LECTURE 1: Introduction

C++ at what level?

- typical quants spend significant portion of their time with coding
- ”core quants” implement quantitative libraries, optimise code, they know the hard core stuff
- non-core quants are users of the quantitative libraries, they must understand what these libraries do, but how they do it is not necessarily interesting

What is this course about?

Objective: learn what is required for non-core quants, in particular

- Reading and understanding C++ code, structure, built-in types, control flow
- Designing and implementing classes, object-base programming
- Template functions, template classes
- Standard template library, containers
- Object oriented principles, inheritance
- Composing objects, a bit of design patterns

Brief history of C++

- Bell Labs: UNIX, B, C
- Bell Labs, Bjarne Stroustrup: extended C to ”C with classes”
- New features
  - classes and object oriented features
  - exceptions
  - templates
  - namespaces
  - STL
- C++ aimed to be standardized and portable
- Extensions: Boost
- Most recent standards: C++11 – about its new features . . . see later

Platforms and environments:

- Unix, Linux
- Windows (e.g. MS Visual Studio)
- Mac (e.g. Xcode)
- Eclipse, CodeBlocks etc. available on all of these platforms.

Some of the new features

- Container improvements, new algorithms
- Smart pointers
- Random number generators and distributions
- Threads, Locks
- Lambdas
- Range-for statement
- Rvalue references, move semantics
- Initializer lists
- `auto`, `decltype` - deduction of a type from an initializer
- `static_assert` - static (compile-time) assertions
- `long long` at least 64bit long integers
- `operator ""` - user-defined literals
- `override`, `final` - override controls

A piece of simple C++ code (`Lecture01/Lecture01.cpp`):

```cpp
#include <iostream>

int main()
{
   //This program plots some text
  std::cout << "Technically, this is the \"Hello" <<
            " world!\" program." << std::endl;

  return 0;
}
```

The output is:

```
Technically, this is the "Hello world!" program.
```

```cpp
#include <iostream>
```

- The `#include` directive is to be used when asking for facilities not included in the **core language**.
- `iostream` is part of the **standard-library**, the basic package for displaying values.
- we will use several other **header files** from the standard library
- we will create many of our own header files (`*.h`, `*.hpp`, etc.)

```
1 int main()
2 {
3 }
```

- Declaration of a function has a strict form:
  `return_type func_name(arg1_type arg1_name, ...)`
- The function body goes between curly braces (**block**).
- Normally the declaration and the implementation (definition) are separated.
- The `main` function can have arguments and can be called from console command line with arguments.
- There is a version of `main` taking arguments $\longrightarrow$ one can pass arguments to and execute applications written in `c++` from command line.

```
1 //This program plots some text
```

- The part of the line after `//` is a comment and not part of the code.
- Alternatively, multiple lines can be commented using the opening `/*` and closing tag `*/`, e.g:

```
1 std::cout << std::endl; /* Comment starts here
2 and this is also part of the comment
3 until the closing tag. */
```

```
1 std::cout << "Technically, this is the \"Hello" <<
2           " world!\" program." << std::endl;
```

- **Namespaces** are used to separate pieces of code into different domains (or **scope**).
- A namespace can spread over several (header) files, and one (header) file may contain several namespaces. Namespaces can be nested.
- The standard-library is placed in the namespace `std`.
- One can access the facilities in `std`, by using the *address* `std::` (`::` is the *scope*-operator).
- `cout` is the command to display values (string, int, double, etc.).
- `<<` is the operator which inserts values on its right into the stream on its left (`<<` is left associative).
- `std::endl` is the command inserting the end-of-line into the output stream.
- Statements can be broken into several lines (a line break is just a space), each statement must end with "`;`".
- The string inserted into the stream starts and ends with `"`, however the character after `\` is regarded as part of the text.

```
1 return 0;
```

- If a function does not return value, its return type is `void` (except constructors, and destructors, see later)
- If a function has a non-`void` return type, it should explicitly return a value of that type using the `return` keyword
- Any command after the `return ...;` part is ignored by the compiler (unless it's one branch of an if statement and the other branch is implemented after that)
- The `main` returns 0 if the execution is successful, otherwise a non-zero return indicates a failure.

How to get to an executable?

> *"... please remember, the compiler is your friend; possibly, the compiler is the best friend you have when you program."*
> Bjarne Stroustrup

- After the code is typed in, it must be compiled.
- The **compiler** translates the human-readable **source code** into **object code**.
- The files containing source code have extension `cpp`, `h`, `hpp`, the object code is saved in files with extension `obj` (in Windows) or `o` (in Unix, Linux)
- The compiler checks syntax and many things, and compilation might result in error messages.
- The compiler might change the (object) code to something equivalent and does lots of optimization.
- Some of the most popular C++ compilers: Visual C++, GCC, Intel, etc.
- The pieces of the object code are linked together by the **linker** and an executable file is created.
- The source code is portable between different platforms, the object code and the executable are not.
- Errors could be: **compile-time**, **link-time** or **run-time-errors**.

