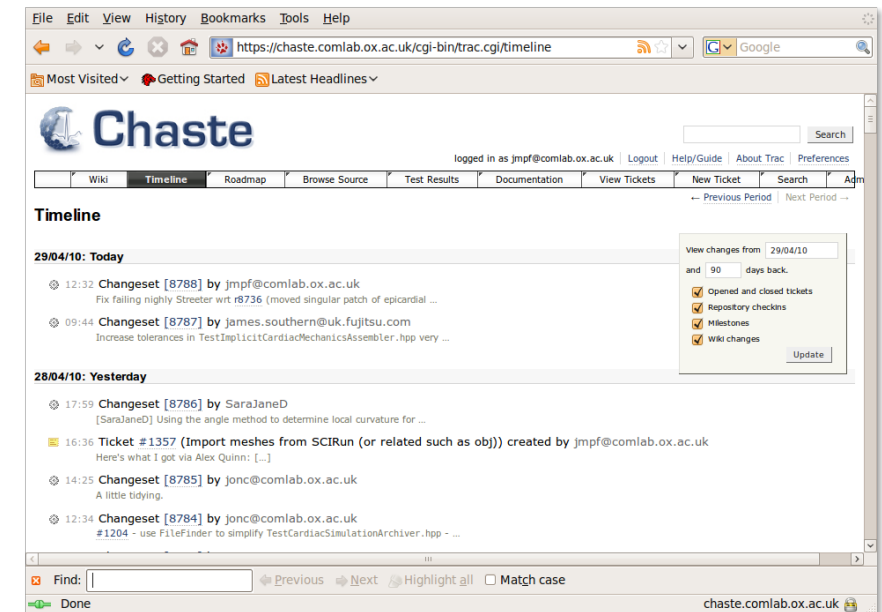
```
$ ./show_args
argv 0 is ./show_args
$ ./show_args one two
argv 0 is ./show_args
argv 1 is one
argv 2 is two
```

Note that the first element of the values `argv[0]` is always the name of program itself.

`argv` is an array of strings. If you want to interpret an argument as an integer then you will need `atoi`.

Tip: version control

- Big IDEs like Xcode and Visual Studio do this for you (in the same disk space)
- Modern version control systems like Subversion, Git and Mercurial let you work on a code-base concurrently
- There are many places that will host your project for you:
 - Bitbucket (Git and Mercurial)
 - GitHub (Git)
 - GitLab (Git)
 - Unfuddle (Subversion and Git)
- Eclipse has a Subversion and Git plug-ins (Subclipse & eGit)
- GitHub and Trac offer repository, bug-control and Wiki



Lecture 4 — Pointers and arrays

Pointers

A variable's address in the computer's memory is called a pointer

If a variable has been declared by

```
int total_sum
```

then the address of `total_sum` is given by `&total_sum`

`&total_sum` takes a constant value, because the address of `total_sum` in the computer's memory was allocated when it was declared. `&total_sum` is therefore known as a constant pointer

The contents of the memory that a pointer `p_x` points to is given by `*p_x`, for example

```
1 double y, z;    // y, z store double precision numbers
2 double* p_x;    // p_x stores the address
3                 // of a double precision number
4 z = 3.0;
5 p_x = &z;       // p_x stores the address of z
6 y = *p_x + 1.0; // *p_x is the contents of the memory p_x,
7                 // i.e. the value z
```

Note here that `* p_x` means two different things, depending on where it is

Variable pointers may be declared as follows

```
double* p_x;
int* p_i;
```

`p_x` is a variable pointer to a variable of type `double`, `p_i` is a variable pointer to an integer

Note that spacing can vary: `double* p_x` and `double *p_x` are equivalent, but the first is clearer.

If `p_x` and `p_y` are both to be declared as pointers this is done by

```
double *p_x, *p_y;
```

In the declaration

```
int *p_i, j;
```

`p_i` is a pointer to an integer, and `j` is an integer. It is generally clearer to declare each new pointer on a separate line.

A variable pointer cannot be used until first having been assigned a valid address, for example the following portion of code is incorrect

```
double* p_x; // p_x can store the address of a double
              // precision number - haven't said which
              // address yet

*p_x = 1.0; // trying to store the value 1.0 in an unspecified
            // memory location
```

because an address must first be assigned to `p_x` before a value is stored in that address

One way of assigning a valid address is to use the operator **new**

```
1 double* p_x;    // p_x can store the address of a double
2                // precision number
3
4 p_x = new double; // assigns an address to p_x
5 *p_x = 1.0;      // stores 1.0 in memory with
6                // address p_x
7 delete p_x;      // relinquish memory
```

Pointers should be used with care. Consider the following example:

```
1 double y;
2 double* p_x;
3 y = 3.0;
4 p_x = &y;
5 std::cout << "y = " << y << "\n";
6 *p_x = 1.0; // This changes the value of y
7 std::cout << "y = " << y << "\n";
```

The first time **y** is printed it takes the value 3: the second time **y** is printed it takes the value 1.

The line between the **cout** statements has altered the value of **y**, possibly unintentionally

Dynamic allocation of memory for arrays

Pointers can be used to allocate memory dynamically for arrays

Dynamic allocation of memory has several advantages

1. The size of the array doesn't need to be known at compile time
2. The size of the array may be changed while the program is running
3. If a large array is only needed for a small section of the code, memory can be allocated for the array when it is needed and de-allocated when it isn't
4. Efficient handling of irregularly sized arrays

Memory can be allocated using the **new** operator, and de-allocated using the **delete** operator

To create a one-dimensional array of double precision numbers of length 10 called **x** we use the following section of code

```
double* x;
x = new double [10];
```

The array then may be used as if it had been created by using the declaration

```
double x[10];
```

The memory allocated to **x** may be de-allocated by the command

```
delete[] x;
```

Always be sure to free any memory allocated—a code can very quickly use all available memory otherwise

In the dynamic allocation of the array allocated using the pointer **x** on the previous slide, **x** is the address of the first element of the array: this can be tested by using the code

```
std::cout << x << "\n";
std::cout << &x[0] << "\n";
```

To create a two-dimensional array of double precision numbers with 5 rows and 3 columns called **A** we use the following section of code

```
1 double** A;
2 A = new double* [5];
3 for (i=0; i<5; i++)
4 {
5     A[i] = new double[3];
6 }
```

The array then may be used as if it had been created by using the declaration

```
double A[5][3];
```

A is a pointer to a pointer:

- **A[i]** contains the address of **A[i][0]**
- **A** contains the address of the pointer **A[0]**

A is therefore an array of pointers

The memory allocated to **A** may be de-allocated by the code

```
1 for (i=0; i<5; i++)
2 {
3     delete[] A[i];
4 }
5 delete[] A;
```

Always be sure to delete any memory dynamically allocated, particularly memory allocated inside loops — if not you will run out of memory

Irregularly sized arrays

Suppose we want to define a lower triangular matrix A of integers with 10,000 rows and 10,000 columns

This may be done by the following declaration

```
int A[10000][10000];
```

but this wastes a considerable amount of memory saving the super-diagonal elements which are all 0

Instead, we may allocate the memory dynamically using the code on the following slide

```
1 int** A;
2 A = new int* [10000];
3 for (i=0; i<10000; i++)
4 {
5     A[i] = new int[i+1];
6 }
```

Modern C++ aside: the shared pointer

Modern C++ compilers allow the user to create new memory which doesn't need to be deleted explicitly.

```
1 int main()
2 {
3     std::shared_ptr<int> p_x(new int);
4     std::cout<<"p_x use count: "<<p_x.use_count()<<"\n";
5     *p_x = 5; // 'de-reference' to alter contents
6     std::shared_ptr<int> p_y = p_x;
7 }
```

A “number of uses” of the pointer maintained at run-time.

When this drops to zero, then the memory is automatically deleted.



