Introduction to C++ for CFD modeling

Tommaso Lucchini

Department of Energy
Politecnico di Milano

Outline

Overview of the main C++ capabilities applied to CFD practical examples:

- Classes to protect data
- Use of function and operator overloading
- Class derivation
- Virtual functions
- Generic programming with Templates

Why C++?

Current generation of CFD codes

- Very big size and complexity, beyond their expectations
- New functionalities grow their complexity
- 6-12 months required to new engineers to understand and develop new parts of the code
- Due to the software complexity, most of the time is spent on testing and validation

Problems

- Global data can be corrupted anywhere in the software
- Possible interaction between new software components and the existing ones

Solution

- Software separation into manageable units
- Develop and test units in isolation
- Build complex systems from simple components

Each component consists of **data** and **functions**: a **class** (or object).

Classes to protect data

Example: Vector class

Vector, widely used *object* in CFD modeling:

- Position marker (cell centers, face centers, mesh points)
- velocity
- . . .

Define a *Vector* class that can be used for all these purposes. Implementation:

- Class *members*
- Constructors and destructor
- Member functions:
  - Access
  - Operators
  - IO
  - . . .

class Vector
Members and enumeration

```
class Vector
{
    // Private data

    //- Components
    double V[3];

public:

    // Component labeling enumeration
    enum components { X, Y, Z };
```

The vector components are *private data*. In this way the vector components are protected from corruption.

Enumeration: a type that can hold a set of values specified by the user. Once defined, an enumeration is used like an integer type. Use `v[Vector::X]` or `v[X]` instead of `v[0]`.

class Vector

Constructors

```
// Constructors
//- Construct null
Vector(){}

//- Construct given three scalars
Vector(const double& Vx, const double& Vy, const double& Vz)
{
    V[X] = Vx; V[Y] = Vy; V[Z] = Vz;
}

//Destructor
~Vector(){}
```

class Vector
Member functions

```
// Member Functions

const word& name() const;
static const dimension& dimensionOfSpace();

const double& x() const { return V[X]; }
const double& y() const { return V[Y]; }
const double& z() const { return V[Z]; }

double& x() { return V[X]; }
double& y() { return V[Y]; }
double& z() { return V[Z]; }
```

Member functions provide an interface for data manipulation, but the data are **directly accessible** only within the class: **data protection**.

`class Vector`
Member operators

```
// Member Operators

void operator=(const Vector& v);

inline void operator+=(const Vector&);
inline void operator-=(const Vector&);
inline void operator*=(const scalar);

//Friend Functions

friend Vector operator+(const Vector& v1, const Vector& v2)
{
    return Vector(v1[X]+v2[X], v1[Y]+v2[Y], v1[Z]+v2[Z]);
}
```

Member operators and *friend functions* perform operations on the class members.

class Vector

Member operators

```
        friend double operator&(const Vector& v1, const Vector& v2)
        {
            return (v1[X]*v2[X] + v1[Y]*v2[Y] + v1[Z]*v2[Z]);
        }

        friend Vector operator^(const Vector& v1, const Vector& v2)
        {
            return Vector
            (
                (v1[Y]*v2[Z] - v1[Z]*v2[Y]),
                (v1[Z]*v2[X] - v1[X]*v2[Z]),
                (v1[X]*v2[Y] - v1[Y]*v2[X])
            );
        }

    }; // end of the Vector class implementation
```

`class Vector`
Considerations

Summary

- Class is the only responsible for his own data management.
- Class provides the interface for data manipulation.
- Data are directly accessible only within the class implementation: **data protection**.
- The Vector class is a *code component* and can be developed and tested in isolation.

$\Rightarrow$ ...easy debug: any problem is related to the class.

Manipulating vectors:

```
        Vector a, b, c;
        Vector area = 0.5 * ((b-a)^(c-a));
```

class Vector
Constant and non-constant access

Pass-by-value and pass-by-reference: is the data being changed?

```
        const double& x() const { return V[X]; }
        double& x() { return V[X]; }
```

The user interface for the vector class provide both the constant and non-constant access to the class members.

```
        class cell
        {
            Vector centre_;
        public:
            const Vector& centre() const;
        };
```

The cell center is a class member. The `centre()` *member function* provides the constant access to the center vector and it is not possibile to modify it outside the class.

Operator overloading

New classes + built-in operators

Implementing the **same operations** on **different types**

- Some operators are generic, like magnitude (same name, different arguments):

```
        label m = mag(-3);
        scalar n = mag(3.0/m);

        Vector r(1, 3.5, 8);
        scalar magR = mag(r);
```

- Function/operator syntax:

```
        Vector a, b;
        Vector c = 3.4*(a - b);
```

is indentical to (the compilar does the same thing):

```
        Vector c(operator*(3.7, operator+(a, b));
```

Class derivation

Particle class

Defining the class particle. Position and location.

- Position in space: vector = point
- Cell index, boundary face index, is on a boundary?

```
class particle
:
    public Vector
{
    // Private data

    //- Index of the cell it is
    label cellIndex_;

    //- Index of the face it is
    label faceIndex_;

    //- is particle on boundary/outside domain
    bool onBoundary_;
};
```

- *is-a relationship*: class is derived from another class.
- has-a relationship: class contains member data.

