Timetable

There are two lecture slots for this module on Mon/Tues and two Computing labs scheduled for Thursday and Friday afternoons.

This is largely a 'hands-on' module. So the Labs form an important element. During the labs you will be expected to do the weekly exercises, which form an important component of the module.

Students with Surnames beginning

A - L should attend the Thursday Lab

M - Z should attend the Friday Lab.
Assessment is by means of the exercises (30%)

And a final exam (70%)

The exercises are important, if you do the exercises, you should do well in the exam!
Exercises

An exercise sheet will be handed out most weeks and is due by the following Friday (midnight).

Your work should be submitted via the web forms accessible from the module web page. Please ensure you fill in the form correctly. Model answers will appear on the website following the submission deadline.

Don't submit other people's work...you will be easily caught!
C++ is a standard language, so any book offering an introduction to the language should be fine in terms of the basic features. However, many books don't have many examples from scientific programming. The following book has a lot of scientific programming examples relevant to this module.

**C++ FOR ENGINEERS AND SCIENTISTS**

Gary Bronson

ISBN 978143939502
A good reference book which covers the C, C++ and Java Languages:

H.M. Deitel and P.J. Deitel,

`C how to Program (3rd Edition)',


ISBN: 0130895725
The de facto standard C++ reference book, by the creator of the language, which you may wish to refer to is:

B. Stroustrup,

`The C++ Programming Language',

Addison-Wesley, (1997).

ISBN: 0201889544

(A New edition is to be released in April)
What is C++?

C++ is a general purpose programming language intended to be a "better C".

Unlike some other languages (e.g. Java), C++ (and "C") is a compiled language. This means it is intrinsically fast and good, particularly for CPU-intense scientific computing.

It is very close to being a superset of the C programming language which was itself was developed in the '70's to write the Unix operating system.

It can be thought of as "C with new features to facilitate Object Oriented programming". But because it still retains all the features of "C" it can be used for low-level programming. It is one of a family of languages with similar syntax.

First developed in early '80's but ISO/ANSI standardization process completed in 1998. The current standard extending C++ with new features was ratified and published by ISO in September 2011.
Why use C++

Until fairly recently, procedural languages such as Fortran was felt to be adequate by most physicists. However, some areas of physics such as HEP have been pushing Fortran to its limits (millions lines of code) and have typically had to use C for some functions.

C++ is an established open standard, and there is at least one high-quality Open Source compiler (g++) which has been ported to many platforms including Linux.

Code written using Object Oriented techniques can make life much easier:) (facilitates code reuse)

"Market Forces", C++ use is increasing rapidly and widely, Fortran use is more-or-less restricted to Scientific Computing.

BUT

C++ is much harder than a traditional languages such as Fortran, perhaps ~10 times more complex! It is more difficult to master, and it is easy to write inefficient code:(. However, with skill it is possible to write code more efficient than Fortran:(
Compiling and linking a C++ Program
Dev-C++ is a full-featured Integrated Development Environment (IDE) for C++ programming. It is available from the website:

http://www.bloodshed.net/devcpp.html

It uses the free GCC (GNU Compiler Collection), which is the standard compiler for Linux, as its compiler, so programs can be written that are easily ported to other environments.

The Dev-C++ IDE, installed on the student network is a powerful system that allows one to build programs for both Windows and console-based applications. However, because the primary aim of this course is to teach you the skills needed to write simple scientific command line programs which interact with the user only via the console. I recommend that you build and run your programs as console applications. Such programs also have the advantage in that they should also be readily portable to the (more powerful) Unix/Linux systems used, for example, in your projects and for research in the Physics Department.
Using the Compiler

An Integrated Development Environment allows the programmer to carry out all stages of development (editing, compiling, linking, debugging and testing) from within the same application. To use this environment to develop a program, do the following:

Start the IDE by selecting from the Public Menu: 12 Programming --> GNU Dev C++ Compiler --> DEVCPP This should start an IDE for the compiler.
Using the Compiler

From the "File" menu select the New Project item. This will bring up the following dialogue:

![New Project Dialogue]

- **Windows Application**
- **Console Application**
- **Static Library**
- **DLL**
- **Empty Project**

**Description:**
A console application (MSDOS window)

**Project options:**
- **C Project**
- **C++ Project**
- **Make Default Language**

Name: `G:\MCPS\hello`
Using the Compiler

Give a project name corresponding to the program name, e.g. For the "hello" program I called the project "hello".