To use modern C++ features compile with `g++ -std=c++11`

Tip: memory leak detection

```
1 double x[10];
2 for (int i=0; i<10; i++){
3     x[i]=i;
4 }
5 int total=0;
6 for (int i=0; i<=10; i++)//Equality is wrong
7 {
8     total += x[i];
9 }
10 std::cout<<"Total is "<<total<<"\n";
```

Lecture 5 — Blocks, functions and references

Blocks

A block is a piece of code between curly brackets

A variable, when declared, may be used throughout that block

```

1 {
2     int i;
3     i = 5;    // OK
4     {
5         int j;
6         i = 10; // OK
7         j = 10; // OK
8     }
9     j = 5;    // incorrect - j not declared here
10 }
```

The same name may be used for a variable both inside the block (local variable) and outside the block (global variable)

This is bad programming practice, as it can lead to confusion

```

1 {
2     int i = 5;
3     std::cout << i << "\n";
4     {
5         int i = 10;
6         std::cout << i << "\n"; // local value of i is 10
7         // variable i with value 5 is not accessible
8     }
9     std::cout << i << "\n"; // value of i is 5
10 }
```

Functions

The code on the next slide is an example containing a function that multiplies two double precision floating point numbers

Note the *function prototype* that is the second line of code

The function prototype tells the compiler about function's return value and parameters

The variable names *x* and *y* in the prototype are ignored by the compiler and don't have to be included. But including them can clarify the program

```

1 #include <iostream>
2
3 double multiply(double x, double y); // function prototype
4
5 int main()
6 {
7     double a = 1.0, b = 2.0, z;
8     z = multiply(a, b);
9     std::cout << a <<" times "<< b <<" equals "<< z <<"\n";
10    return 0;
11 }
12
13 double multiply(double x, double y)
14 {
15     return x * y;
16 }

```

A function may also return no value, and be declared as `void`

An example of a `void` function is shown on the next slide

The pass mark for an exam is 30 marks. This function prints out a message informing a candidate whether or not they have passed the exam

```

1 #include <iostream>
2 void output(int score, int passMark);
3
4 int main()
5 {
6     int score = 29, pass_mark = 30;
7     output(score, pass_mark);
8     return 0;
9 }
10 void output(int score, int passMark)
11 {
12     if (score >= passMark)
13         std::cout << "Pass - congratulations!\n";
14     else
15         std::cout << "Fail - better luck next time\n";
16 }

```

Note the poor software engineering practice on the previous slide – curly brackets should have been used with the `if` statements

Any variables that are used in the function must be declared as in the main program

For example

```

double mult5(double x)
{
    double y = 5.0;

    return x * y;
}

```

Under most conditions a function can only change the value of a variable inside the function, and not in the main program

This is because the code makes a copy of the variable sent to a function, and sends this copy to the function

On return from the function changes in this copied variable have no effect on the original variable

One exception to a function being unable to change the value of a variable outside a function is when an array – either dynamically allocated or not – is sent to a function

For example the following code does alter the value of `x[0]`

```
1 double x[10];
2 x[0] = 2.0;
3 someeffect(x);
4 std::cout << x[0] << "\n"; // will print out 3.0
5
6 void someeffect(double x[10])
7 {
8     x[0] += 1.0;
9 }
```

For example the following function has no effect on the variable `x` outside the function

```
1 x = 2.0;
2 noeffect(x);
3 std::cout << x << "\n"; // will print out 2.0
4
5 void noeffect(double x)
6 {
7     // x takes the value 2.0 here
8     x += 1.0;
9     // x takes the value 3.0 here
10 }
```

One method of allowing a function to change the value of a variable is to send the address of the variable to the function

```
1 #include <iostream>
2 void add(double x, double y, double* pz);
3 int main()
4 {
5     double a = 1.0, b = 2.0, z;
6     add(a, b, &z);
7     std::cout << a <<" plus " << b <<" equals " << z <<"\n";
8     return 0;
9 }
10
11 void add(double x, double y, double* pz)
12 {
13     *pz = x + y;
14 }
```

On the previous slide, the variables **a** and **b** are sent to the function—these values cannot be changed by the function

We also send the address of **z** to the function—we therefore cannot change the address of **z**, but we can change the contents of **pz**

The contents of **pz** are changed in the function using the line of code

```
*pz = x + y;
```

Suppose **a** is a vector whose size is allocated dynamically, and **b** is a matrix whose size is allocated dynamically. These arrays may be sent to the functions as follows. The prototype is

```
void example(int* a, int** b);
```

an example function is

```
void example(int* a, int** b)
{
    b[0][0] = a[1];
}
```

and this function may be called with the following statement

```
example(a, b);
```

for suitably declared **a** and **b**

Arrays whose sizes are allocated at compile time may be sent to functions as follows. There should be a prototype, for example

```
void example(int a[8], int b[3][3]);
```

and function

```
void example(int a[8], int b[3][3])
{
    b[0][0] = a[1];
}
```

This function may then be called with the following statement

```
int x[8], y[3][3];
example(x, y);
```

References

Another way of allowing a function to change the value of a variable outside the function is to use *references*

These are much easier to use: all that has to be done is the inclusion of the symbol **&** before the variable name in the declaration of the function and the prototype.

For example, see the code on the next slide

```

1  #include <iostream>
2
3  void add(double x, double y, double& rz);
4
5  int main()
6  {
7      double x = 1.0, y = 2.0, z;
8      add(x, y, z);
9      std::cout << x <<" plus " << y <<" equals " << z <<"\n";
10     return 0;
11 }
12
13 void add(double x, double y, double& rz)
14 {
15     rz = x + y;
16 }

```

Tip: use local variables

Consider the following piece of code

```

1  int i, j;
2  double a[10][10], b[10];
3  for (i=0; i<10; i++)
4  {
5      for (j=0; j<10; j++)
6      {
7          a[i][j] = 1.0;
8      }
9  }
10 for (i=0; i<10; i++)
11 {
12     b[j] = 10.0; // bug: should read b[i] = 10.0
13 }

```

This bug will not be picked up by the compiler

If the variables were localised within loops this would not happen—for example the code below is equivalent to that on the previous slide, but the compiler would flag the bug

```

1  double a[10][10], b[10];
2  for (int i=0; i<10; i++)
3  {
4      for (int j=0; j<10; j++)
5      {
6          a[i][j] = 1.0;
7      }
8  }
9  for (int i=0; i<10; i++)
10 {
11     b[j] = 10.0; // j is not defined here
12 }

```

Lecture 6 — Functions and modules

Default values for function parameters

It is possible to allow a function to be called without specifying all the parameters needed

Default parameters will be used for the other parameters

These parameters should be declared in the function prototype

The arguments with default parameters must be the last parameters in the parameter list

This should be used with care: it is easy to forget that default parameters exist

For example, a solver may be written

```
void solver(double x, double epsilon, int maxiter)
{
    ...
}
```

The function prototype may be written

```
void solver(double x, double epsilon = 0.0001, int maxiter = 100);
```

This solver may be called using any of the following

```
solver(x, 0.01, 10000);
solver(x, 0.01); // default value used for maxiter
solver(x);       // default value used for epsilon and maxiter
```

```
1  std::cout <<"7 times 10 equals "<< mult(7, 10) <<"\n";
2  std::cout <<"21.5 times 14.5 equals "<< mult(21.5, 14.5) <<"\n";
3
4  double mult(double x, double y)
5  {
6      return x * y;
7  }
8
9  int mult(int x, int y)
10 {
11     return x * y;
12 }
```

Function overloading also allows the definition of what is meant by multiplying a floating point number by an integer

Function overloading

When a function is declared, the return type and parameter type must be specified

If a function `mult` is to be written that multiplies two numbers, we would like it to work for floating point numbers and for integers

This can be achieved by *function overloading*

More than one function `mult` can be written—one that takes two integers and returns an integer, one that takes two floating point numbers and returns a floating point number, etc

Modules

A module is a collection of functions that performs a given task

An example of a module is a collection of functions that comprise a linear solver for solving the n by n matrix equation $A\mathbf{x} = \mathbf{b}$

Every module has an *interface* — this may be thought of as a list of variables that contains (i) those that must be input to the module, and (ii) those that are output by the module

The module may then be used as a “black box”, provided the interface is known

Using the example of the linear solver, this module may take form shown on the next slide


```

void SolveLinearSys(double** A, double* x, double* b, int n)
{
    //...lines of code;
    // a line of code that uses SolveFunc1;
    // a line of code that uses SolveFunc2;
}

void SolveFunc1(...)
{
    ...
}

void SolveFunc2(...)
{
    ...
}

```

Suppose you want to solve the m by m matrix equation $Pu = v$ using the module on the previous slide

After including the module in your code all you have to do is include the line of code

```
SolveLinearSys(P, u, v, m);
```

There is no need to understand the code used by this module.

All that is required is to understand the interface, i.e. the ordered list of variables in the line of code above

Modules are useful for code re-use, and for fast code development by programmers with no understanding of the operations that a module performs

Consider the linear solver example.

- Numerical analysts use linear solvers in almost every code they write. A module allows them to re-use this code rather than write a new linear solver each time.
- Other scientists with little mathematical expertise may have to solve a linear system. A module allows them to do so without learning how

But the use of modules may cause problems

Problems that may arise when using modules

Suppose the linear solver that has been written has been based on the GMRES solver

This solver requires the calculation of the scalar product between two vectors

A function would therefore be written to calculate the scalar product of two vectors of a given length

This function may be used by another module of the code, for example an ODE solver

Suppose whoever was programming the ODE solver decided to change the inputs to the scalar product function. This would inadvertently cause the linear solver to stop functioning correctly

The linear solver module could then **not** be treated as a black box

More problems

Suppose matrices are stored in an array as in a previous lecture

When a function is written to multiply two matrices, the arrays and the sizes of the arrays must be sent to this function

There is no way of checking that the size is correct

It would be more useful to have a new data type called “**matrix**” that contained both the size and the entries of the array—i.e. all the information is stored within one object

Encapsulation using classes

The shortcomings of modules described on the previous two slides for solving the linear system $A\mathbf{x} = \mathbf{b}$ could be overcome if we could write code in such a way that the variables A , \mathbf{x} and \mathbf{b} could be processed by a “module” that:

1. contains all the functions needed to solve the system;
2. can not be accessed by any other part of the program except through the interface; and
3. can not itself access any other part of the program
4. consistently handles data as well as functions

This is possible, through the use of **classes**

The specifications described above are known as **encapsulation**

Lecture 7 — An introduction to classes

As an example we will develop a class of books

Each book will have the following attributes:

- an author;
- a title;
- a format;
- a price;
- a year of publication; and
- a publisher.