- In C++, variables must be **declared** first.
- Variables can be **initialised** when defined, but this could also be done later.
- These variables are **local** and will be **destroyed** at the end of the scope (i.e. at the brace "}" closing the brace "{" before the definition, excluding the braces denoting nested scopes)
- The definition must contain the **type** of the variable.
- There are built-in types in the core language:
  - `int`, `short int`, `long int`, `unsigned int`, `unsigned short int`, etc.
  - `float`, `double`, `long double`
  - `char`, `unsigned char`, `bool`
  - `enum` (see later)
- The standard library contains lots of types (e.g. see `#include<string>`).
- One can implement his/her own types, i.e. classes (see later).
- Variables can be defined to be `const`, e.g.: `const double tiny=1e-21;`
- The value of `const` variables **must be initialised at declaration** and won't change during its life time. If at some point in the code, one tries to modify the value of a `const` variable, the compiler will give an error message.
- `cin` reads from console into variables.
- The operator >> is used for reading in values.
- If the typed in value has a type different from what is expected, then it is converted. For example, `double` is converted to `int` with losing some information:

A program with input

```
1  std::cout << "Please tell me your first " <<
2    "name and your age." << std::endl;
3
4  std::string sFirstName;
5  int iAge;
6
7  std::cin >> sFirstName;
8  std::cin >> iAge;
9  std::cout << "Hi, " << sFirstName << "! " ;
10
11 if( iAge<30 )
12   std::cout<< "Five years from now, " <<
13     "you'll be only " << iAge+5 << ".\n";
14 else
15   std::cout << "No offense, but you're old. "
16       "Five years from now, you'll be "
17        << iAge+5 << "." << std::endl;
```

The output of the previous code:

```
Please tell me your first name and your age.
Greg
33
Hi, Greg! No offense, but you're old. Five years
from now, you'll be 38.
```

```
1 std::string sFirstName;
2 int iAge;
```

```
1 std::cin >> sFirstName;
2 std::cin >> iAge;
```

More on programs with input. You can also relate to the following examples as they are better explained in the videos. The notes are just a summary but the videos are fully expanded with several examples and explanations.

THE IF AND IF-ELSE STATEMENTS

```
1 if( CONDITION_TRUE ) {
2   ...
3 }
4 else if {
5   ...
6 }
7 else {
8   ...
9 }
```

- An `if` block uses the keywords `if`, `else if`, `else`
- The condition is to be written between " (" and ") "
- The statements of each branch form a scope, i.e. variables defined there will be local, i.e. restricted to their scope.
- One can skip the braces in case of single statement scopes.

Summary

`#include` directive	`iostream`
namespaces	standard library, `std`
variable declaration	variable initialisation
types	built-in types
`std::cout<<`	`std::cin>>`
`if then else`	`return`
function declaration	function definition

Control flow and operations

```
1  #include <iostream>
2  #include <iomanip>
3  #include <vector>
4  #include <algorithm>
5  void Example2()
6  {
7    double dMark;
8    std::vector<double> vMarks;
9    std::cout << "Please, type in your marks." << std::endl;
10
11   while(std::cin >> dMark)
12     vMarks.push_back(dMark);
13   double dAverage(0);
14   std::vector<double>::size_type MarksSize=vMarks.size(), i;
15
16   for(i=0; i<MarksSize; ++i)
17     dAverage+=vMarks[i];
18
19   dAverage/=MarksSize;
20
21   std::sort(vMarks.begin(), vMarks.end());
22   double median=vMarks[MarksSize/2];
23
24   std::cout<< "The median is " << median <<
25     " and the average is: " << std::setprecision(4)
26       << dAverage << std::endl;
27 }
```

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
```

- `iomanip` contains `setprecision`
- `vector` contains the implementation of the standard-library's vector container
- `algorithm` is a collection of routines for manipulating standard-library style containers

```
while(std::cin >> dMark)
  vMarks.push_back(dMark);
```

In general:

```
while( CONDITION_TO_CONTINUE ){
  // ...
}
```

- The `while` loop can be used to execute a set of statements repeatedly as long as the condition-to-continue is true
- The set of statements forms a scope, i.e. anything declared within the scope is destroyed when it goes out of scope
- In case of single statement while scopes, the braces can be omitted.
- Use the `break;` statement to terminate the execution of the nearest enclosing loop.
- Use the `continue;` statement to jump to the beginning of the next iteration (including the test) in the nearest enclosing `while` statement.

