Object orientation in C++ and OpenFOAM

Object orientation in C++ and OpenFOAM

- To begin with: The aim of this part of the course is not to teach all of C++ and object orientation, but to give a short introduction that is useful when trying to understand the contents of OpenFOAM.
- After this introduction you should be able to *recognize* and make *minor modifications* to most C++ features in OpenFOAM.
- Some books:
  - *C++ direkt* by Jan Skansholm (ISBN 91-44-01463-5)
  - *C++ from the Beginning* by Jan Skansholm (probably similar)
  - *C++ how to Program* by Paul and Harvey Deitel
  - *Object Oriented Programming in C++* by Robert Lafore

C++ types, classes and objects

- The *types* that we have just had a look at are in fact *classes*, and the variables we assign to a *type* are *object*s of that class.
- Object orientation focuses on the *objects* instead of the functions.
- An *object* belongs to a *class* of objects with the same attributes. The class defines:
  - The construction of the object
  - The destruction of the object
  - Attributes of the object (member data)
  - Functions that can manipulate the object (member functions)

  I.e. it is the `int` class that defines how the operator `+` should work for objects of that class, and how to convert between classes if needed (e.g. $1 + 1.0$ involves a conversion).

C++ types, classes and objects

- The objects may be related in different ways, and the classes may inherit attributes from other classes.
- A benefit of object orientation is that the classes can be re-used, and that each class can be designed and bug-fixed for a specific task.
- In OpenFOAM, the classes are designed to define, discretize and solve PDE's.

C++ class definition

- The following structure defines the class with name `myClass` and its public and private member functions and member data. This is a general description. We will have a look at examples in the code later.

```
class myClass {
public:
    declarations of public member functions and member data
private:
    declaration of hidden member functions and member data
};
```

- `public` attributes are visible from outside the class.
- `private` attributes are only visible within the class.
- If neither `public` nor `private` are specified, all attributes will be `private`.
- Declarations of member functions and member data are done just as functions and variables are declared outside a class.

C++ class usage

- An object of a class `myClass` is defined in the main code as:
  `myClass myObject;` (c.f. `int i`)
- The object `myObject` will then have all the attributes defined in the class `myClass`.
- Any number of objects may belong to a class, and the attributes of each object will be separated.
- There may be pointers and references to any object.
- The member functions operate on the object according to its implementation.
  If there is a member function `write` that writes out the contents of an object of the class `myClass`, it is called in the main code as:
  `myObject.write();`
- When using the member functions through a pointer, the syntax is slightly different (here `p1` is a pointer to the object `myObject`, and `p2` is a pointer to a nameless new `myClass`:

```
p1 = &myObject;
p2 = new myClass;
p1->write();
p2->write();
```

C++ member functions

- The member functions may be defined either in the *declaration* of the class, or in the *definition* of the class. We will see this when we look inside OpenFOAM. The syntax is basically:

```
inline void myClass::write()
{
  Contents of the member function.
}
```

where

  - `myClass::` tells us that the member function `write` belongs to the class `myClass`.
  - `void` tells us that the function does not return anything
  - `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 declaration will automatically be inlined if possible.

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

C++ organization of classes

- A good programming standard is to make the class files in pairs, one with the class *declarations*, and one with the class *definitions*.
- The class *declaration* file must be included in the files where the class is used, i.e. the class *definition* file and files that inherits from, or construct objects of that class.
- The compiled *definition* file is statically or dynamically linked to the executable by the compiler.
- Inline functions must be implemented in the class *declaration* file, since they must be inlined without looking at the class *definition* file. In OpenFOAM there are usually files named as `VectorI.H` containing `inline` functions, and those files are included in the corresponding `Vector.H` file.
- Let's have a look at some examples in the OpenFOAM `Vector` class:
  `$FOAM_SRC/OpenFOAM/primitives/Vector`
  (go there while looking at the following slides)

C++ constructors

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

```
// 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` will be initialized differently depending on which of these constructors is chosen.

C++ constructors

- The actual initialization usually takes place in the corresponding `.C` file, but since the constructors for the `Vector` are inlined, it takes place in the VectorI.H file:

```
// Construct given three Cmpts
template <class Cmpt>
inline Vector<Cmpt>::Vector(const Cmpt& vx, const Cmpt& vy, const Cmpt& vz)
{
    this->v_[X] = vx;
    this->v_[Y] = vy;
    this->v_[Z] = vz;
}
```

  Here, `this` is a pointer to the current object of the current class, i.e. we here set the static data member `v_` (inherited from class `VectorSpace.H`) to the values supplied as arguments to the constructor.

- It is here obvious that the member function `Vector` belongs to the class `Vector`, and that it is a constructor since it has the same name as the class.

C++ constructors

- A copy constructor has a parameter that is a reference to another object of the same class:

  ```
  myClass(const myClass&);
  ```

  The *copy constructor* copies all attributes. A copy constructor can only be used when initializing an object (since a constructor 'constructs' a new object). Usually there is no need to define a copy destructor since the default one does what you need.

- 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. (There can actually be more parameters, but then they have to have default values)

C++ 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 as the class, but with a $\sim$ 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 explicitly.

C++ constant member functions

- An object of a class can be constant (`const`). Some member functions might in fact not change the object, but we need to tell the compiler that it doesn't (*constant functions*). 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 of the object (first `const`) without modifying the original object (second `const`)

C++ 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.
- `friend`s are defined in the definition of the class using the special word `friend`.

C++ 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 name `operatorX`, where X is an operator symbol.
- OpenFOAM has defined operators for all classes, including `iostream` operators << and >>
- See example at the end of `VectorI.H`

C++ static members

- Static members of a class only exist in a single instance in a class, for all objects, i.e. it will be equivalent in all objects of the class.
- They are defined as `static`, which can be applied to member data or member functions.
- Static members do not belong to any particular object, but to a particular class, so they are used as:

  ```
  myClass::staticFunction(parameters);
  ```

  They can actually also be used as `myObject.staticFunction(parameters)`, but that would be a bit mis-leading since nothing happens explicitly to `myObject`, and that all objects of the class will notice the effect of the call of `staticFunction`.
- In `Vector.H` we have:

```
// Static data members
    static const char* const typeName;
    static const char* componentNames[];
    static const Vector zero;
    static const Vector one;
    static const Vector max;
    static const Vector min;
```

C++ inheritance

- A class can inherit attributes from already existing classes, and extend with new attributes.
- Syntax, when defining the 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 class `Vector` is a sub-class to `VectorSpace`.

- An attribute of `newClass` may have the same name as one in `oldClass`. Then the `newClass` attribute will be used for `newClass` objects and the `oldClass` attribute 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++ inheritance and visibility

- A hidden member of a base-class can be reached by `oldClass::member`
- Members of a class can be `public`, `private` or `protected`.
- `private` members are never visible in a sub-class, while `public` and `protected` are. However, `protected` are only visible in a sub-class (not in other classes).
- The visibility of the inherited members can be modified in the new class using the reserved words `public`, `private` or `protected` when defining the class. (`public` in the previous example). It is only possible to make the members of a base-class *less visible* in the sub-class.
- A class may be a sub-class to several base-classes (multiple inheritance), and this is used to combine features from several classes. Watch out for ambigous (tvetydiga) members!

C++ virtual member functions

- Virtual member functions are used for dynamic binding, i.e. the function will work differently depending on how it 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 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 defined).

C++ 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 can not be created for an abstract class.
- The OpenFOAM `LESModel` is such an abstract class since it has a number of pure member functions, such as
  (see `$FOAM_SRC/turbulenceModels/incompressible/LES/LESModel/LESModel.H`)

```
//- Return the SGS viscosity.
virtual tmp<volScalarField> nuSgs() const = 0;
```

  (you see that it is *pure* virtual by '`= 0`', and that there is no description of member function `nuSgs()` in the `LESModel` class - the description is specific for each turbulence model that inherits the `LESModel` class).
- The most important function in the LESModel class is the `correct()` function, which is called as a pointer. E.g. `$FOAM_SOLVERS/incompressible/pimpleFoam/pimpleFoam.C`:
  `turbulence->correct();`

C++ container classes

- A container class contains and handles data collections. It can be viewed as a list of entries of objects a specific class. A container class is a sort of *template*, and can thus be 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 `$FOAM_SRC/OpenFOAM/containers`, for example `Lists/UList`
- `forAll` is defined in `UList.H` to help us march through all entries of a list of objects of any class:

```
#define forAll(list, i) \
    for (Foam::label i=0; i<(list).size(); i++)
```

Search OpenFOAM for examples of how to use `forAll`, e.g.:

```
forAll(anyList, i) { statements; }
```

C++ 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 ...)
- 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++ typedef

- 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;
```

- A list of vectors can then simply be constructed as

```
integerVector iV;
```

This is used to a large extent in OpenFOAM, and the reason for this is to make the code easier to read.

C++ namespace implementation

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

```
namespace myNamespace {
    declarations
}
```

  You can see

```
namespace Foam { }
```

  all over OpenFOAM.

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