We will begin by writing a class with these attributes, and then develop the class further

This class may be coded as

```

1 #include <string>
2
3 class Book
4 {
5 public:
6     std::string author, title, publisher, format;
7     int price; //Given in pence
8     int yearOfPublication;
9 };

```

Don't worry about the term **public** – that will be explained later

Note the semicolon after the curly bracket at the end of the class

Save the code on the previous slide as `Book.hpp`

The class can then be used using the code on the following slide

```

1  #include <iostream>
2  #include "Book.hpp"
3  int main()
4  {
5      Book fave_book;
6      fave_book.author = "Lewis Carroll";
7      fave_book.title = "Alice's adventures in Wonderland";
8      fave_book.publisher = "Macmillan";
9      fave_book.price = 199;
10     fave_book.format = "hardback";
11     fave_book.yearOfPublication = 1865;
12     std::cout << "Year of publication of "
13               << fave_book.title << " is "
14               << fave_book.yearOfPublication << "\n";
15     return 0;
16 }
```

Recall that we want to write functions that are associated only with the class

We will write a function `CalculateEuroPrice` that takes a floating point number (Euros to the pound) as input and returns the price of the book in Euro cents

We define the function inside the file `Book.hpp`

The body of the function is written inside another file `Book.cpp`

The files `Book.hpp` and `Book.cpp` are shown on the next two slides

Technically a function on an object is known as a *method* of the object

Book.hpp

```

1  #include <string>
2
3  class Book
4  {
5  public:
6      std::string author, title, publisher, format;
7      int price; //Given in pence
8      int yearOfPublication;
9      int CalculateEuroPrice(double rate);
10 };
```

Book.cpp

```

1 #include <cmath> // For ceil
2 #include "Book.hpp"
3
4 int Book::CalculateEuroPrice(double rate)
5 {
6     double euro_price = price*rate;
7     // Round up to nearest cent
8     return ((int) ceil(euro_price));
9 }

```

On the previous slide note that the function written is associated with the class **Book** through the statement

```
int Book::CalculateEuroPrice(double rate)
```

This function may be used outside the class by using statements such as

```
std::cout << fave_book.CalculateEuroPrice(1.13)<<"\n";
```

Some example code is given on the next slide

Loosely speaking, a list of variables and functions is included in the .hpp file, and the functions are included in the .cpp file

use_book.cpp

```

1 #include <iostream>
2 #include "Book.hpp"
3
4 int main()
5 {
6     Book fave_book;
7     fave_book.author = "Lewis Carroll";
8     fave_book.title = "Alice's adventures in Wonderland";
9     fave_book.price = 199;
10    fave_book.format = "hardback";
11
12    std::cout << "Price in Euro cents = "
13              << fave_book.CalculateEuroPrice(1.13)<<"\n";
14    return 0;
15 }

```

Compiling multiple files

Before we can compile the file **use_book.cpp** on the previous slide we first need to compile the **Book** class. This is done by using the **-c** option when compiling:

```
g++ -Wall -O -c Book.cpp
```

This produces an **object file** **Book.o**. We can now compile **use_book.cpp** by typing

```
g++ -Wall -O -o use_book use_book.cpp Book.o
```

The code may be run as before by typing

```
./use_book
```

Using a Makefile

Suppose a code `UseClasses.cpp` uses two classes: `Class1` and `Class2`

It is easy to forget to compile the files `Class1.cpp` and `Class2.cpp` each time they are modified

Also, you only need to compile classes that have been changed since they were last compiled

Makefiles are a very efficient solution to these problems

Save the following code as `Makefile`

```

1 all : UseClasses
2
3 Class1.o : Class1.cpp Class1.hpp
4     g++ -c -O Class1.cpp
5
6 Class2.o : Class2.cpp Class2.hpp
7     g++ -c -O Class2.cpp
8
9 UseClasses : Class1.o Class2.o UseClasses.cpp
10    g++ -O -o UseClasses Class1.o Class2.o UseClasses.cpp

```

The executable `UseClasses` may be created by typing
“`make UseClasses`”

When using a makefile, the line

```
UseClasses : Class1.o Class2.o UseClasses.cpp
```

means that the executable `UseClasses` depends on the files `Class1.o`, `Class2.o` and `UseClasses.cpp`

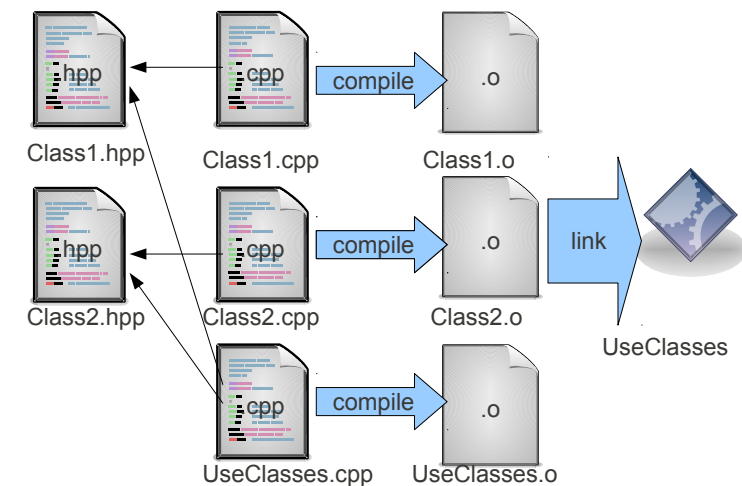
In turn, the line

```
Class1.o : Class1.cpp Class1.hpp
```

means that the object file `Class1.o` depends on the files `Class1.cpp` and `Class1.hpp`

The file `Class1.cpp` will only be compiled if `Class1.cpp` or `Class1.hpp` have changed since the last time `Class1.o` was created

When “`make`” is invoked without a target, it will attempt to build the first target in the `Makefile`



Setting variables

We can use functions to set variables — this allows us to check that the values assigned are sensible. For example

```
1 void Book::SetYearOfPublication(int year)
2 {
3     assert ((year > 1440) && (year < 2025));
4     yearOfPublication = year;
5 }
```

First we add the function definition to the file `Book.hpp` as we did with the function `CalculateEuroPrice`:

```
void SetYearOfPublication(int year);
```

Then we add the function on the previous slide to the file `Book.cpp`

This allows us to set the variable using statements such as

```
fave_book.SetYearOfPublication(1865);
```

Now we have written a function to set the variable `yearOfPublication` that checks that it takes an appropriate value, it seems sensible only to allow ourselves to assign a value to this variable through this function

We now need to think about access privileges

Access privileges

An instance of a class is known as an *object*. For example in the previous slide `fave_book` is an object of the class

Variables and functions associated with a class – for example `yearOfPublication` and `CalculateEuroPrice` – are known as *class members* and *methods*

There are three degrees of access to class members

- **private** - these class members are only accessible to other class members, unless `friend` is used
- **public** - these class members are accessible to everyone
- **protected** - these class members are accessible to other class members and to derived classes

To make the variable `yearOfPublication` only accessible from outside the class through the function `SetYearOfPublication` we make `yearOfPublication` a **private** class member

To highlight that it is now private we might also modify the name from `yearOfPublication` to `mYearOfPublication`:

```
1 class Book
2 {
3     private:
4         int mYearOfPublication;
5     public:
6         std::string author, title, publisher, format;
7         int price;
8         ...
```

The reserved words `private` and `public` may be used as often as desired. For example the following is acceptable code

```

1 class Book
2 {
3 public:
4     std::string author, title, publisher, format;
5 private:
6     int mYearOfPublication;
7 public:
8     int price;
9     ...

```

The default for variables in a class is `private`

For example, in the following code the variables `author`, `title`, `publisher`, `format` are actually `private`

```

1 class Book
2 {
3     std::string author, title, publisher, format;
4 private:
5     int mYearOfPublication;
6 public:
7     int price;
8     ...

```

But now we can't access the variable `mYearOfPublication` outside the class. We need to write a `public` class member to access this variable

```

1 int Book::GetYearOfPublication()
2 {
3     return mYearOfPublication;
4 }

```

and this function may be used in the main code as follows

```

1 std::cout << "Year of publication of "
2           << fave_book.title << " is "
3           << fave_book.GetYearOfPublication() << "\n";

```

It is good software engineering practice to access as many variables as possible in this way

Lecture 8 — More on classes

Constructors and Destructors

Each time an object of the class of books is created the program calls a function that allocates space in memory for all the variables used

This function is called a `constructor` and is automatically generated

This default constructor can be overridden if desired – for example in our class we may want to set all the string variables to “`unspecified`” when a new object is created

This function has the same name as the class, takes no arguments, has no return type and must be `public`

An overridden default constructor function is included in the class shown below

```

1 class Book
2 {
3 private:
4     int mYearOfPublication;
5 public:
6     Book();
7     ...