Select Console Application

and specify a C++ Project

Once you have created a project a "tree" of program files should be visible.

You should now be able to write/edit the main .cpp file by clicking on the .cpp node:
Using the Compiler

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World" << endl;
    return 0;
}
```
Using the Compiler

Build the program by selecting the compile option from the "Execute" menu.

This produces a window showing whether the compilation was successful or produces errors or warnings:
Using the Compiler

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```
Using the Compiler

If there are any compilation errors then a dialogue will look something like:
An extra diagnostics window will appear which should help you find your mistake. You should be able to click on message to go to the line in which (the compiler) thinks the error occurred.

With some errors, such as e.g. missing semi-colons the line containing the error may occur somewhere before the point the compiler an error.
To Run a console program it is best to create a Windows console.

You should change the working directory to the directory (folder) which you have created to hold your work:
You can then run a program by typing its name:
Compiling and linking a C++ Program

Working with C++

- Edit
- Source file
- C++ compiler
- Object files executable
- Load + run
Source Code

The Source Code for a typical C++ program consists of one or more header files, which usually have the suffix .h or .hh. These usually contain declarations of variables, functions, and classes.

Header files are either provided by the programmer or supplied with the compiler, operating system or some external package. They allow the programmer to access standard library functions. e.g. the maths functions sin(), cos() etc.

The C++ Source Files, which usually have the suffix .cpp or .C, contain the implementations (definitions) of the functions etc declared in the header files. i.e. the actual lines of C++ code.
Compiling and linking a C++ Program

What happens when the Compiler is run?

To produce an executable program from the source code requires a 3-stage process. The first 2 steps are carried out independently for each .cpp file.

1) Pre-process: First the C pre-processor (cpp) is run on each source (.cpp) file. This interprets pre-processor directives, which are indicated by the # symbol. The most important of these is the #include statement. The statement

    #include "afile.h"

places the contents of the local header file "afile.h" in the body of the code. Whilst

    #include <afile>

places the contents of the standard header file afile in the code.
Compiling and linking a C++ Program

What happens when the Compiler is run?

(2) Compile: The expanded source code file resulting from pre-processing (translation unit) is compiled to produce an object file (suffix .obj or .o) containing the raw machine instructions.

(3) Linking: The Linker (Loader) combines one or more object files together with standard libraries (which contains collection of object files) to produce an executable (suffix .exe or no suffix). This is the program that one actually runs.
/ The classic first example of a C++ program

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World" << endl;

    // causes program to pause
    system("PAUSE");
    return 0;
}
The main() function (program)

C++ programs basically consist of a series of building blocks functions). The most important of these is the main() function. When a program is run the main() function is automatically called. When the main() finishes, after the return statement the program is terminated. Therefore the whole program must be constructed around the main() function.

The statement using namespace std; inports a set of symbols defined in the standard namespace including, for example, cout so they can be used in your programs. You will probably need to have this in all your programs.
**Simple Text Output**

The statement:

```
cout << "Hello World" << endl;
```

prints the string "Hello World" on the Standard Output Stream. The pair of less than signs `<<` is a special operator. The special keyword `endl` denotes an end-of-line. (One can also use the newline character escape sequence "\n")

Several things can be printed separated by `<<` symbols. e.g.