```
for(i=0; i<MarksSize; ++i)
  dAverage+=vMarks[i];
```

In general:

```
for(INIT_LOOP_VARIABLES; CONDITION_TO_CONTINUE; UPDATE_LOOP_VARIABLES)
{
  // ...
}
```

- The `for` loop requires
  - a loop variant,
  - a condition to continue and
  - a statement updating the loop variant

- The set of statements forms a scope, i.e. anything declared within the scope is destroyed when it goes out of scope
- In case of single statement `for` scopes, the braces can be omitted.
- Use the `break` statement to terminate the execution of the nearest enclosing loop.
- Use the `continue;` statement to jump to the beginning of the next iteration in the nearest enclosing `for` statement. The next iteration includes the "update-loop-variant" expression as well.

```
1   vMarks.push_back(dMark);
2 std::vector<double>::size_type MarksSize=vMarks.size(), i;
3   dAverage+=vMarks[i];
```

- `std::vector` is a templated container, it can contain elements of any type (if copyable and assignable)
- The most important vector operations are
  - `vector<T> v;` - defines an empty vector to hold elements of type `T`
  - `vector<T> v(n);` - defines a vector of size `n`
  - `v.push_back(value)` - inserts a new element at the end of the vector
  - `v.size()` - returns the number of entries
  - `v.resize(n);` - sets the size to `n`, **note: resizing of vectors is possible**, it's also possible to reset the capacity.
  - `v[i]` - accesses the `i`th entry, **note: indexing starts at 0**
  - `v.begin()` - returns an **iterator** to (the address of) the first entry
  - `v.end()` - returns an iterator to the memory place one past the last element (`push_back` will write here)
- Vectors allow random access of elements, vectors are quick at inserting/erasing elements at the end,
- On the other hand, vectors are inefficient at removing/inserting elements other than at the end.
- We will see more about `std::vector`'s in lecture 5.

```
1 std::sort(vMarks.begin(), vMarks.end());
```

- `sort` is a standard-library algorithm.
- It takes a range and does efficient sorting.
- Works on many container types, however some of the containers have their own sorting algorithms (possibly even more efficient).

Operators

- `a=b` assignment
- `a=b+c`, `b-c`, `b*c`, `b/c`, `b%c` (modulo)
- `++i`, `i++` pre-increment and post-increment, with analogy: `--i`, `i--`
- `a+=b` is equivalent to `a=a+b`, with analogy: `a-=b`, `a*=b`, `a/=b`
- `!a` logical negation
- `(a && b)`, `(a || b)` logical "and" and "or" (return `bool`)
- `a==b`, `a!=b`, `a>b`, `a>=b`, `a<b`, `a<=b` relation operators (return `bool`)
- `(a ? b : z)` yields `b` if `a` is true, `z` otherwise.
- `vec[i]` accesses the `i`th entry of the container `vec`
- `obj.prop` accesses the member named `prop` of the object `obj`
- `obj.mfun(...)` calls the member function named `mfun` of the object `obj`

The action of the operator on the simple types is straightforward. What do they do on user defined types? - Whatever the user makes them do!

Control flow - switch

```
1  void Example3()
2  {
3    enum Month{jan=1, feb, mar, apr, may, jun,
4          jul, aug, sep, oct, nov, dec};
5    std::cout << "Please type in the current " <<
6       "month (in form of an integer)!" << std::endl;
7    int iInput;
8    std::cin >> iInput;
9    Month mInput=Month(iInput);
10
11   switch(mInput){
12     case jan: case mar: case may: case jul:
13     case aug: case oct: case dec:
14       std::cout<< "There are 31 days in this month."<< std::endl;
15       break;
16     case apr: case jun: case sep: case nov:
17       std::cout<< "There are 30 days in this month."<< std::endl;
18       break;
19     case feb:
20       std::cout << "There are 28 days in this month, " <<
21     "unless it's a leap year." << std::endl;
22       break;
23     default:
24       std::cout << "That's not a month." << std::endl;
25       break;
26   }
27 }
```

```
enum Month{jan=1, feb, mar, apr, may, jun,
      jul, aug, sep, oct, nov, dec};
```

Examples:

```
Month m=feb;
// ...
if(m==feb)
  //...
```

- Use `enum` when types with names constants are required.
- Enumeration by default assigns consecutive integers starting at 0 to labels.
- One can specify different starting value, or even non-consecutive integer values.
- In general, avoid hard coded constants – *magic numbers*.