```

and the function is written

```

Book::Book()
{
    author = "unspecified";    title = "unspecified";
    publisher = "unspecified"; format = "unspecified";
}

```

Other constructors

Other constructors may be written.

For example you can write a constructor that initialises the title of a new `Book` object

This allows code to be written such as

```
Book recent_book("The explorer");
```

The constructor must be added to the list of class members in the file `Book.hpp`:

```
Book(std::string bookTitle);
```

Copy constructors

Another constructor that is automatically generated is a **copy constructor**

The line of code

```
Book fave_book_copy(fave_book);
```

will create another object `fave_book_copy` with variables initialised to those of `fave_book`

This constructor may also be overridden in the same way as for the default constructor

The constructor is then written as

```

Book::Book(std::string bookTitle)
{
    title = bookTitle;
}

```

This code is added to the file `Book.cpp`

You can write as many constructors as you like

Destructors

When an object leaves scope it is destroyed

A **destructor** is automatically created that deletes the variables associated with that object

Destructors can also be overridden – there will be an example later

Use of pointers to classes

An instance of a class can be sent to a function in the same way as data types such as **double**, **int**, etc.

This is shown in the example code on the next slide

```
1  #include "Book.hpp"
2  void SomeFunction(Book book);
3
4  int main()
5  {
6      Book fave_book;
7      SomeFunction(fave_book);
8      return 0;
9  }
10
11 void SomeFunction(Book book)
12 {
13     //...some code...
14 }
```

Note on the previous slide that as the function **SomeFunction** was called without the argument **fav_book** being a pointer, **SomeFunction** is unable to change any of the members of **fav_book** outside the body of the function

If you do want to change members of **fav_book** outside the body of the function, the function should be written to accept a pointer or a reference to **fav_book**.

Code for sending a **pointer** to **fav_book** into a function is shown on the next slide

```
#include "Book.hpp"
void SomeFunction(Book* pBook);

int main()
{
    Book fave_book;
    //...
    SomeFunction(&fave_book);
    return 0;
}

void SomeFunction(Book* pBook)
{
    // Uplift price by one pound
    (*pBook).price += 100;
}
```

The line of code on the previous slide

```
(*pBook).price += 100;
```

is a little clumsy. An equivalent statement is

```
pBook->price += 100;
```

Tip: coding standards

If you stick to a coding standard, then you're more likely to know what's what. For example,

- Proper indentation (with spaces rather than tabs)
- Braces ({}) on a line of their own
- Pointer names begin with 'p' (p_return_result)
- Locally declared names have underscores
- Names are meaningful (local_index)
- Method names arguments are in camel-case (firstDimension)
- Method names are in camel-case with verbs (GetSize())
- Class data is camel-case with 'm' to denote private (mSize)
- Lots of descriptive comments
- Avoid floating point comparisons, cut-and-paste...

An example from code I work with

```
void VentilationProblem::SetDynamicResistance(bool dynamicResistance)
{
    mDynamicResistance = dynamicResistance;
}

void VentilationProblem::SetPressureAtBoundaryNode(const Node<3>& rNode, double pressure)
{
    if (rNode.IsBoundaryNode() == false)
    {
        // Handle error ...
    }
    assert(mFluxGivenAtInflow == false);

    // Store the requirement in a map for the direct solver
    mPressureCondition[rNode.GetIndex()] = pressure;
}

double VentilationProblem::GetFluxAtOutflow()
{
    return mFlux[mOutletNodeIndex];
}
```

Lecture 9 — Inheritance and derived classes

Suppose we are running a bookshop and want to write a class of electronic e-books

An electronic book is a book, and so each object in this new class is also an object in the class of books

It has two special features: the format is “electronic” and it has an additional class member that contains a private URL

The “is a” relationship allows us to derive the class of e-books (Ebook) from the class of books (Book)

Each object in the class of e-books inherits many properties from the class of books

Inheritance allows us to re-use code

Book is the base class and Ebook is the derived class

The header file for the class of electronic books, Ebook.hpp may be written

```
1 #include <string>
2 #include "Book.hpp"
3
4 class Ebook: public Book
5 {
6 public:
7     Ebook();
8     std::string hiddenUrl;
9 };
```

The word “public” in the first line of code on the previous slide has the effect that

- public members of Book are public members of Ebook
- protected members of Book are protected members of Ebook
- private members of Book are hidden from Ebook

These access privileges may be changed by using protected or private

In practice public inheritance is far more common than protected or private inheritance

The file Ebook.cpp that contains the functions (methods) of the class is given by

```
1 #include "Ebook.hpp"
2
3 Ebook::Ebook()
4 : Book()
5 {
6     format = "electronic";
7 }
```

This constructor sets the format of all members of the class of electronic books to “electronic”

Example code using this class is on the next slide

Note that we can still use the functions and variables of the class Book when using an object of type Ebook

The code—which may be written on one line—on the previous slide

```
Ebook::Ebook()
: Book()
```

indicates which constructor in the class `Book` we want to call when creating an object of the class `Ebook`

To re-use the constructor of the class `Book` that required a string as input we would have

```
Ebook::Ebook(std::string bookTitle)
: Book(bookTitle)
{
    format = "electronic";
}
```

in the file `Ebook.cpp`

The following constructor is declared in the file `Ebook.hpp`

```
Ebook(std::string bookTitle);
```

and the original constructor

```
Book(std::string bookTitle);
```

was declared in the file `Book.hpp`

```
1 #include <iostream>
2 #include "Book.hpp"
3 #include "Ebook.hpp"
4
5 int main()
6 {
7     Ebook reading("The skull beneath the skin");
8     reading.author = "P D James";
9     std::cout << "The author is " << reading.author << "\n";
10    std::cout << "The title is " << reading.title << "\n";
11    std::cout << "The format is " << reading.format << "\n";
12
13    reading.hiddenUrl = "http://ebook.example.com/ex-book";
14    std::cout << "The URL is " << reading.hiddenUrl << "\n";
15    return 0;
16 }
```

It doesn't matter if we include header files such as `iostream`, `string`, etc. more than once

But we should not include files such as `Book.hpp` more than once

This can occur inadvertently: on the code on the previous slide we include the file `Book.hpp`.

The same header file is included in the file `Ebook.hpp`

We are therefore including the file `Book.hpp` twice—this can cause problems

To avoid this code being included twice we adapt our header functions to be of the form shown on the next slide

`example.hpp`

```

1 #ifndef EXAMPLEDEF__    // if variable EXAMPLEDEF__
2                        // not defined then execute lines of
3                        // code until #endif statement
4
5 #define EXAMPLEDEF__    // define the variable EXAMPLEDEF__.
6                        // Ensures that this code is only
7                        // compiled once, no matter how
8                        // many times it is included
9
10 class example
11 {
12     lines of code // body of header file
13 };
14 #endif // need one of these for every #ifndef statement

```

For example the header file `Book.hpp` would be written

```

1 #ifndef BOOKHEADERDEF__
2 #define BOOKHEADERDEF__
3
4 #include <string>
5
6 class Book
7 {
8 private:
9     int mYearOfPublication;
10     ...
11 };
12
13 #endif

```

A similar mechanism `#ifndef EBOOKHEADERDEF__` could be used in `Ebook.hpp` to prevent it from being read twice

Polymorphism

Polymorphism may be used when a number of classes are derived from the base class, and for some of these derived classes we want to override one of the functions of the base class

For example, consider a class of guests who stay at a hotel

The class guest will have variables such as name, room type, arrival date, number of nights booked, minibar bill, telephone bill

This class will also have a function that computes the total bill

Suppose the hotel has negotiated special rates for individuals from particular organisations. The function that computes the total bill will be different for these clients

This can be handled with `if` statements, but this can get messy. A more practical solution is to use **virtual functions** where the function used to compute the total bill does different things for different derived classes before returning the value of the total bill

This is known as **run-time polymorphism**

The function should be defined as **virtual** in the base class as shown on the next slide

The class `Guest` has header file `Guest.hpp`

```

1 #ifndef GUEST__
2 #define GUEST__
3 #include <string>
4 class Guest
5 {
6 public:
7     std::string name, roomType, arrivalDate;
8     int numberOfNights;
9     double minibarBill, telephoneBill;
10    virtual double CalculateBill();
11 };
12 #endif

```

The file `Guest.cpp` is

```

1 #include "Guest.hpp"
2
3 double Guest::CalculateBill()
4 {
5     double room_bill, total;
6
7     room_bill = numberOfNights * 50.0;
8     total = room_bill + minibarBill + telephoneBill;
9
10    return total;
11 }

```

Suppose the hotel have negotiated a deal with a company that reduces the room rate to £45 for the first night and £40 for subsequent nights that a guest stays in the hotel

