051176 - Computational Techniques for Thermochemical Propulsion

Master of Science in Aeronautical Engineering

Basics of C++ and OpenFOAM

Prof. **Federico Piscaglia**

Dept. of Aerospace Science and Technology (DAER)
Politecnico di Milano - Italy

federico.piscaglia@polimi.it

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:

- **To be a vehicle** to specify actions to be executed
- **To provide a set of concepts** for the programmer to use when thinking about what can be done

For these reason 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: 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; 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 is 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 - Variables

Data types

- In C++ every variable must have an explicit type

```
int myInteger = 10;

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

- Variables must be declared before they are used; it is possible to declare a variable anywhere in the code
- Standard data types are: `bool char int double`
- External libraries can define new data types: `complex vector string`
- 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. (e.g. `double` → `complex`)
- User-defined types must have the required conversions defined.

Streams and operators

Streams

- A particular C++ type is the `stream`
- Streams are used for I/O operations:

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

cin >> myInteger;
```

- Standard streams are `cin` (standard input) and `cout` (standard output)
- operators << and >> are used to put-data-to/get-data-from a stream.
- Streams can be defined to/from a file, a string, a processor, etc. . .

C++ basics - operators, math, flow control

- Math standard functions (trigonometry, logarithmic,. . .) are not part of the C++ itself, but they are in the standard library. Use:

  `"#include <cmath>"`.

- Some common operators are:
  - Arithmetic: `+ - * / % ++ += -- -= *= /=`
  - Comparison: `< > <= >= == !=`
  - Logical: `&& || ! ^ & |`
  - Programming: `& * -> .`

  All of them can be reimplemented for user-defined data types

C++ basics - operators, math, flow control

- `if` statements:

```
if (var1 > var2)
{
  ... code ...
}
else
{
  ... code ...
}
```

- `while` statements

```
while(condition)
{
  ... code ...

  break; //breaks the execution of the loop!
}
```

- `for`-statements:

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

C++ basics - arrays and array templates

- Standard C/C++ arrays:
  - Declaration: `double f[5];` (**Note**: components numbered from 0!)
  - Usage: `f[3] = 2.5;` (**Note**: no out-of-bounds control!)
  - declaration and initialization:
    ```
    int a[6] = {2, 1, 4, 5, 3, 9};
    ```
  - Arrays have strong limitations, but are a good basis for array **templates**.

C++ basics - arrays and array templates

- Array templates (for example `vector`, `list`,..):

  `#include <vector> using namespace std`

- Some common functions are defined, e.g.:
  `size()`, `empty()`, `assign()`, `push_back()`, `clear()`

- Vector declaration and initialization:

```
vector<double> v(3); //return {0, 0, 0}

vector v2(4, 1.5);   //return {1.5, 1.5, 1.5, 1.5}

vector v3(v2);       //copies v2 to v3

v.assign(4, 1);      //return (1.5, 1.5, 1.5, 1.5)
```

C++ basics - functions

- Function declaration paradigm:

```
[const] <returned type> <function name>(<argument list>) [const]
```

  - Functions may or may not return a value (returned type: `void`)
  - Everything written after the `return` has no effect;
  - Number of arguments can vary from 0 to infinity;
  - **Function overloading**: there may be several functions with the same name, as long as there is a difference in the argument list (number of arguments and/or their type) or the returned type.

- 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 function body.

C++ basics - visibility of variables

The visibility of a variable depends on where it is defined:

- A variable defined in a block between ({ }) is visible in that block and its sub-blocks only.
- A variable defined in a function head is visible in the entire function only.
- There can be several variable with the same name, but only one in each block.

```
int x;

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

- the first `x` is not visible inside the `f1` function
- `c` is visible in the entire `f1` but not outside
- the second `x` is not visible outside `f1`

Variables definition and functions

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

- Declaration (The argument names may be omitted here)

```
double average(double x1, double x2) // function head

main()
{
  /*
  ...
  */
}
```

- Definition

```
double average(double x1, double x2) // function definition
{
     return (x1+x2)/2.0;
}
```

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

Function parameters/arguments

- It is possible to pass an argument “by reference” instead than “by copy”.

  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, so that those variables does not have to be explicitly passed when the function is called:

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

int main()
{
  /*
     ...
  */

  integrate(x, y); // will take t=1 integrate(x, y, 5);
}
```