Virtual functions
Implementing boundary condition

- Boundary conditions represent a class of related objects, all doing the same job:
  - Hold boundary values and rules on how to update them.
  - Specify the boundary condition effect on the matrix.
- . . . but each boundary condition does this job in it own specific way!
- Examples: fixed value (Dirichlet), zero gradient (Neumann), mixed, symmetry plane, periodic and cyclic etc.
- However, the code operates on all boundary conditions in a consistent manner

Virtual functions
Implementing boundary condition

- Possible implementation of boundary conditions

```
enum kind {fixedValue, zeroGradient, symmetryPlane, mixed};

class boundaryCondition
{

    kind k;

    //other objects

public:

    void updateCoeffs();
    void evaluate();
};
```

Virtual functions

Implementing boundary condition

- The type field k is necessary to identify what kind of boundary condition is used. In this case the `evaluate` function will be something like:

```
boundaryCondition::evaluate()
{
    switch k
    {
        case fixedValue: {// some code here}
        case zeroGradient : {// some code here}
        // implementation of other boundary conditions
    }
}
```

- This is a mess!
  - This function should know about all the kinds of boundary conditions
  - Every time a new boundary condition is added, this function grow in shape
  - This introduces bugs (touch the code...)
- Virtual functions solve this problem

Virtual functions

Implementing boundary condition

- There is no distinction between the general properties of each boundary condition, and the properties of a specific boundary condition.
- Expressing this distinction and taking advantage of it defines object-oriented programming.
- The inheritance mechanism provides a solution:
  - Class representing the general properties of a boundary condition

```
class fvPatchField
{
public:
    virtual void evaluate() = 0;
    virtual void updateCoeffs() = 0;
};
```

  - In the generic boundary condition, the functions `evaluate()` and `updateCoeffs()` are *virtual*. Only the calling interface is defined, but the implementation will be done in the specific boundary condition classes.

Virtual functions

Implementing boundary condition

- Then, specific boundary conditions are derived from the generic class:

```
//Dirichlet boundary condition

class fixedValueFvPatchField
:
    public fvPatchField
{
    double value;

public:
    virtual void evaluate()
    {
        // some code..
    }
    virtual void updateCoeffs()
    {
        // some code..
    }
};
```

- And they contain the implementation of the virtual functions.

Virtual functions

Implementing boundary condition

- The rest of the code operates only with the generic conditions

```
List<fvPatchField*> boundaryField;
forAll (boundaryField, patchI)
{
    boundaryField[patchI]->evaluate();
}
```

- When a virtual function is called (generic; on the base class), the actual type is recognised and the specific (on the derived class) is called at run-time
- The ”generic boundary condition” only defines the behaviour for all derived (concrete) classes and does not really exist
- Consequences
  - New functionality does not disturb working code
  - New derived class automatically hooks up to all places
  - Shared functions can be implemented in base class

Generic programming
Templates

- Someone who want a list is unlikely always to want a list of integers.
- A list is a general concept independent on the notion of an integer.
- If an algorithm can be expressed independently of representation details and if it can be done so affordably without logical contorsions, it should be ought to be done so.
- In C++ it is possible to generalize a list-of-integers type by making it a *template* and replacing the specific type *integer* with a template parameter. For example:

```
template<class T>class List  {  // ...  }
```

- Once the class is defined, we can use it as follows:

```
List<int> intList;
List<cell> cellList;
```

- The compiler will expand the code and perform optimisation after expansion.
- Generic programming techniques increase the power of software: less software to do more work.
- Easy debug: if it works for one type, it will work for all.

Generic programming

Example - List class

```
template<class T>
class List
{
public:
    //- Construct with given size
    explicit List(const label);

    //- Copy constructor
    List(const List<T>&);

    //- Destructor
    ~List();

    //- Reset size of List
    void setSize(const label);

    //- Return subscript-checked element of List
    inline T& operator[](const label);

    //- Return subscript-checked element of constant LList
    inline const T& operator[](const label) const;
};
```

Generic programming

List class - bubble sort algorithm implementation and application

```
template<class Type>
void Foam::bubbleSort(List<Type>& a)
{
    Type tmp;
    for (label i = 0; i < n - 1; i++)
    {
        for (label j = 0; j < n - 1 - i; j++)
        {
            // Compare the two neighbors
            if (a[j+1] < a[j])
            {
                tmp = a[j]; // swap a[j] and a[j+1]
                a[j] = a[j+1];
                a[j+1] = tmp;
            }
        }
    }
}

List<cell> cellList(55); // Fill in the list here
bubbleSort(cellList);
```

Conclusions

C++ Object-oriented programming techniques for CFD modeling

- The code complexity is handled by splitting up the software into smaller and protected units, implemented and tested in isolation.
- The **class** is the base unit. Consists of data and functions that operate on it. Possibility to protect the data from outside corruption.
- Classes allow introduction of user-defined types, relevant to the problem under consideration $\Rightarrow$ *vector, field, matrix, mesh*.
- Virtual functions handle cases where a set of classes describe variants of related behaviour through a common interface $\Rightarrow$ *boundary conditions*.
- Generic programming with **templates**.
  - Use for algorithms which are type-independent.
  - Combines convenience of single code with optimisation of hand-expanded code.
  - Compiler does additional work: template instantiation.
- C++ is a large and complex language; OpenFOAM uses it in full.

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

Dr. Hrvoje Jasak is gratefully acknowledged for providing most of the material displayed in this presentation