

Basics of C++ and OpenFOAM

Tommaso Lucchini



Department of Energy
Politecnico di Milano

Outline

- Why C++?
- How to learn C++ efficiently
- Basics of C++ and examples of C++ implementation in OpenFOAM
- Some C++ bibliography

Why C++?

- Programming language serves two related purposes:
 - ▶ **A Vehicle** to specify actions to be executed.
 - ▶ **Provide a set of concepts** for the programmer to use when thinking about what can be done.
- For these reasons it has to be:
 - ▶ **Close to the machine** to handle simply and efficiently its aspects.
 - ▶ **Close to the problem to be solved** so that the concept of a solution can be expressed directly and concisely.
- There is a **very close connection** between the language **we think/program** and the problems and solutions **we can imagine**.

Learning C++ (1/2)

- It is important to **focus** on the **concepts** and not get lost in language-technical details.
- The purpose of learning a programming language is to become a better programmer:
 - ▶ To become more effective at designing and implementing new system and at maintaining old ones.
- Appreciating programming and design techniques is far more important than understanding details; that understanding will come with time and practice.
- However, this takes time to people coming from different languages (C, Fortran, ...).

Learning C++ (2/2)

- Applying techniques effective in one language to another typically leads to awkward, poorly performing, and hard-to-maintain code.
- Ideas must be transformed into something that fits with the general structure and type system of C++ in order to be effective in the different context.
- C++ supports a gradual approach to learning, and it can be easy and fast, but not as much easier and faster as most people would like it to be. It makes possible to learn the concepts in a roughly linear order, and gain practical benefits along the way.
- C++ can be learned without knowing C. C++ is safer and reduce the need to focus on low-level techniques.

The design of C++

- C++ was designed with **simplicity** in mind.
- C++ has no built-in high-level data types and no high-level primitive operations:
 - ▶ It does not provide a matrix type with operations
 - ▶ It does not provide a string type with a concatenation operator
- If a user wants such types, they can be defined in the language itself.
- Defining a **new, general-purpose or application-specific type** is the most fundamental programming activity in C++.

C++ basics - Declarations, I/O, operations

- Variable declaration - data of different types

```
int myInteger = 10;
```

```
const int myInteger = 10; //constant variable
```

In C++ it is possible to define special variable types.

- `iostream` (standard library) is used for input and output

```
cout << "Please type your age" << endl;
```

```
cin << myInteger;
```

`<<` and `>>` are input-output operators, and `endl` is a manipulator that generates a new line. OpenFOAM uses the `Info` output stream which is more robust with parallel simulations.

- Variables can be added, subtracted, multiplied and divided as long as they have the same type, or if the types have definitions on how to convert between the types. User-defined types must have the required conversions defined. Some of the types in OpenFOAM can be used together in arithmetic expressions, but not all of them.

C++ basics - Operators, Math library, if-statements, while statements

- `+`, `-`, `*`, `/`: Operators that define how the operands should be used. Other operators are `%` (integer division modules), `++` (add 1), `--` (subtract 1), `+=` (`i+=2`, adds 2 to `i`), `-=`, `*=`, `/=`, etc. User-defined types should also define their operators.
- Mathematic standard functions (trigonometry, logarithmic,...) are not part of the C++ itself, but they are in the standard library. Use `#include <cmath>`. For other functions (arithmetic), `#include <cstdlib>`.
- if statements: `(if var1 > var2) {...code...} else {...code...}`
 - ▶ Comparing operators: `<` `>` `<=` `>=` `==` `!=`
 - ▶ Logical operators: `&&` `||` `!`
 - ▶ Generates `bool`
- while statements `while(statement) {... code ...}`
- `break`, breaks the execution of a `while`