C++ basics - pointers

A **pointer** is a variable that contains the memory address of an object

- 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
- The value of the object pointed by `p` is obtained with the unary `*` operator:

```
char a = *p;
```

C++ basics - references

A **reference** is an **alternative name of an object**:

```
char& b = a; // b is ACTUALLY the same as a
```

- Changing `b` will actually change `a`, and vice-versa
- To prevent `a` is changed by changing `b`, the latter can be declared as `const`:

```
int a = 10;

const int& b = a;

a++; // ok, now a=11;

b++; // error, b is const!
```

C++ basics - Typedef


```
typedef GeometricField<Vector<scalar>, fvPatchField, volMesh>
volVectorField;
```

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);
```

and

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

are equivalent!

`Typedef`(s) make the code easy to read. OpenFOAM makes a wide use of it!

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 object's
  1) construction
  2) destruction
  3) attributes
  4) functions that can manipulate the object itself (methods)
- 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.

C++ basics - Class definition

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

```
class myClass
:
public baseClass  //- class from which myClass is inherited
{

  private:

      //- hidden member functions and data members
      ...
  protected:

      //- members which can be seen ONLY by derived classes
      ...
  public:

      // public member functions and data
      ...
}
```

- `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`
- A member of a class can be – on turn – a new class (member-class or sub-class)

C++ basics - Using a class

An object of class `myClass` is created 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.

Member functions 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..
}
```

- `myClass::` shows that the member function `write` belongs to the class `myClass`
- `void` means that the function does not return any value,
- `inline` – if present – specify that the function must 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.

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. (e.g. `myClass.H` and `myClass.C`)
- 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 used 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 declarad 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 preferrable to define the destructor esplicitly.

C++ basics - Constant member functions

An object of a class can be constant (`const`). Some member functions may not change the object (*constant functions*), but we need to tell the compiler that they don'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 Cmpt>
inline const Cmpt& Vector<Cmpt>::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. (e.g.: `myClass::operator+`)

- OpenFOAM has defined operators for all classes, including `iostream` `<<` and `>>`

C++ basics - Static members

- Static members belongs to a particular class (i.e.: user-defined type) and not to a particular object;

- 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&);
```

Inheritance

A class can **inherit** attributes from already existing classes, and extend with new attributes. The syntax to define a new class is:

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

where `newClass` inherits all the attributes from `oldClass`; `newClass` is now a *derived* class of `oldClass`.

OpenFOAM example:

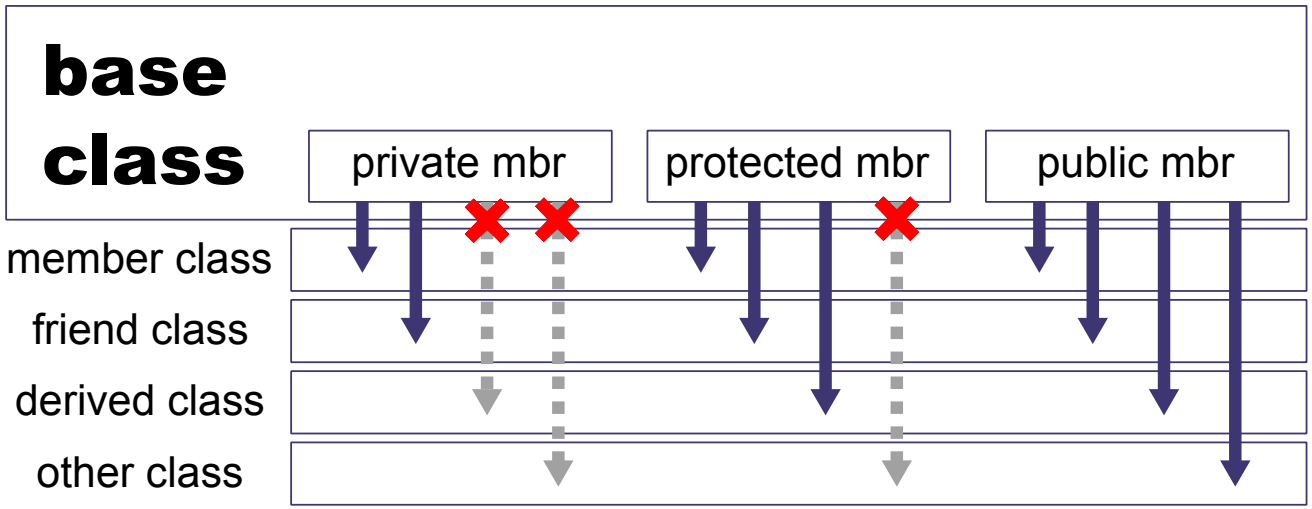
```
template <class Cmpt>
class Vector
: public VectorSpace<Vector<Cmpt>, Cmpt, 3>
{}
```

where `Vector` is a derived class of `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.

Inheritance/visibility

Members of the class can be `public`, `private` or `protected`. A hidden member of the base class can be reached by `oldClass::member`.



There is a sort of "hierarchy" governing 'who can see whom'

- `public` attributes are visible by all classes
- `private` attributes are visibile only by the class itself, its member classes and any **friend** class
- `protected` attributes are visible also by **derived** classes

Virtual member functions