```
cout << " x= " << x << " y= " << y << endl;
```

prints the string " x= " followed by the value of the variable `x` then does the same for `y`.

The use of iostreams is discussed in more detail **later in the course**.
General C++ Syntax

General Rules of C++ Syntax

C++ is a strongly typed Language

You MUST declare all variables/functions/classes before you use them.

C++ is Case Sensitive

The case of the text used in a C++ program Always matters

All the language keywords and standard library functions are lower case.

Can use any mixture of upper and lower case letters in variable and function names (except you must not use names which are reserved keywords).
General C++ Syntax

**Code Format**

C++ uses a Free Format: In (most) cases the amount of white space doesn't matter. So you can use tabs etc

Every statement must be terminated by a semi-colon:

```
;
```

Blocks of code (which define variable scope) are defined using "Curlies":

```
{...
```

There are several popular styles for laying out C++ codes, particularly as to where to place the "Curlies", and indent I will (try to) follow that of Stroustrup
Comments

There are two styles of comments in C++:

C++ style line comments:

```
// this text is a comment
```

Which indicate the rest of the text on the line is a comment

C style block comments:

```
/*
   Here are several lines of comments
*/
```

Which indicate several lines of text are comments
Variables

A variable is a name representing a location in the computer's memory in which one can store, manipulate and retrieve data.

C++, like most other programming languages, has many different types of variables. However, there are only 3 basic types. Integer variables, which are mostly used for counting, Floating point variables which can handle non-integer quantities and scientific notation and variables which can be used to store text characters.

For scientific computing we are obviously going to use a lot of floating point variables. But we are going to have to use integer and character quantities as well. We need to understand when to use the different variable types.
Variables which represent integer type quantities. These are identified by the keywords:

- `short [int]`
- `int`
- `long [int]`

The [int] here is optional (and usually missed out).

On a modern 32-bit or 64-bit computer system the plain int variable type, which is 32 bits (or 4 bytes) in size, is sufficient for most purposes and this is integer variable type that you should normally use in this course.
## Variables

### C++ Integers

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Most Positive</th>
<th>Most Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>short</code></td>
<td>16</td>
<td><code>(2^{16} - 1)</code></td>
<td><code>-2^{16}</code></td>
</tr>
<tr>
<td><code>signed short</code></td>
<td>16</td>
<td><code>32,767</code></td>
<td><code>-32,768</code></td>
</tr>
<tr>
<td><code>int</code></td>
<td>32</td>
<td><code>(2^{31} - 1)</code></td>
<td><code>-2^{31}</code></td>
</tr>
<tr>
<td><code>long</code></td>
<td>32</td>
<td><code>2,147,483,647</code></td>
<td><code>-2,147,483,648</code></td>
</tr>
<tr>
<td><code>long long</code></td>
<td>64*</td>
<td><code>(2^{63} - 1)</code></td>
<td><code>-2^{63}</code></td>
</tr>
<tr>
<td><code>unsigned short int</code></td>
<td>16</td>
<td><code>(2^{16} - 1)</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>unsigned short</code></td>
<td>16</td>
<td><code>65,535</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td>32</td>
<td><code>(2^{32} - 1)</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>unsigned long</code></td>
<td>32</td>
<td><code>4,294,967,295</code></td>
<td><code>0</code></td>
</tr>
<tr>
<td><code>unsigned long long</code></td>
<td>64*</td>
<td><code>(2^{64} - 1)</code></td>
<td><code>0</code></td>
</tr>
</tbody>
</table>
To declare an integer variable called `a` in C++, one writes:

```c
int a;
```

```c
int b = 10;
```

Both these statements reserve a region of memory (just big enough to hold an int variable) and attaches a label to it...the variable name `a` or `b`. The second statement both declares the variable `b` and initializes it to 10.
Variables which represent floating point numbers. These also come in 3 types:

- `float`
- `double`
- `long double`

For scientific computing the double type is of most use and this is what you should normally use in this course.
Variables

Variables which can store a (single) ASCII character constant

```
char
```

Characters are stored as numbers (bit patterns) using ASCII codes:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
<tr>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
</tr>
<tr>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
<tr>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
</tr>
<tr>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
<tr>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>`</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
</tr>
<tr>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>
Variables

How much memory is needed to store a variable of a given type?

To find out how much memory (in bytes) is used for a particular variable type, a special operator `sizeof()` of a variable or variable type. e.g.