C++ basics - for statements, array, array templates

- for-statements: `for(init; condition; change) {...CODE...}`
- Arrays:
 - ▶ `double f[5];` (**Note:**components numbered from 0!)
 - ▶ `f[3] = 2.5;` (**Note:**no index control!)
 - ▶ `int a[6] = {2, 1, 4, 5, 3, 9};` (declaration and initialization)
 - ▶ Arrays have strong limitations, but are a good basis for array **templates**.
- Array templates (for example `vector`, `list`,...):
`#include <vector>`
`using namespace std`
 - ▶ `vector<double> v(3);` gives `{0, 0, 0}`
 - ▶ `vector v2(4, 1.5);` gives `{1.5, 1.5, 1.5, 1.5}`
 - ▶ `vector v3(v2);` copies `v2` to `v3`
- Array templates operations: the template classes define member functions that can be used for those types, for instance: `size()`, `empty()`, `assign()`, `push_back()`, `clear()`, *etc*
`v.assign(4, 1);` gives `{1.0, 1.0, 1.0, 1.0}`

C++ basics - functions

- Functions may or may not return a value
- Example function: `average`

```
double average(double x1, double x2)
{
    int nValues = 2;
    return (x1 + x2)/nValues;
}
```

This function takes two arguments of type `double` and return a `double`. `nValues` is a local variable and is not seen outside the code. Everything written after the `return` instruction will not be executed by the code.

- A function does not have to take arguments, and it does not have to return anything. The `main` function should return an integer, but the `return` statement can be skipped in the `main` function.
- There may be several functions with the same name, as long as there is a difference in the arguments to the functions (the number of arguments or the type of the arguments).

C++ basics - Scope of variable

- The scope and visibility of a variable depends on where it is defined.
 - ▶ A variable defined in a block between ({ }) is visible in that block.
 - ▶ A variable defined in a function head is visible in the entire function.
 - ▶ There can be several variable with the same name, but only one in each block.

Example:

```
int x;
```

```
char f1(char c)
{
    double y;
    while(y > 0)
    {
        char x;
    }
    int w;
}
```

C++ basics - Variables definition and functions

- Variables and functions must be *declared* before they can be used.

```
double average(double x1, double x2) // function head
main()
{
    double a = 3;
    double b = 7.6;
    double mv = average(a, b);
}
double average(double x1, double x2) // function definition
{
    return (x1+x2)/2.0;
}
```

The argument names may be omitted in declaration.

- Declarations are often included from include-files (`#include "file.h"` or `#include <standardfile>`)
- Use separate files for function declaration and definition (`file.H` and `file.C`)

C++ basics - Function parameters/arguments

- Pass by reference if you want to change the argument value within the function, example:

```
▶ void change(double& x1);
```

The reference parameter `x1` will not be a local variable but a reference to the argument of the function.

- Passing argument by reference will reduce the memory charge especially with large objects. Use `const` reference to avoid the arguments to be changed in the function. Example:

```
▶ int size(const& string);
```

- Default arguments can be specified, and this avoid to specify that argument with the function is called:

```
double integrate(const double& x, const double& y, int x = 1);  
integrate(x, y);  
integrate(x, y, 5);
```

C++ basics - Types

- *Types* define what values a variable may obtain, and which operations can be performed on that variable.
- Pre-defined C++ types:
 - ▶ signed char
 - ▶ short int
 - ▶ int
 - ▶ unsigned char
 - ▶ unsigned short int
 - ▶ unsigned int
 - ▶ unsigned long int
 - ▶ float
 - ▶ hdouble
 - ▶ long double
- **Classes** are the user-defined types. In OpenFOAM there is a wide number of classes defined.

C++ basics - Pointers

- Declaration of a pointer:

```
char* p; // pointer to a character
```

- In declarations, * means "pointer to".
- A pointer variable can hold the the address of an object of the appropriate type:

```
p = &v[8]; // p points to the v's eight element
```

- Unary & is the address of operator

C++ basics - Typedef (1/2)

- To simplify the variable declaration in C++. By using `typedef` the variable type can be used with a new name:

```
typedef vector<double> doubleVector;
```

In this way:

```
vector<double> c(8);
```

is equivalent to:

```
doubleVector c(8);
```

- Typedefs make the code easy to read. OpenFOAM uses them a lot!

