Object Orientation and C++: An Introduction for CFD Specialists

Hrvoje Jasak

h.jasak@wikki.co.uk

Wikki Ltd, United Kingdom

Outline

Objective

- Quick overview of object orientation and generic programming in C++ on CFD-related practical examples

Topics

- Classes: protecting your data
- Handling data: value, reference and pointer access
- Function and operator overloading
- Class derivation
- Virtual functions
- Templating: generic programming in C++
- Summary and further reading

Background

Rationale

- Current generation of CFD software has grown in size beyond expectations
- Addition of new functionality hampered by software complexity
- A new (development) engineer needs a minimum of 6-12 months training to understand and develop isolated parts of software
- As a consequence of complexity, large percentage of development time is spent on testing and validation: not productive and bug-prone

Major Problems

- Data is global and can be corrupted anywhere in the software
- Introduction of new components interferes with already implemented functionality

Solution: **Divide and Conquer**

- Separate software into manageable units
- Develop, test and deploy units in isolation
- Build complex system from simpler components

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

Object Orientation

WIKKI

Object-Oriented Software: Create a Language Suitable for the Problem

- Analysis of numerical simulation software through object orientation: "Recognise main objects from the numerical modelling viewpoint"
- Objects consist of **data** they encapsulate and **functions** which operate on the data

Example: Sparse Matrix Class

- **Data members**: protected and managed
  - Sparse addressing pattern (CR format, arrow format)
  - Diagonal coefficients, off-diagonal coefficients
- Operations on matrices or data members: **Public interface**
  - Matrix algebra operations: $+, -, *, /$,
  - Matrix-vector product, transpose, triple product, under-relaxation
- Actual data layout and functionality is important only internally: efficiency

Example: Linear Equation Solver

- Operate on a system of linear equations $[A][x] = [b]$ to obtain $[x]$
- It is irrelevant how the matrix was assembled or what shall be done with solution
- Ultimately, even the solver algorithm is not of interest: all we want is new $[x]$!
- Gauss-Seidel, AMG, direct solver: all answer to the same interface

Classes: Protecting Your Data

WIKKI

```
class vector
{
    // Private data

        //- Components
        double V[3];

public:

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

    // 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;
        }
```

Classes: Protecting Your Data

WIKKI

```
// Destructor

    ~vector();

// 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 Operators

    void operator=(const vector& v);

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

Classes: Protecting Your Data

```
    // Friend Functions

        friend vector operator+(const vector& v1, const vector& v2)
        {
        return vector(v1.V[X]+v2.V[X],v1.V[Y]+v2.V[Y],v1.V[Z]+v2.V[Z]);
        }

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

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

Classes: Protecting Your Data

WIKKI

Vector Class: Summary

- Class is responsible for managing its own data: (x, y, z)
- Class provides interface for data manipulation; private data is accessible only from within the class: **data protection**
- Vector class (code component) can be developed and tested in isolation

Manipulating Vectors

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

Constant and Non-Constant Access

- Accessing a vector from another class

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

- Cell class allows me to look at the vector **but not to change it!**
- First const promises not to change the vector
- Second promises not to change class object

Value, Pointer, Reference

Types of Data Access

- **Pass by value**: make a copy of local data. Changing a copy does not influence the original, since it is a different instance
- **Pass by const reference**: Give read-only access to local data
- **Pass by reference**: Give read-write access to local data. This type of access allows local value to be changed, usually changing the class
- **Pointer handling**: Instead of dealing with data, operative with memory locations (addresses) where the data is stored

Constant and Non-Constant Access

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

```
class cell
{
public:
    vector centre() const;
    vector* centrePtr();
    vector& centre();
    const vector& centre() const;
};
```

Value, Pointer, Reference

Return a Value, a Pointer or a Reference

- In actual compiler implementation, pointers and references are handled with the same mechanism: memory address of object storage
- Dealing with pointers and protecting pointer data is cumbersome and ugly

```
class cell
{
public:
    vector* centrePtr();
    const vector const* centre() const;
};

cell c(...);

if (c.centrePtr() != NULL)
{
    vector& ctr = *(c.centrePtr());
    ctr += vector(3.3, 1, 1);
}
```

- Is the pointer set? Is it valid: check for NULL. Should I delete it?
- A **reference** is an “always on” pointer with object syntax. No checking required

Operator Overloading

WIKKI

Implementing the Same Operation on Different Types

- User-defined types (classes) should behave exactly like the “built-in” types
- Some operations are generic: *e.g.* magnitude: same name, different argument

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

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

- Warning: **implicit type conversion** is a part of this game! This allows C++ to convert from some types to others according to specific rules
- Function or operator syntax

```
vector a, b;
vector c = 3.7*(a + b);
```

is identical to

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

- Operator syntax is regularly used because it looks nicer, but for the compiler both are “normal” function calls

Class Derivation

Particle Class: 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

WIKKI

Implementing Boundary Conditions

- 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

```
forAll (boundaryConditions, i)
{
    if (boundaryConditions[i].type() == fixedValue)
    {
        // Evaluate fixed value b.c.
    }
    else if (boundaryConditions[i].type() == zeroGradient)
    {
        // Evaluate zero gradient b.c.
    }
    else if (etc...)
}
```

Virtual Functions

WIKKI

Implementing Boundary Conditions

- The above pattern is repeated throughout the code. Introducing a new boundary condition type is complex and error-prone: many distributed changes, no checking
- In functional code, the for-loop would contain an if-statement. When a new boundary conditions is added, the if-statement needs to be changed (for all operations where boundaries are involved)
- We wish to consolidate a boundary condition into a class. Also, the actual code remains independent on **how** the b.c. does its work!
- Codify a generic boundary condition interface: **virtual base class**