```
cout << sizeof( int ) << endl;
```

prints the number of bytes used by the int type. The program `sizes.cpp` does this for most of the basic types.
What range of numbers can be stored in each variable type?

The maximum and minimum size of numbers which can be stored in variables is available in the form of pre-defined constants:

In the standard header file `<climits>` defines the constants specifying the range of the integer types:

```
SHRT_MIN, SHRT_MAX, INT_MIN, INT_MAX, LONG_MIN, LONG_MAX
```

Whilst the corresponding values for the floating point types may be found in `<cfloat>`:

```
FLT_MIN, FLT_MAX, DBL_MIN, DBL_MAX, LDBL_MIN, LDBL_MAX
```

NB Some other useful maths constants can be found in `<cfloat>` e.g.

```
M_PI
M_E
```

Always use the compiler supplied constants such as `M_PI` when you need to have to access to a number such as π.
Variables

**Literal Constants**

Integer numbers (Literals) are, by default of int type.

```cpp
int i = 123456;
```

The prefix `0x` denotes an hexadecimal (base 16) constant:

```cpp
int i = 0xFFFF;
```

Floating point literals are, by default, of type double. They may be in scientific notation:

```cpp
double e = 1.602e-19;
```
Character and Character String Constants

(Single) character constants are indicated by single quotes e.g.

```
char a = 'a';
```

Character strings (>1 characters) are indicated by double quotes e.g.

```
"Hello World"
```

The storage and manipulation of character strings is a more difficult topic that will be discussed further later.
**Constant Variables**

If you use a variable to represent a constant, it is a good idea to declare it `const`. This ensures the constant is not accidentally changed:

```cpp
const double twopi = 2*M_PI;
```
Basic Arithmetic Operators

We have already come across several C++ operators; these are (mostly) simple tokens of one or more characters which represent the basic build-in operations between variables of the basic types.

If one writes the statement:

\[ a = b + c; \]

\( a \), \( b \) and \( c \) are variables whilst \( = \) and \( + \) are operators.
The Assignment Operator

Perhaps the most fundamental operator is the assignment operator =. Consider the following block of code:

```c
a = 10;
b = 0;
b = a;
```

It is important to understand that in C++ (as well as Java, Perl, Fortran and most other programming languages)

The final statement means "copy the data held at the memory location represented by a to that represented by b (and leave a unchanged)"

It has nothing to do with equality as in a maths equation!
Example Program

// Description: Declare and use some variables

#include <iostream>
using namespace std;

int main()
{
    int i=10, j=20;
    double x=1.0;
    int k;
    k = i + j;
    double y = x / 3.0;
    cout << "k = " << k << endl;
    cout << "y = " << y << endl;
    return 0;
}
# The Basic Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Evaluated</th>
<th>How?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>parentheses</td>
<td>1</td>
<td>innermost pair first</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>2</td>
<td>from left to right</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>3</td>
<td>from left to right</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
<td>last</td>
<td></td>
</tr>
</tbody>
</table>
The operators:  +  -  *  /

represent the usual mathematical operations of addition, subtraction, multiplication and division. These work with all the built in integer and floating point variable types. But N.B.:

```cpp
int    i = 21;
int    j =  5;
int    k = i / j;
double r = i / j;
```

Will result in 4 being assigned to k and 4.0 being assigned to r.

Because integer division arithmetic is truncated (NOT rounded!)
Integer Division

This also applies to constant expressions as well as variables. For example, if one writes:

```c
double r = 1/3;
```

*r* will be assigned the value 0, which is unlikely to be what is intended! Instead one should force the compiler to use floating point numbers e.g.