C++ basics - Object orientation

- Object orientation focuses on the objects instead of functions
- An *object* belongs to a *class* of objects with the same attributes. The class defines the construction of the object, destruction of the object, attributes to the object and the functions that can manipulate the object
- The objects may be related in different ways and the classes may inherit other attributes from other classes.
- Classes can be re-used, and each class can be designed or bug-fixed for a specific task.
- In C++, a *class* is a *user-defined type*.
- In OpenFOAM, classes are used to define, discretize and solve PDE systems.

C++ basics - Class definition

- The class `Name` and its public and hidden member functions and data are defined as follows:

```
class myclass:{  
public:  
    declaration of public member functions and data members  
private:  
    declaration of hidden member functions and data members  
};
```

- `public` attributes are visible from outside the class
- `private` attributes are visible **only** within class
- If neither `public` or `private` are defined, everything will be considered `private`
- Member functions and data member are declared as usual

C++ basics - Using a class

- An object of class `myClass` is defined in the main code as:

```
myClass data1; (like int i;)
```

- `data1` will have all the attributes defined in class `myClass`.
- Any number of objects may belong to a class, and the attributes of each object will be separated.
- In any class, there may be pointers and reference to other objects.
- The member functions will operate according to their implementation in the class. If there is a member function `write` that writes out the content of an object of the class `myClass`, it is called in the main code as:

```
data1.write();
```

- Pointer, references and member functions:

```
myClass* p1 = &data1  
p1->write();  
myClass& p2(data1);  
p2.write();
```

C++ basics - Member functions

- The member functions may be implemented in the class definition or outside. The syntax is:

```
inline void myClass::write()  
{  
  // some code..  
}
```

where `myClass::` tells us that the member function `write` belongs to the class `myClass`. `void` means that the function does not return any value, and `inline` tells us that the function will be *inlined* into the code, where it is called instead of jumping to the memory location of the function at each call (good for small functions). Member functions defined directly in the class definition will automatically be inlined when possible.

- The member functions have direct access to all the data member and all the member functions of the class.

C++ basics - Organization of the class

- Make the class files in pairs, one with the definitions and the other with the function declarations.
- Classes that are closely related to each other can share files, but keep the class definitions and functions declarations separate. This is done in OpenFOAM
- The class definitions must be `included` in the object file that will use the class, and in the function declarations file. The object file from the compilation of declaration file is statically or dynamically linked to the executable by the computer.
- Inline functions must be declared in the definition file since the compiler does not look for them in the declaration file. For example in OpenFOAM there is the `VectorI.H` file containing inline functions, and those files are included in the corresponding `Vector.H` file.
- Examples: `src/OpenFOAM/primitives/Vector.H`

C++ basics - Constructors (1/2)

- A constructor is a special initialization function that is called each time a new object of that class is defined. Without a specific constructor, all attributes will be undefined. A null constructor must be always defined.
- A constructor can be used to initialize the attributes of the object. A constructor is recognized by it having the same name of the class (here `Vector`. `Cmpt` is the typedef for component type. i.e.: the `Vector` class works for all component types):

```
// Constructors
  //- Construct null
  inline Vector();
  //- Construct given VectorSpace
  inline Vector(const VectorSpace<Vector<Cmpt>, Cmpt, 3>&);
  //- Construct given three components
  inline Vector(const Cmpt& vx, const Cmpt& vy, const Cmpt& vz);
  //- Construct from Istream
  inline Vector(Istream&);
```

- The `Vector` object will be initialized differently depending on which of these constructors will be chosen.

C++ basics - Constructors (2/2)

- A copy constructor has a parameter that is a reference to another object of the same class (`class myClass(const myClass&);`). The copy constructor copies all the attributes. A copy constructor can only be used when initializing an object. Usually there is no need to define a copy constructor since the default one does what is needed.
- A type conversion constructor is a constructor that takes a single parameter of a different class than the current class, and it describes explicitly how to convert between the two classes.

C++ basics - Destructors

- When using dynamically allocated memory it is important to be able to destruct an object.
- A destructor is a member function without parameters, with the same name of the class, but with a `~` in front of it.
- An object should be destructed when leaving the block it was constructed in, or if it was allocated with `new` it should be deleted with `delete`.
- To make sure that all the memory is returned it is preferable to define the destructor explicitly.