The header file for our derived class `SpecialGuest.hpp` is

```

1 #ifndef SPECIALGUEST__
2 #define SPECIALGUEST__
3
4 #include "Guest.hpp"
5
6 class SpecialGuest : public Guest
7 {
8 public:
9     double CalculateBill();
10 };
11 #endif

```

The file `SpecialGuest.cpp` is

```

1 #include "SpecialGuest.hpp"
2
3 double SpecialGuest::CalculateBill()
4 {
5     double room_bill, total;
6
7     room_bill = (numberOfNights - 1) * 40.0 + 45.0;
8     total = room_bill + minibarBill;
9
10    return total;
11 }

```

Example use of the class `SpecialGuest` is

```

1  #include <iostream>
2  #include "Guest.hpp"
3  #include "SpecialGuest.hpp"
4
5  int main()
6  {
7      SpecialGuest harry;
8      harry.numberOfNights = 2;
9      harry.minibarBill = 30.99;
10
11     std::cout << "Harry's bill = "
12               << harry.CalculateBill() << "\n";
13
14     return 0;
15 }
```

Note that declaring the function `CalculateBill()` as virtual in the class `Guest` does not require that this function must be redefined in derived classes – instead it gives us the option to redefine it

If the function wasn't redefined then objects of the class `SpecialGuest` would use the function `CalculateBill()` defined in the class `Guest`

Note that the function `CalculateBill()` could have been declared as virtual in the class `SpecialGuest`

This would allow any class derived from `SpecialGuest` to redefine the function `CalculateBill()` if desired

When using derived classes, the destructor for the base class should always be a virtual function

Example of polymorphism in action. (Imagine an array of pointers to `Guests`.)

```

1  Guest* p_gu1 = new Guest;
2  Guest* p_gu2 = new Guest;
3  Guest* p_gu3 = new SpecialGuest; //Pointer of different type
4
5  //Set the three guests identically
6  p_gu1->numberOfNights = 3;
7  p_gu2->numberOfNights = 3;
8  p_gu3->numberOfNights = 3;
9
10 std::cout << "Bill 1 = " << p_gu1->CalculateBill() << "\n";
11 std::cout << "Bill 2 = " << p_gu2->CalculateBill() << "\n";
12 std::cout << "Bill 3 = " << p_gu3->CalculateBill() << "\n";
13 // The last one gets a smaller bill
```

Tip: plugging C++ into MATLAB

You may need to interface C++ with MATLAB (or with Gnu Octave)

- to get the speed of compiled code in a critical place
- to use an external library written in C++

This is possible with a MATLAB executable file (MEX)

In the simplest case the C++ code is a single file containing a function called `mexFunction` with a specific signature

The function `mexFunction` takes points to arrays for output and input

This is compiled with `mex` (a wrapper compiler to `g++`) and a `.mex` file is produced

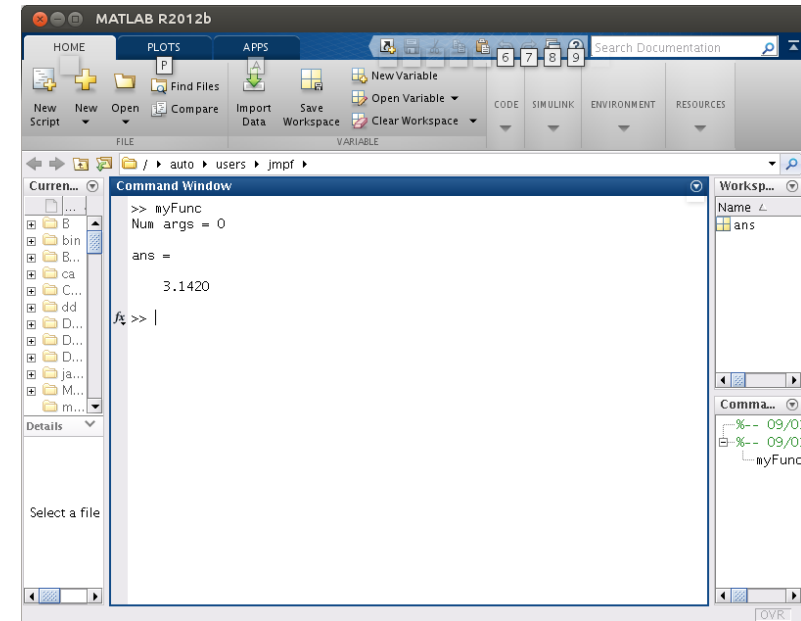
The `.mex` file is treated like a `.m` file by MATLAB

An example file myFunc.cpp

```

1 #include "mex.h"
2 #include <iostream>
3
4 void
5 mexFunction(int nlhs, mxArray* plhs[], int nrhs,
6             const mxArray* prhs[])
7 {
8     mxArray* v = mxCreateDoubleMatrix(1, 1, mxREAL);
9     double* data = mxGetPr(v);
10    *data = 3.142;
11    std::cout<<"Num args = "<<nrhs<<" \n";
12    plhs[0] = v;
13 }

```



Lecture 10 — A class of vectors

We will design a class of vectors in such a way that:

1. objects of this class behave like a new data type; and
2. code similar in style to MATLAB may be written using objects of this class.

We will define operations on and between objects of the class of vectors, and between these objects and data types such as `int` and `double`

We want to be able to write code such as

```

Vector u(3), v(3);    // vectors of length 3
Matrix A(3,3);        // matrix of size 3 by 3

v = A * u;
u = gmres(A, v);

```

Constructors for vectors

We want the declaration

```
Vector u(3);
```

to have the following effect:

- The compiler allocates memory for an array of three double precision floating point numbers and initialises these numbers to zero
- We can access the elements of `u`
- We can access the size of `u`

The code on the next two slides achieves these goals

The following file should be saved as **Vector.hpp**

```

1  #ifndef VECTOR_
2  #define VECTOR_
3  // a simple class of vectors
4  class Vector
5  {
6  public:
7      // member variables
8      double* data; // data stored in vector
9      int size;     // size of vector
10     // construct vector of given length
11     Vector(int sizeVal);
12 };
13 #endif

```

The following file should be saved as **Vector.cpp**

```

1  #include "Vector.hpp"
2
3  // constructor that creates vector of given size with
4  // double precision entries all initially set to zero
5  Vector::Vector(int sizeVal)
6  {
7      data = new double[sizeVal];
8      size = sizeVal;
9      for (int i=0; i<size; i++)
10     {
11         data[i] = 0.0;
12     }
13 }

```

An example of use of the class of vectors:

```

1  #include <iostream>
2  #include "Vector.hpp"
3
4  int main()
5  {
6      Vector u(3), v(3);
7      std::cout << u.size << "\n";
8      u.data[0] = v.data[1] = 5.0;
9      std::cout << u.data[0] + v.data[1] << "\n";
10     return 0;
11 }

```

The size of **u** is accessed by **u.size**

Element **n** of **u** is accessed by **u.data[n]**

see: neater way of accessing elements of **u**

Default constructor (no longer exists)

The default constructor is not appropriate when declaring a vector:
we need to know the length of the vector in advance

Because we have given an alternative constructor, and have not overridden the default constructor, the compiler automatically revokes the default constructor

The code

```
Vector a_vector;
```

will now give a compiler error.

Copy constructors

When using the copy constructor generated by the compiler statements such as

```
Vector w(u);
```

will not have the desired effect.

The length of the vector will be correctly set

However, instead of setting the elements of **w** equal to the elements of **u**, the pointer to the first element of **w** will be set to the pointer to the first element of **u**

This has the effect that the computer will attempt to store both **u** and **w** in the same memory space, which will obviously lead to errors

Statements such as

```
Vector w(u);
```

are not commonly used in MATLAB, there is little need for this constructor

However, if the compiler generated copy constructor is not overridden there may be errors

The copy constructor may as well be written. First the constructor must be added to the list of public members in the file **Vector.hpp**

```
1 Vector(const Vector& rOther);
```

The copy constructor may then be written

```
1 Vector::Vector(const Vector& rOther)
2 {
3     size = rOther.size;
4     data = new double[size];
5     for (int i=0; i<rOther.size; i++)
6     {
7         data[i] = rOther.data[i];
8     }
9 }
```

and should be included in the file **Vector.cpp**

Note the use of **const Vector& rOther** on the previous slide

The function could have been prototyped by

```
Vector(Vector otherVec)
```

The argument of the function would then not be a reference variable. A copy of this variable would be made for use in the constructor function. If the vector is big this will slow down the program

Using a reference variable allows us to use the same variable in the function

The qualifier **const** in the prototype ensures that this reference variable cannot be altered inside the function

Destructors

Recall that a destructor function is automatically generated and is called when a variable is destroyed, i.e. goes out of scope

We want our destructor to do more than this: we want it to free the memory that was allocated to the vector

We therefore need to override the default destructor

First the destructor must be included in the file `Vector.hpp`

```
~Vector();
```

The destructor should then be added to the file `Vector.cpp`

```
Vector::~Vector()
{
    delete[] data;
}
```

Destructors are called automatically

A further advantage of writing classes of vectors and matrices is that a destructor will automatically be called when a vector or matrix goes out of scope — this ensures that memory allocated to these objects is automatically deleted.