```
switch(mInput){
 case jan: case mar: case may: case jul:
 case aug: case oct: case dec:
   std::cout<< "There are 31 days in this month."<< std::endl;
   break;
 case apr: case jun: case sep: case nov:
   std::cout<< "There are 30 days in this month."<< std::endl;
   break;
 case feb:
   std::cout << "There are 28 days in this month, " <<
 "unless it's a leap year." << std::endl;
   break;
 default:
   std::cout << "That's not a month." << std::endl;
   break;
}
```

- The `switch` statement evaluates an expression and jumps to the matching `case` label
- Use the `break;` statement at the end of the block, otherwise execution will follow from one label to the next.
- The value on which `switch` switches must be integer, `char` or `enum`. Other types (`string`, `double` etc.) are not accepted.
- The value in the `case` labels must be constant expression.

Functions

A reminder of what has been said about functions.

- Functions must be declared before use.
- The declaration is of the form:

```
type_name name(arg1_type, arg2_type, ...);
```

- Any statement of the same syntax structure is regarded as a function declaration.
- If a function does not return value, its return type must be `void`.
- The implementation of a function can be placed after the `main`, or even in different `cpp` file.
- Each function is determined by its name and the list of argument types. I.e. functions can share their name if their argument lists are different. During execution it will be recognized which one to call. E.g.:

```
void MyMax(int, int );
void MyMax(std::vector<double>);
```

Functions - passing arguments

```
void swap1(int a, int b)
{
   int temp=a;
   a=b;
   b=temp;
}
void Example4()
{
  header(4);
  int a(1), b(2);
  std::cout<< "Before calling swap1 on a and b," << std::endl;
  std::cout<< "a: " << a << " b: " << b << std::endl;
  swap1(a,b);
  std::cout<< "After calling swap1 on a and b," << std::endl;
  std::cout<< "a: " << a << " b: " << b << std::endl;
    //...
}
```

```
Before calling swap1 on a and b,
a: 1 b: 2
After calling swap1 on a and b,
a: 1 b: 2
```

OK, what has just happened here???

Functions - passing arguments problem explained

```
1 void swap1(int a, int b)
2 {
3    int temp=a;
4    a=b;
5    b=temp;
6 }
```

- The function `Swap1` in fact takes two arguments.
- However it does not work on the variables `a` and `b`, but it works on copies of them. I.e. **the variables themselves are not passed to the function, they are NOT in the rabbit**.
- This strategy is referred to as **pass-by-value**.
- The variables in the function body are local variables, created in the scope and destroyed when leaving the scope.
- Pass-by-value is useful to pass objects of small size to a function if only their values are needed, especially if the original objects are not to be modified.

Functions - passing arguments problem fixed

```
1 void swap2(int & a, int & b)
2 {
3    int temp=a;
4    a=b;
5    b=temp;
6 }
```

- The symbol "`&`" indicates that the variables are **passed by reference**.
- In this case, no copies of the original variables are created, but the function directly accesses them.
- The operations in the function body will act on the original variables, changing their values.
- Pass-by-reference can be used to avoid copying objects of significant size.
- An alternative solution is to use pointers. See later.

Functions - passing arguments by (const) reference

```
1  void AddVector(const vec_type & a, const vec_type & b, vec_type & c)
2  {
3    vec_type::size_type n=std::min(a.size(),b.size());
4    c.resize(n);
5    for(vec_type::size_type i=0; i<n; ++i)
6      c[i]=a[i]+b[i];
7  }
```

- Further possibility: use **pass-by-const-reference** if one wants to avoid copying and wants to protect the argument from modification.
- In this example, `a` and `b` are passed by const reference, `c` is passed by reference
- No copies of `a` or `b` are created, saving memory and time
- The `const` keyword protects `a` and `b` from modification (the compiler would give an error if any modification was attempted).
- The function works on the original `c`, the entries of `c` will be modified (that's fine, `c` is passed by non-const reference).
- Note: the function explicitly returns `void`, however, implicitly, it returns `c`. This strategy avoids copying the returned value into some objects, again saving time and memory.

Some useful functions

Use `#include <cmath>` to access the following functions

- trigonometric functions: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`
- hyperbolic functions: `cosh`, `sinh`, `tanh`
- exponential/logarithmic functions: `exp`, `log`, `log10`, `modf`, `frexp`, `ldexp`
- power functions: `pow`, `sqrt`
- rounding, absolute value, remainder: `ceil`, `floor`, `fabs`, `fmod`

Use `#include <cmath>` to access the following functions

- trigonometric functions: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`
- hyperbolic functions: `cosh`, `sinh`, `tanh`
- exponential/logarithmic functions: `exp`, `log`, `log10`, `modf`, `frexp`, `ldexp`
- power functions: `pow`, `sqrt`
- rounding, absolute value, remainder: `ceil`, `floor`, `fabs`, `fmod`

Summary

`#include<iomanip>`	`std::setprecision()`
`while`	`for`
`#include<vector>`	`std::vector<T>`, `resize()`, etc.
`#include<algorithm>`	`sort()`
=, +, -, *, /, %	++, --, +=, -=, etc.
`enum`	`switch`, `case`
function signature	function scope
pass by value	pass by (const) reference
`#include<cmath>`	