C++ basics - Constant member functions

- An object of a class can be constant (`const`). Some member functions might not change the object (*constant functions*), but we need to tell the compiler that it doesn't. That is done by adding `const` after the parameter list in the function definition. Then the function can be used for constant objects.

```
template <class Cmppt>
inline const Cmppt& Vector<Cmppt>::x() const
{
    return this->v_[X];
}
```

this function returns a *constant* reference to the X-component to the object (first `const`) without modifying the original object (second `const`)

C++ basics - Friends

- A `friend` is a function (not a member function) or class that has access to the private members of a particular class.
- A class can invite a function or another class to be its friend, but it cannot require to be a friend of another class.

C++ basics - Operators

- Operators define how to manipulate objects
- Standard operator symbols are:

```

new delete new[] delete[]
+ - * / % ^ & ~ |
! = < > += -= *= /= %=
^= &= << >> >>= <<= == != |=
<= >= && || ++ -- , ->* ->
() []

```

when defining operators, one of these must be used.

- Operators are defined as member functions or friend functions with the name `operatorX`, where `X` is one of the operators.
- OpenFOAM has defined operators for all classes, including `iostream <<` and `>>`

C++ basics - Static members

- Static members belongs to a particular object and not to a particular class
- There is exactly one copy of a `static` member instead of one copy per object as for ordinary `non-static` members.
- Examples

```
static const double tol = 1e-3;  
static void setTol(const double&);
```

C++ basics - Inheritance

- A class can inherit attributes from already existing classes, and extend with new attributes.

- Syntax, when defining new class

```
class newClass: public oldClass {...members...}
```

where `newClass` will inherit all the attributes from `oldClass`.

`newClass` is now a *sub-class* to `oldClass`.

- OpenFOAM example:

```
template <class Cmpt>
```

```
class Vector
```

```
:
```

```
    public VectorSpace<Vector<Cmpt>, Cmpt, 3>
```

where `Vector` is a sub-class to `VectorSpace`.

- A member of `newClass` may have the same name of one in the `oldClass`. Then the `newClass` member will be used for `newClass` objects and the `oldClass` member will be hidden. Note that for member functions, all of them with the same name will be hidden, irrespectively of the number of parameters.

C++ basics - Inheritance/visibility

- A hidden member of the old class can be reached by `oldClass::member`.
- Members of the class can be `public`, `private` or `protected`.
- `private` members are never visible in the sub-class, while `public` and `protected` are. However, `protected` are visible only in a sub-class (not in other classes).
- The visibility of the inherited members can be modified in the new class using the reserved word `public`, `private` or `protected` when defining the class.
- A class may be a sub-class to several base classes (multiple inheritance), and this is used to combine features of different classes.

C++ basics - Virtual member functions

- Virtual member functions are used for *dynamic binding*, i.e. the function will work differently depending on how is called, and it is determined at run-time
- The reserved word `virtual` is used in front of the member function declaration to declare it as virtual.
- A sub-class to a class with a virtual function should have a member function with the same name and parameters, and return the same type as the virtual function. That sub-class member function will automatically be a virtual function.
- By defining a pointer to the base class a dynamic binding can be realized. The pointer can be made to point at any of the sub-classes to the base class.
- The pointer to a specific sub-class is defined as:

```
p = new subClass(..parameters..)
```
- Member functions are used as `p->memberFunction` (since `p` is a pointer)
- OpenFOAM uses this to dynamically choose the turbulence model.
- Virtual functions make it easy to add new turbulence models without changing the original classes (as long as the correct virtual functions are declared).