Functions

Recall that functions (*methods*) may be defined on classes

For example, the method `norm(p)`, the p -norm of `u`

We will assign `p` the default value 2

Add the following line of code into the list of `public` members of the class `Vector`

```
double norm(int p=2) const;
```

together with the function given on the following slide. The `const` keyword after the method informs the compiler that there should be no changes to the class. The norm of a `Vector` may be calculated using statements such as

```
x = u.norm();
y = v.norm(1);
```

```
1 double Vector::norm(int p) const
2 {
3     double temp, sum, norm_val;
4     sum = 0.0;
5     for (int i=0; i<size; i++)
6     {
7         temp = fabs(data[i]); // floating point absolute value
8         sum += pow(temp, p);
9     }
10    norm_val = pow(sum, 1.0/((double)p) );
11    return norm_val;
12 }
```

External function versus member method

We have written a function that calculates the p -norm a vector `u` using statements such as

```
dp = u.norm();
```

This notation is clumsy – we would rather write statements similar to those that we would use in MATLAB such as

```
dp = norm(u);
```

This can be done, but at the expense of compromising on good software engineering principles

Tip: documenting code

In the `norm` method on a previous slide, it is not obvious what is happening, even though there are only a few lines of code

Comments should be added to the code to aid anyone reading the code

For example a description of the function should be given first

```
1 // Function to calculate the p-norm of a vector
2 // See ‘‘An Introduction to Numerical Analysis’’ by
3 // Endre Suli and David Mayers, page 60, for definition
4 // of the p-norm of a vector
5
6 double Vector::norm(int p) const
7 {
8     ...
9 }
```

The function `norm` is declared as a **friend** in the file `Vector.hpp`

```
friend double norm(Vector vec, int p);
```

Declaring a function as a **friend** of a class allows this function to access the **private** members of the class

The function `norm` is now no longer encapsulated within the class of vectors, and so must be prototyped in `Vector.hpp`:

```
double norm(Vector vec, int p);
```

and is now written

```
double norm(Vector vec, int p)
{
    ...
}
```

see: another example of **friend**

Explain what is happening in the loop

```
1 // Loop over all elements of vector to calculate the
2 // sum required for the p-norm
3 sum = 0.0;
4 for (int i=0; i<size; i++)
5 {
6     temp = fabs(data[i]);
7     sum += pow(temp, p);
8 }
```

Documenting code is an art rather than a science

A few tips:

Describe what part of the problem the code is solving. Don't describe the code. For example, don't include documentation such as

```
// Loop over values of p going from 0 to n-1
for (p=0; p<n; p++)
```

Using lots of empty lines can make the code look more readable

If you want to emphasise something you can simulate underlining, for example

```
// Very important comment
// -----
```

Alternatively, can emphasise something by putting it in a box:

```
// *****
// *****
// **                                     **
// **   Function to calculate p-norm of vector   **
// **                                     **
// *****
// *****
```

Lecture 11 — Operator overloading

We want to write code such as

```
w = u + v;
```

where **u**, **v**, **w** are defined to be objects of the class **Vector**

We have to define within the class what is meant by the operators + and = in this context

This can be achieved by *overloading* the + operator and the = operator for the class of vectors

First we will restrict access to the data in the vector class

Overloading the () operator

We may overload the () operator in order to access elements of an array in the same way as MATLAB and FORTRAN

This allows us to assign indices from 1 to n inclusive for an array of length n, instead of 0 to n-1 as when using conventional C++

We will worry about error checking ($0 \leq i \leq \text{size}$) later

The function required to do this is

```
double& Vector::operator()(int i)
{
    return data[i-1]; // (NB we should check that i is in range)
}
```

and the following line must be included in the file **Vector.hpp**

```
double& operator()(int i);
```

We can now access the first element of `u` by writing `u(1)`, instead of the clumsy notation `u.data[0]`

Note the appearance of the symbol `&` on the previous slide

This indicates that the operator returns a reference

This allows us to use terms such as `u(1)` on the left hand side of expressions such as `u(1)=2.0`

We can also overload the square brackets operator with

```
double& operator[](int i);
```

(This is an exercise)

see: accessing elements of an array

Access privileges

Having overloaded the `()` operator we may now access elements of a vector using identical syntax to MATLAB.

We are now unlikely to access elements of a vector `u` by using the expressions of the form `u.data[2]`

There is now no need for the member `data` to be available outside the class, and so it could be declared as private

As discussed earlier, a good reason for declaring the member `data` as private is that this makes it harder for code to inadvertently alter the elements of a vector

We can also make the member `size` a private member, and access the length of the vector through a function `length` that replicates the `length` function in MATLAB

The renamed members `mData` and `mSize` are declared as private by writing the file `Vector.hpp` as follows

```
1 class Vector
2 {
3 private:
4     int mSize;
5     double* mData;
6 public:
7     Vector(int);
8     ...
9     friend int length(const Vector& rVec);
10    ...
11 };
12 int length(const Vector& rVec);
```

Note that the function `length` is declared as a `friend` of the class `Vector` and its prototype is also given

The function `length` is given below

```
int length(const Vector& rVec)
{
    return rVec.mSize;
}
```

Other functions that are used should also be declared as a `friend`

Binary operators

To write code such as

```
w = u + v;
```

then the lines

```
Vector& operator=(const Vector& rVec);
friend Vector operator+(const Vector& rVec1, const Vector& rVec2);
```

should be added within the class description in the file `Vector.hpp`, and then the following two functions should be included in the file `Vector.cpp`

```
1 Vector& Vector::operator=(const Vector& rVec)
2 {
3     // (We should check that the sizes match)
4
5     for (int i=0; i<rVec.mSize; i++)
6     {
7         mData[i] = rVec.mData[i];
8     }
9     return *this;
10 }
```

`this` is a pointer to the object

`*this` is the contents of the object that the pointer points at

```
1 Vector operator+(const Vector& rVec1,
2                 const Vector& rVec2)
3 {
4     // (We should check that the sizes match)
5
6     Vector result(rVec1.mSize);
7     for (int i=0; i<rVec1.mSize; i++)
8     {
9         result.mData[i] = rVec1.mData[i] + rVec2.mData[i];
10    }
11    return result;
12 }
```

It is essential that a destructor and copy constructor have been written before the code on the previous slide is used

A vector `result` is declared within the function—this will go out of scope at the end of the function (but its content will be copied)

If the destructor or copy constructor are not written correctly this may cause problems

The binary operators - and * can be overloaded in a similar way as to +

When overloading * we first have to define what $u * v$ means for a vector, i.e. do we mean the scalar product or the vector product?

We can also also overload * to define multiplication between an array of double precision numbers and a double precision number

For example, if a is a double precision floating point variable, and u is an array of double precision floating point numbers we can define what is meant by the operator * in the case $a*u$

```

1 Vector operator*(double a, const Vector& rVec)
2 {
3     Vector result(rVec.mSize);
4     for (int i=0; i<rVec.mSize; i++)
5     {
6         result.mData[i] = a * rVec.mData[i];
7     }
8     return result;
9 }

```

after the following line has been included into the file `Vector.hpp`

```
friend Vector operator*(double a, const Vector& rVec);
```

Binary operators without 'friend'

In overloading operator+ we have used an external friend function rather than a local method because it feels natural to be adding two objects.

```

1 Vector operator+(const Vector& rVec1,
2                 const Vector& rVec2)
3 {
4     Vector result(rVec1.mSize);
5     for (int i=0; i<rVec1.mSize; i++)
6     {
7         result.mData[i] = rVec1.mData[i] + rVec2.mData[i];
8     }
9     return result;
10 }

```

However, it is more efficient to write a binary operator as a member of a class.

```

1 Vector Vector::operator+(const Vector& rOther)
2 {
3     Vector result(mSize);
4     for (int i=0; i<mSize; i++)
5     {
6         result.mData[i] = mData[i] + rOther.mData[i];
7     }
8     return result;
9 }

```

Both operators would be instantiated as $a = b+c$, but in the case of the second style, it would be run as an internal method of b . (`mSize` evaluates to b 's `mSize`.)

see: previous use of `friend`

Unary operators

The unary operators - and + may also be overloaded in a similar manner to binary operators: add the line

```
friend Vector operator-(const Vector& rVec);
```

to the list of public members of Vector in the file Vector.hpp, and add the function on the following slide to the file Vector.cpp

```
1 Vector operator-(const Vector& rVec)
2 {
3     // (We should check that the sizes match)
4
5     Vector result(rVec.mSize);
6     for (int i=0; i<rVec.mSize; i++)
7     {
8         result.mData[i] = -rVec.mData[i];
9     }
10    return result;
11 }
```

Overloading the output operator

C++ does not know how to print out a vector, unless you tell it how. (If you print a pointer, then a memory address will be printed.)