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

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

Virtual Functions

Implementing Boundary Conditions

- Working with **virtual functions**
  1. Define what a “generic boundary condition” is the through functions defined on the base class: *virtual functions*
  2. Implement the specific (*e.g.* fixed value) boundary conditions to answer to generic functionality in its own specific way
  3. The rest of the code operates only with the generic conditions
  4. 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
- Note that the “generic boundary condition” does not really exist: it only defines the behaviour for all derived (concrete) classes
- Consequences
  - Code will not be changed when new condition is introduced: no if-statement to change: new functionality does not disturb working code
  - New derived class automatically hooks up to all places
  - Shared functions can be implemented in base class

Virtual Functions

```
template<class Type>
class fvPatchField
:
    public Field<Type>
{
public:

    //- Construct from patch, internal field and dictionary
    fvPatchField
    (
        const fvPatch&,
        const Field<Type>&,
        const dictionary&
    );

    //- Destructor
    virtual ~fvPatchField();

    virtual bool fixesValue() const { return false; }

    virtual void evaluate() = 0;
};
```

Virtual Functions

```
template<class Type>
class fixedValueFvPatchField
:
    public fvPatchField<Type>
{
public:

    //- Construct from patch, internal field and dictionary
    fixedValueFvPatchField
    (
        const fvPatch&,
        const Field<Type>&,
        const dictionary&
    );

    //- Destructor
    virtual ~fixedValueFvPatchField();

    virtual bool fixesValue() const { return true; }

    virtual void evaluate() {}
};
```

Generic Programming: Templates

WIKKI

What are Templates?

- Some operations are independent of the type on which they are being performed. Examples: container (list, linked list, hash table), sorting algorithm
- C++ is a **strongly type language**: checks for types of all data and functions and gives compiler errors if they do not fit
- Ideally, we want to implement algorithms generically and produce custom-written code before optimisation
- Templating mechanism in C++
  - Write algorithms with a generic type holder

    ```
    template<class Type>
    ```
  - Use generic algorithm on specific type

    ```
    List<cell> cellList(3);
    ```
  - Compiler to expand the code and perform optimisation after expansion
- Generic programming techniques massively increase power of software: less software to do more work
- Debugging is easier: if it works for one type, it will work for all
- . . . but writing templates is trickier: need to master new techniques
- Many “CFD” operations are generic: lots of templating in OpenFOAM

Generic Programming: Templates

```
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: Templates

WIKKI

Bubble sort algorithm

```
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);
```

Object Orientation

WIKKI

Main Objects

- Computational domain

Object	Software representation	C++ Class
Tensor	(List of) numbers + algebra	vector, tensor
Mesh primitives	Point, face, cell	point, face, cell
Space	Computational mesh	polyMesh
Time	Time steps (database)	time

- Field algebra

Object	Software representation	C++ Class
Field	List of values	Field
Boundary condition	Values + condition	patchField
Dimensions	Dimension set	dimensionSet
Geometric field	Field + mesh + boundary conditions	geometricField
Field algebra	$+ \; - \; * \; / \; tr(), sin(), exp() \ldots$	field operators

Object Orientation

WIKKI

Main Objects

- Linear equation systems and linear solvers

Object	Software representation	C++ Class
Linear equation matrix	Matrix coefficients	lduMatrix
Solvers	Iterative solvers	lduMatrix::solver

- Numerics: discretisation methods

Object	Software representation	C++ Class
Interpolation	Differencing schemes	interpolation
Differentiation	ddt, div, grad, curl	fvc, fec
Discretisation	ddt, d2dt2, div, laplacian	fvm, fem, fam

- Top-level code organisation

Object	Software representation	C++ Class
Model library	Library	eg. turbulenceModel
Application	main()	–

Summary

Object Orientation and Generic Programming Techniques in CFD

- Object-oriented approach handles complexity by splitting up the software into smaller and protected units, implemented and tested in isolation
- Base unit: a **class**. Consists of data and functions that operate on it. Data is protected from outside corruption: const access
- Classes allow introduction of **user-defined types**, relevant to the problem under consideration. Examples: vector, field, matrix, mesh
- **Virtual functions** handle cases where a set of classes describe variants of related behaviour through a common interface. Example: boundary conditions
- **Templates**: generic programming mechanism in C++. 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
- Question on efficiency in object orientation: resolved. However, care in implementation and understanding of the language is required!

Learning C++

Further Info

- Extensive bibliography on C++ and object orientation
- OpenFOAM library as a source of quality examples of C++ in use
- Basic and advanced C++ courses offered by Wikki Ltd.

Standard References and Recommended Textbooks

- The C++ Primer, Stanley B. Lippman, Josee Lajoie, Barbara E. Moo. Addison Wesley 2005
- The C++ Programming Language, Bjarne Stroustrup, Addison Wesley 2000
- The C++ Standard Library: A Tutorial and Reference, Nicolai M. Josuttis, Addison Wesley 1999
- ISO Standard document ISO/IEC 14882:2003 Programming languages - C++: Normative References: ISO/IEC 9899:1999, ISO/IEC 10646-1:2000

Learn it Well

- Effective C++: 55 Specific Ways to Improve Your Programs and Designs (Professional Computing S.), Scott Meyers, Addison Wesley 1996
- C++ Templates: The Complete Guide, David Vandevoorde, Nicolai M. Josuttis, David Vandervoorde, Addison Wesley 2002