C++ basics - Abstract classes

- A class with at least one virtual member function that is undefined (a *pure* virtual function) is an abstract class.
- The purpose of an abstract class is to define how the sub-classes should be defined.
- An object cannot be created for an abstract class.
- The OpenFOAM `turbulenceModel` is such an abstract class since it has a number of pure member functions, such as

```
//- Return the turbulence viscosity  
virtual tmp<volScalarField> nut() const = 0;
```

(you see that it is *pure* virtual by '= 0')

C++ basics - Containers

- A container class contains and handles data collections. It can be viewed as a list of entries of objects of a specific class. A container class is a sort of *template*, and can be thus used for objects of any class.
- The member functions of a container class are called *algorithms*. There are algorithms that search and sort the data collection etc.
- Both the container classes and the algorithms use *iterators*, which are pointer-like objects.
- The container classes in OpenFOAM can be found in `src/OpenFOAM/containers`, for example `UList`.
- `forAll` is defined to help us march through all entries of a list of objects of any class

C++ basics - Templates

- The most obvious way to define a class is to define it for a specific type of object. However, often similar operations are needed regardless of the object type. Instead of writing a number of identical classes where only the object type differs, a generic *template* can be defined. The compiler then defines all the specific classes that are needed.
- Container classes should be implemented as class templates, so that they can be used for any object (i.e. List of integers, List of vectors...).
- Function templates define generic functions that work for any object.
- A template class is defined by a line in front of the class definition, similar to:

```
template<class T>
```

where `T` is the generic parameter (there can be several in a 'comma' separated list), defining *any* type. The work `class` defines `T` as a *type* parameter.
- The generic parameter(s) are then used in the class definition instead of the specific type name(s).
- A template class is used to construct an object as:

```
templateClass<type> templateClassObject;
```

C++ basics - Typedef (2/2)

- OpenFOAM is full of templates.
- To make the code easier to read, OpenFOAM re-defines the templated class names, for instance:

```
typedef List<vector> vectorList;
```

so that an object of the class template `List` of type `vector` is called `vectorList`.

C++ basics - Namespace

- When using pieces of C++ code developed by different programmers there is a risk that the same name has been used for the same declaration, for instance two constants with the name `size`.
- By associating a declaration with a namespace the declaration will only be visible if that namespace is used. Remember that the standard declarations are used by the starting with:

```
using namespace std;
```

- OpenFOAM declarations belong to namespace `Foam`, so in OpenFOAM we use:
- Explicit naming in OpenFOAM

```
Foam::function();
```

where `function()` is a function defined in namespace `Foam`. This must be used if any other namespace containing a declaration of another `function()` is also visible.

C++ basics - Namespace

- A namespace with the name `name` is defined as

```
namespace name{  
  // declarations  
}
```

- New declarations can be added to the namespace using the same syntax in another part of the code.

Bibliography

The C++ Programming Language, *B. Stroustrup*, Addison-Wesley, 1997

The C++ Standard Library, *N. Josuttis*, Addison-Wesley, 1999

The C++ Standard, John Wiley and Sons, 2003

Accelerated C++, *A. Koenig and B. Moo*, Addison-Wesley, 2000

C++ by Example: UnderC Learning Edition, *S. Donovan*, Que, 2001

Teach Yourself C++, *A. Stevens*, Wiley, 2003

Computing Concepts with C++ Essentials, *C. Horstmann*, Wiley, 2002

Bibliography

Thinking in C++: Introduction to Standard C++, Volume One (2nd Edition) , *B. Eckel*, Prentice Hall, 2000

Thinking in C++, Volume 2: Practical Programming (Thinking in C++) , *B. Eckel*, Prentice Hall, 2003

Effective C++: 55 Specific Ways to Improve Your Programs and Designs, *S. Meyers*, Addison-Wesley, 2005

More Effective C++: 35 New Ways to Improve Your Programs and Designs, *S. Meyers*, Addison-Wesley, 2005

C++ Templates: The Complete Guide, *David Vandevoorde, Nicolai M. Josuttis*, Addison-Wesley, 2002

Bibliography

More bibliography can be found at:

<http://www.a-train.co.uk/books.html>

<http://damienloison.com/Cpp/minimal.html>

Acknowledgements

Prof. Hakan Nilsson from Chalmers University of Technology is gratefully acknowledged for its contribution to the slides presented in this work.