Overload the << operator in the hpp file:

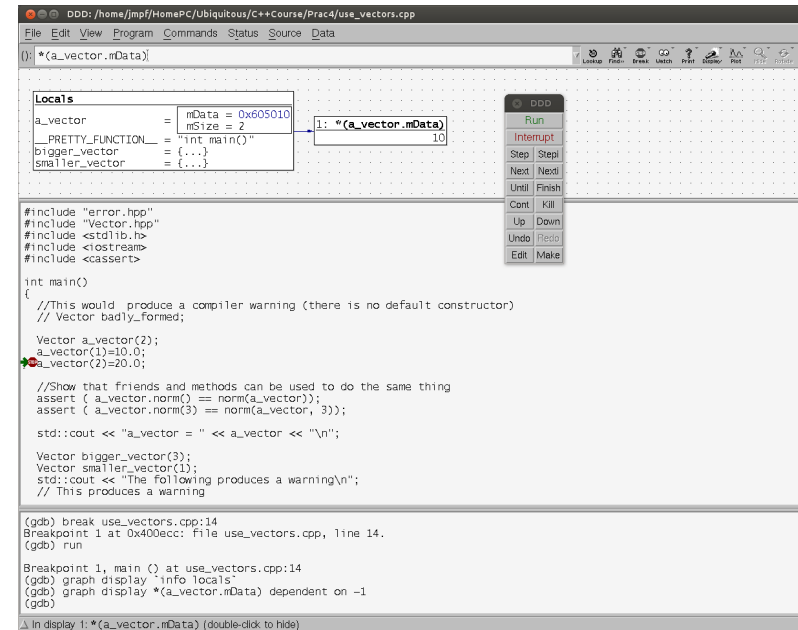
```
class Vector
{
private:
    int mSize;
    double* x;
public:
    Vector(int);
    ...
    friend std::ostream&
        operator<<(std::ostream& output, const Vector& rVec);
};
```

The implementation might look like this:

```
1 // std::cout << "a_vector = " << a_vector << "\n";
2 // appears as: a_vector = (10, 20)
3 std::ostream& operator<<
4     (std::ostream& output, const Vector& rVec)
5 {
6     output << "(";
7     for (int i=0; i<rVec.mSize; i++){
8         output << rVec.mData[i];
9         if (i != rVec.mSize-1)
10            output << ", ";
11        else
12            output << ")";
13    }
14    return output; // for multiple << operators.
15 }
```

Tip: step-through debuggers

- Visual Studio and Borland Delphi etc. have their own integrated step-through debuggers
- Eclipse offers a debugging plug-in which by default links to Gnu's `gdb` debugger, but supplies a user interface over the top
- Other graphical front-ends to `gdb` exist. I like `ddd`.
- **Note:** Executable files suitable for debugging need to be compiled with `-g` (debug) rather than `-O` (optimized).



Lecture 12 — Exceptions

Suppose we have lines of code that read

```
bigger_vector = a_vector;
smaller_vector = a_vector;
```

where `smaller_vector`, `a_vector` and `bigger_vector` are vectors that have been declared as having 1, 2 and 3 elements respectively.

The assignment operator is expecting that the size of its input vector (on the right-hand side) matches the object which it is assigning to.

There are clearly errors here – the current implementation will attempt to add too much data into `smaller_vector`.

What should the program do when the sizes do not match?

The answer is – “It depends”.

It's good to have a hierarchy of errors

Level 1 If the error can be fixed safely, then fix it. If need be, warn the user.

Level 2 If the error could be caused by user input then throw exception up to calling code, since the calling code should have enough context to fix the problem.

Level 3 If the error should not happen under normal circumstances then trip an assertion.

Exceptions are a compromise between carrying on regardless and stopping completely.

Exceptions require use of the keywords `try`, `throw` and `catch`

`try` tells the code to execute some statements

`throw` identifies an error

`catch` attempts to fix the error

We will use the example of assigning to a vector of the wrong length using the overloaded `=` operator for vectors

When assigning to a longer vector we will treat it as a *Level 1* error – pad the extra entries with zeroes and warn the user. When assigning to a shorter vector we will treat it as a *Level 2* error and throw an exception, because data would be lost otherwise.

When an error occurs we want the code to “throw” two pieces of information

1. A summary
2. A description of the error

We will write a class `Exception` to store these two pieces of information, and with the ability to print this information when required

The file `Exception.hpp` may be written

```

1  #ifndef EXCEPTION__
2  #define EXCEPTION__
3  #include <string>
4  class Exception
5  {
6  public:
7      std::string problem, summary;
8      Exception(std::string sum, std::string prob);
9      void DebugPrint();
10 };
11 #endif

```

The file `Exception.cpp` may be written

```

1  #include "Exception.hpp"
2
3  Exception::Exception(std::string sum, std::string prob)
4  {
5      problem = prob;
6      summary = sum;
7  }
8
9  void Exception::DebugPrint()
10 {
11     std::cerr << "** Exception (" << summary << ") **\n";
12     std::cerr << "Problem: " << problem << "\n\n";
13 }

```

Here's the new assignment operator (2 slides)

```

1 Vector& Vector::operator=(const Vector& rVec)
2 {
3     // if rhs vector is too long then throw
4     // if rhs vector is too short, assume missing entries are 0
5     if (rVec.mSize == mSize)
6     {
7         for (int i=0; i<mSize; i++)
8             mData[i] = rVec.mData[i];
9     }
10    else if (rVec.mSize > mSize)
11    {
12        throw Exception("length mismatch",
13            "vector assignment operator - vectors have different lengths");
14    }

```

```

15    else //if (rVec.mSize < mSize)
16    {
17        for (int i=0; i<rVec.mSize; i++)
18            mData[i] = rVec.mData[i];
19        for (int i=rVec.mSize; i<mSize; i++)
20            mData[i] = 0.0;
21        std::cout << "vector assignment - copied vector too short";
22        std::cout << " and has been extended with zeroes\n";
23    }
24    return *this;
25 }

```

We may now test the exception written in our overloaded assignment operator for vectors

```

1 Vector smaller_vector(1);
2 Vector a_vector(2);
3 Vector bigger_vector(3);
4 //This produces a warning
5 bigger_vector = a_vector;
6 //This produces an exception
7 try
8 {
9     smaller_vector = a_vector;
10 }
11 catch (Exception& err)
12 {
13     err.DebugPrint();
14 }

```

Tip: test first

- Test driven development means that you always start with the code for a test (not the code itself)
- Choose the simplest piece of functionality which you want to implement first
- Make a test *before* you make the implementation (it won't compile and it won't pass)
- Write the missing functionality until the test passes
- Always check that all tests pass as you add new functionality (then you know as soon as the program gives different behaviour)

An example from my own code (with CxxTest)

```
void TestPatientData() throw (Exception)
{
    VentilationProblem problem("continuum_mechanics/test/data/all_of_tree", 0u);
    problem.SetOutflowPressure(0.0);
    problem.SetConstantInflowPressures(50.0);
    TetrahedralMesh<1, 3>& r_mesh=problem.rGetMesh();
    TS_ASSERT_EQUALS(r_mesh.GetNumNodes(), 56379u);
    TS_ASSERT_EQUALS(r_mesh.GetNumElements(), 56378u);

    problem.SetDynamicResistance(false);
    problem.Solve();
    std::vector<double> flux, pressure;
    problem.GetSolutionAsFluxesAndPressures(flux, pressure);
    double top_radius = r_mesh.GetNode(0)->rGetNodeAttributes()[0];
    TS_ASSERT_DELTA(top_radius, 8.0517, 1e-4); //mm
    double top_reynolds_number = fabs( 2.0 * problem.GetDensity() * flux[0]
                                     / (problem.GetViscosity() * M_PI * top_radius) );

    // Poiseuille
    TS_ASSERT_DELTA(problem.GetFluxAtOutflow(), -7.975182e6, 1.0);
    TS_ASSERT_DELTA(top_reynolds_number, 49591, 1.0);
    //...
}
```

Lecture 13 — Templates and the STL

- Templates introduce compile-time polymorphism: generics
- They are used where the same code may need to be repeated for small numbers of different values or for different types

```
1 mesh_3d.SolvePoisson();
2 mesh_2d.Create();
3
4 double GetMin(double a, double b)
5 {
6     if (a<b){return a;} return b;
7 }
8 int GetMin(int a, int b)
9 {
10    if (a<b){return a;} return b;
11 }
```

Use the template keyword to produce as many functions as may be required

```
1 template <class T>
2 T GetMin (T a, T b) {
3     if (a<b){
4         return a;
5     }
6     return b; // When a>=b
7 }
8
9 main(){
10     std::cout << GetMin<int>(10,-2) << "\n";
11     double ans=GetMin<double>(22.0/7.0, M_PI);
12 }
```

Each new instance of the function requires the code to be re-compiled for that particular value

```
1 template<unsigned DIM>
2 class TemplatedVector
3 {
4     double mData[DIM]; // Static size (fixed at compile-time)
5 public:
6     double& operator[](int pos){
7         assert(pos<DIM); return(mData[pos]);
8     }
9 };
10 int main()
11 {
12     TemplatedVector<5> a;
13     a[0]=10;    a[1]=11;
14     std::cout<<a[0]+a[1]<<"\n";
15     a[5]=0; //Trips assertion
16 }
```