```c
double r = 1.0/3.0;
```

Indeed there is a second operator % which results in the Modulus (remainder)

```c
int a = 21 % 5;
```

will result in *a* being assigned the value 1.
Operator Precedence

The *, / and % operators have a higher precedence than the + and - operators.

This means that

```c
int a = 3;
int b = 4;
int c = 5;
int r = a * b + c;
```

is evaluated as \((a \times b) + c\) i.e. \(r\) is assigned the value 17.

If you require the addition to be done first, then you must explicitly use parentheses (round brackets) in your program:

```c
int r = a * (b + c);
```
Arithmetic Operators

Precedence

\[ y = a + b \times c; \]

\[ y = (a + b) \times c; \]
Arithmetic Operators

$y = ((a + b) + c) \times d$;
If you are in doubt about operator precedence explicit parentheses ( ) can always be used.
The expression

```
int r = a / b * c;
```

is evaluated as

```
(a / b) * c
```

**NOT**

```
a / (b * c),
```

when arithmetic operators have equal precedence, the precedence is left-to-right.
Arithmetic Operators

Precedence

\[ y = a / b * c; \]

\[ y = a / (b * c); \]
C++ has a number of built-in functions which extend the language to provide features beyond the basic operators.

These include a number of the maths functions which provide various mathematical operations. All of these are defined in the header file `<cmath>` which must be included in a program for them to be used. These functions include the following:

- **sqrt(x)**: Square root of x
- **fabs(x)**: Absolute value of x
Maths Functions

\[\sin(x) \quad \text{Sine of } x \text{ (in radians)}\]

\[\cos(x) \quad \text{Cosine of } x \text{ (in radians)}\]

\[\tan(x) \quad \text{Tangent of } x \text{ (in radians)}\]

\[\text{asin}(x) \quad \text{Inverse sine with result between } +/- \pi\]

\[\text{acos}(x) \quad \text{Inverse cosine with result between } 0-\pi\]

\[\text{atan}(x) \quad \text{Inverse tangent with result between } +/- \pi\]

N.B. All the Trig functions expect their arguments in radians.
Maths Functions

- $\log(x)$: Natural logarithm of $x$
- $\log_{10}(x)$: Base 10 logarithm of $x$
- $\exp(x)$: $e$ to the power of $x$
- $\sinh(x)$, $\cosh(x)$, $\tanh(x)$: Hyperbolic functions
- $\text{asinh}(x)$, $\text{acosh}(x)$, $\text{atanh}(x)$: Inverse Hyperbolic functions
Maths Functions

All these functions take a single double (or float) argument. They may used in a similar way to variables, but NEVER on the left-hand side of an assignment. e.g

```cpp
#include <cmath>
...
double x = 2.0;
cout << " sqrt 2 = " << sqrt(x) << endl;
```

N.B. All the Trig functions expect their arguments in radians.
**Maths Functions**

**Functions with 2 arguments**

Other functions are more complicated and take more than one argument e.g.:

- **pow(x,y)**  
  Raise x to the power of y

- **atan2(y,x)**  
  Inverse tangent of y/x with result between $+/- \pi$
  Compare and contrast with atan(y/x)

These are used in a similar way, except two arguments must be supplied. For example:

```cpp
#include <cmath>
...
double f = pow(10.0, 3.0);
```

results in f being assigned 1000.0.
**More complicated arguments**

In all the examples so far the function arguments have been simple variables or constants. In fact the argument to a function can be any expression which evaluates to a variable of the appropriate type.

Suppose one wants to calculate

Instead of writing:

```cpp
#include <cmath>
...
double   y = pow( x, 3.0 );
double   f = sin( y );
```

One can replace the calculation be a single statement:

```cpp
#include <cmath>
...
double   f = sin( pow(x, 3.0) );
```