```
// in myBase.H
virtual int myBase::myVirtFun()
{
    return 0;
};

// in myDer_1.H
int myDer_1::myVirtFun()
{
    return 1;
};

// in myDer_2.H
int myDer_2::myVirtFun()
{
    return 0;
};
```

Virtual member functions are used for `dynamic binding`:

- A function (say `myVirtFun()`) is a member of the base class `myBase`
- A function with the same name and arguments exists in at least one class derived from `myBase` (say `myDer_1` and `myDer_2`), but they can have different *definitions*
- The function to be called is selected **run-time** based on the type of the object calling it.

virtual function (example)


```
int main()
{
  myBase* x; // x is of type 'myBase'

  if (a==1)
  {
    x = new myDer_1;
  }
  else if (a==2)
  {
    x = new myDer_2;
  }
  else
  {
    x = new myBase;
  }
  end

  cout << x->myVirtFun();
}
```

- if `a==1`, `myDer_1::myVirtFun()` is called;
- if `a==2`, `myDer_2::myVirtFun()` is called;
- in all the other cases, `myBase::myVirtFun()` is called.

Abstract classes

A member function can be declared as **pure virtual** with the following syntax:

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

- A class with at least one pure virtual function is an *abstract* class
- Pure virtual functions **have no definition**, thus no object can be created for an abstract class
- Pure virtual function must be implemented **in every derived classes**, otherwise they will be abstract classes as well
- The purpose of an abstract class is to create a common interface for all derived classes
- The OpenFOAM `turbulenceModel` is such an abstract class since it has a number of pure virtual member functions, such as

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

Example: pure virtual functions


```
// pure virtual function
// myBase is an abstract class
virtual int myBase::myVirtFun()=0

// in myDer_1.H
int myDer_1::myVirtFun()
{
    return 1;
};

// in myDer_2.H
int myDer_2::myVirtFun()
{
    return 0;
};
```

```
int main()
{
    // x is of type 'myBase'
    myBase* x;

    if (a==1)
        x = new myDer_1;
    else if (a==2)
        x = new myDer_2;
    else
    // error: cannot create objec
        x = new myBase;
    end
    cout << x->myVirtFun();
}
```

- if `a==1`, `myDer_1::myVirtFun()` is called;
- if `a==2`, `myDer_2::myVirtFun()` is called;
- in all the other cases, the program will issue an error, since you cannot create an object of 'abstract' type.

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/containe` for example `UList`.
- `forAll` macro is defined in OpenFOAM® to help us marching through all entries of a list of objects of any class:

```
forAll(listOfObject, iterator)
{
  ...
}
```

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 indentical 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 word `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 - 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`.

- Namespaces can be used to solve any ambiguity, since the same declaration can exist indipendently in two or more different namespaces

- **Explicit namespace resolving:** The use of a specific namespace can be selected by the scope resolving operator `::`

  `Foam::function();`

- Since this can be tedious, the `using` statement can be used:

  `using namespace Foam;`

  will 'replace' every occurrence of `function()` with `Foam::function()`

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

```
namespace myNameSpace{
  // declarations
}
```

Bibliography

The C++ Programming Language: Special Edition, *B. Stroustrup*, Addison-Wesley, 2000.

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.

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/c++_books.html

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

Thank you for your attention!

contact: federico.piscaglia@polimi.it