The Standard Template Library (STL)

This is a set of commonly used patterns which can be re-used for different types of objects

- Containers. e.g. random access vectors, linked lists
- Algorithms. e.g. sorting
- Iterators
- Special containers e.g. Queues and maps

- The vector template class provides a form of dynamic array that expands at the end as necessary to accommodate additional elements
- It is declared with `std::vector<type>`
- It contains methods like `bool empty()`, `push_back(value)`, `int size()`, `int capacity()`, `reserve(int)`
- The implementation is responsible for doing the memory management for you
- If you grow a vector dynamically, then STL may make a new larger space for it and move it. Therefore, it's not good to store the addresses of vector elements elsewhere.
- Insertions/deletions at the end of the vector are fast
- Insertions/deletions at the front will take linear time
- `std::deque` has fast insertions at front or back

```

1  std::vector<std::string> SS;
2
3  SS.push_back("The number is 10");
4  SS.push_back("The number is 20");
5  SS.push_back("The number is 30");
6  // Loop over index
7  for(int ii=0; ii < SS.size(); ii++)
8  {
9      std::cout << SS[ii] << "\n";
10 }
11 //Loop with iterator
12 std::vector<std::string>::const_iterator cii;
13 for(cii=SS.begin(); cii!=SS.end(); cii++)
14 {
15     std::cout << *cii << "\n";
16 }

```

- The map template class provides the machinery to make a mathematical map
- This lets us recall the value to which a particular key maps, rapidly
- The internal organisation of a map relies on the ability to compare the values of keys
- Many plain data types (int, double) have obvious comparison functions. `std::string` types can be compared lexicographically.
- For more complicated keys, you need to write and add a definition of the 'less than' operator
- There are also `set` (like map with just the keys), `bag` (like set, but with multiple copies of the same element) and `multimap` (allowing the same key to be mapped to many things)

```

1  std::map<std::string, int> Phonebook;
2
3  Phonebook["Joe"] = 83511;
4  Phonebook["Sandy"] = 15208;
5  Phonebook["Sam"] = 10666;
6  std::cout << "Phonebook[Joe]=" << Phonebook["Joe"] << "\n\n";
7  std::cout << "Map size: " << Phonebook.size() << "\n";
8
9  for( std::map<std::string,int>::iterator
10      ii=Phonebook.begin(); ii!=Phonebook.end(); ii++)
11  {
12      std::cout << (*ii).first << ": " << (*ii).second << "\n";
13  }
14  assert(Phonebook.count("Laura") == 0);

```

This class lets us do comparison (lexicographical) on 2D points

```

1  class Point2d
2  {
3      int x, y;
4  public:
5      Point2d(int xval, int yval)
6      {
7          x=xval; y=yval;
8      }
9      bool operator<(const Point2d& r0ther) const
10     {
11         if (x < r0ther.x) return true;
12         if (x > r0ther.x) return false;
13         return (y < r0ther.y); //Returns false if y>=r0ther.y
14     }
15 };

```

Thus we get key comparison (two-dimensional point comparison) for use in a set

```

1  int main()
2  {
3      std::set<Point2d> points;
4      Point2d origin(0,0);
5      points.insert(origin);
6      points.insert(Point2d(0,1));
7      points.insert(Point2d(1,0));
8      points.insert(Point2d(0,0)); //No different from origin
9      std::cout<<points.size()<<"\n";
10     std::cout<<points.count(Point2d(0,0))<<"\n";
11
12 }

```

Comparison can be used for sorting

```

1  #include <algorithm>
2  ...
3      std::vector<int> squares_mod_10;
4      for (int i=0; i<10; i++){
5          squares_mod_10.push_back( rand() % 10 );
6      }
7      // e.g. 3 6 7 5 3 5 6 2 9 1
8      sort(squares_mod_10.begin(), squares_mod_10.end());
9      for (int i=0; i< squares_mod_10.size(); i++){
10         std::cout<<squares_mod_10[i]<<" ";
11     }
12     std::cout<<"\n";
13     // e.g. 1 2 3 3 5 5 6 6 7 9

```

Modern C++ aside: enriched features in the STL

Modern C++ contains many useful features which have been added to the STL:

- a more accessible array container
- easier initialisation of containers
- easier iteration over containers

```
std::vector<int> nums = {0, 1, 2};
// Use a reference to alter the members
for (int& r_num : nums) r_num++
// More compact form
for (auto n:nums) std::cout<<n<<"\n";
```



To use modern C++ features compile with `g++ -std=c++11`

- STL may look heavy-weight at first, since you have to put lots of lists of things into angle-brackets
- Some of the use of iterators (e.g. set union) look very clumsy
- STL concentrates on frequently used design patterns and it's good to know the patterns (even if you don't use the library)
- STL functionality is highly-optimised by compiler writers to give complexity assurances and a low memory-footprint

Tip: re-use of robust libraries

If you write it yourself then you understand it better. However, it's not likely to be correct first time and it's not likely to be the optimal solution. Using code that has had decades of effort put into it is **better**, even if the learning curve is steeper.

- Standard Template Library for extensible vectors, sorting searching, indexing and mappings
- Boost for simple added functionality such as serialisation and small matrices
- PETSc for large-scale, robust and sparse linear algebra
- MPI for distributed-memory parallelism
- ...

The end — Useful tips

Software carpentry: You can increase your programming productivity (in terms of speed and of correctness) by using available tools

The “tips” you have seen apply to all languages (not just to C++)

- Automated builders and IDEs
- Version control
- Memory leak detection
- Coding standards
- Documenting code
- Step-through debuggers
- Test first
- Re-use of robust libraries

Summary

- Administration
- Lecture 1 — The basics
- Lecture 2 — Flow of control
- Lecture 3 — Input and output
- Lecture 4 — Pointers and arrays
- Lecture 5 — Blocks, functions and references
- Lecture 6 — Functions and modules
- Lecture 7 — An introduction to classes
- Lecture 8 — More on classes
- Lecture 9 — Inheritance and derived classes
- Lecture 10 — A class of vectors
- Lecture 11 — Operator overloading
- Lecture 12 — Exceptions
- Lecture 13 — Templates and the STL
- The end — Useful tips

- 26 ASCII characters
- 27 Strings
- 28 Tip: automated builders and IDEs
- 29 Lecture 2 — Flow of control
- 29 The if statement
- 33 Relational and logical operators
- 36 The while statement
- 37 for loops
- 40 Use of `assert` statements for debugging
- 43 Lecture 3 — Input and output
- 43 Console output
- 46 Keyboard input
- 48 Redirecting output

Index

- 2 Administration
- 3 A few introductory remarks
- 4 C++ is ‘object-oriented’
- 5 Lecture 1 — The basics
- 5 General structure of a basic C++ program
- 7 A first C++ program
- 9 Compiling the code
- 10 Numerical variables
- 12 More on numerical variables
- 21 Modern C++ aside: the `auto` type
- 22 Arrays (static allocation)
- 25 Boolean variables

- 49 Writing to file
- 51 Reading from file
- 53 Reading from command-line arguments
- 55 Tip: version control
- 57 Lecture 4 — Pointers and arrays
- 57 Pointers
- 63 Dynamic allocation of memory for arrays
- 69 Irregularly sized arrays
- 71 Modern C++ aside: the shared pointer
- 72 Tip: memory leak detection
- 74 Lecture 5 — Blocks, functions and references
- 74 Blocks
- 76 Functions

88	References
90	Tip: use local variables
92	Lecture 6 — Functions and modules
92	Default values for function parameters
94	Function overloading
96	Modules
100	Problems that may arise when using modules
101	More problems
102	Encapsulation using classes
103	Lecture 7 — An introduction to classes
112	Compiling multiple files
113	Using a Makefile
117	Setting variables

164	Default constructor (no longer exists)
165	Copy constructors
169	Destructors
171	Functions
173	External function versus member method
175	Tip: documenting code
179	Lecture 11 — Operator overloading
180	Overloading the () operator
182	Access privileges
185	Binary operators
191	Binary operators without ‘friend’
193	Unary operators
195	Overloading the output operator

119	Access privileges
124	Lecture 8 — More on classes
124	Constructors and Destructors
126	Copy constructors
127	Other constructors
129	Destructors
130	Use of pointers to classes
135	Tip: coding standards
137	Lecture 9 — Inheritance and derived classes
147	Polymorphism
156	Tip: plugging C++ into MATLAB
159	Lecture 10 — A class of vectors
160	Constructors for vectors

197	Tip: step-through debuggers
199	Lecture 12 — Exceptions
208	Tip: test first
210	Lecture 13 — Templates and the STL
213	The Standard Template Library (STL)
221	Modern C++ aside: enriched features in the STL
223	Tip: re-use of robust libraries
224	The end — Useful tips
225	Summary
226	Index