Basics of C++ and object orientation in OpenFOAM

- To begin with: The aim of this part of the course is not to teach all of C++, 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.
- We will follow the book *C++ direkt* by Jan Skansholm (ISBN 91-44-01463-5)

C++ basics

- Variables can contain data of different *types*, for instance `int myInteger;` for a declaration of an integer variable named `myInteger`, or `const int myConstantInteger = 10;` for a declaration of an *constant* integer variable named `myConstantInteger` with value 10. In C++ it is possible to define special *types*, and there are many types defined for you in OpenFOAM.

- Input and output can be done using the standard library `iostream`, using:

  ```
  cout << "Please type an integer!" << endl;
  cin >> myInteger;
  ```

  where << and >> are output and input operators, and `endl` is a manipulator that generates a new line (there are many other manipulators). In OpenFOAM a new output stream `Info` is however defined, and it is recommended to use that one instead since it takes care of write-outs for parallel simulations.

- Variables can be added, substracted, 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, Example code

```
#include <iostream>
using namespace std;

main()
{
int myInteger;
const int constantInteger=5;
const float constantFloat=5.1;
cout << "Please type an integer!" << endl;
cin >> myInteger;
cout << myInteger << " + " << constantInteger << " = "
     << myInteger+constantInteger << endl;
cout << myInteger << " + " << constantFloat << " = "
     << myInteger+constantFloat << endl;
}
```

Compile and run with:

```
g++ basic.C -o basic
./basic
```

C++ basics

- `+`, `-`, `*` and `/` are operators that define how the operands should be used. Other standard operators are `%` (integer division modulus), `++` (add 1), `--` (substract 1), `+=` (`i+=2` adds 2 to i), `-=`, `*=`, `/=`, `%=` etc. User-defined types should define its operators.

- Mathematic standard functions are available in standard libraries. They are thus not part of C++ itself.
  Standard library `cmath` contains trigonometric functions, logaritmic functions and square root. (use `#include cmath;` if you need them)
  Standard library `cstdlib` contains general functions, and some of them can be used for arithmetics. (use `#include cstdlib;` if you need them)

- if-statements: `if (variable1 > variable2) {...CODE...} else {...CODE...}`
  Comparing operators: `<  >  <=  >=  ==  !=`
  Logical operators: `&&  ||  !` (or, for some compilers: `and  or  not`)
  Generates `bool` (boolean)

- while-statements: `while (...expression...) {...CODE...}`

- `break;` breaks the execution of `while`

C++ basics, Example code

```
#include <iostream>
#include <cmath>
using namespace std;

main()
{
float myFloat;
cout << "Please type a float!" << endl;
cin >> myFloat;
cout << "sin(" << myFloat << ") = " << sin(myFloat) << endl;
if (myFloat < 5.5){cout << myFloat << " is less than 5.5" << endl;} else
                  {cout << myFloat << " is not less than 5.5" << endl;};
}
```

Compile and run with:

```
g++ basic.C -o basic
./basic
```

C++ basics

- for-statements: `for ( init; condition; change ) {...CODE...}`
- Arrays:
  `double f[5];` (Note: components numbered from 0!)
  `f[3] = 2.75;` (Note: no index control!)
  `int a[6] = {2, 2, 2, 5, 5, 0};` (declaration and initialization)
  The arrays have strong limitations, but serve as a base for array **templates**
- Array **templates** (example `vector`. other: `list, deque`):
  `#include <vector>`
  `using namespace std`
  The type of the vector must be specified upon declaration:
  `vector<double> v2(3);` gives $\{0, 0, 0\}$
  `vector<double> v3(4, 1.5);` gives $\{1.5, 1.5, 1.5, 1.5\}$
  `vector<double> v4(v3);` Constructs `v4` as a copy of `v3` (copy-constructor)
- Array template operations: The template classes define member functions that can be used for those types, for instance: `size()`, `empty()`, `assign()`, `push_back()`, `pop_back()`, `front()`, `clear()`, `capacity()` etc.
  `v.assign(4, 1.0);` gives $\{1.0, 1.0, 1.0, 1.0\}$

C++ basics, Example code

```
#include <iostream>
#include <vector>
using namespace std;

main()
{
vector<double> v2(3);
vector<double> v3(4, 1.5);
vector<double> v4(v3);
cout << "v2: (" << v2[0] << "," << v2[0] << "," << v2[0] << ")" << endl;
cout << "v3: (" << v3[0] << "," << v3[0] << "," << v3[0] << ")" << endl;
cout << "v4: (" << v4[0] << "," << v4[0] << "," << v4[0] << ")" << endl;
//Note that the class is not implemented to be able to execute:
//cout << "v2: " << v2 << endl;
cout << "v2.size(): " << v2.size() << endl;
}
```

Compile and run with:

```
g++ basic.C -o basic
./basic
```

C++ basics

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

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

  takes two arguments of type `double`, and returns type `double`. The variable `nvalues` is a local variable, and is only visible inside the function. Note that any code after the `return` statement will not be executed.
- A function doesn't have to take arguments, and it doesn't have to return anything (the output type is then specified as `void`). The `main` function should return an integer, but it is alright to skip the return statement in the `main` function.
- There may be several functions with the same names, as long as there is a difference in the arguments to the functions - the number of arguments or the types of the arguments.

C++ basics, Example code

```
#include <iostream>
using namespace std;

double average (double x1, double x2)
{
  int nvalues = 2;
  return (x1+x2)/nvalues;
  //This line is not executed since it is after the return statement:
  cout << "Hello!" << endl;
}

main()
{
double d1=2.1;
double d2=3.7;
cout << "Average: " << average(d1,d2) << endl;
}
```

Note that the function must be *declared* before it is used (*defined*), i.e. it can not be added *after* the main function. On the other hand, a separate declaration can be used...

C++ basics, declaration of variables and functions

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

```
double average (double x1, double x2);
main ()
{
  mv = average(value1, value2)
}
double average (double x1, double x2)
{
  return (x1+x2)/2;
}
```

  The second occurence of the function head is the definition of the function. The argument *names* may be omitted in the declaration (first occurence).

- Declarations are often included from include-files (`#include "file.h"` or `#include <standardfile>`)

- A good way to program C++ is to make files in pairs, one with the declaration, and one with the definition. This is done throughout OpenFOAM.

C++ basics, Example code

```
#include <iostream>
#include "basic.H"
using namespace std;

main()
{
double d1=2.1;
double d2=3.7;
cout << "Average: " << average(d1,d2) << endl;
}

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

The file `basic.H` contains:

```
double average (double, double);
```

C++ basics, function parameters / arguments

- If an argument variable should be changed inside a function, the type of the argument must be a reference, i.e.
  `void change(double& x1)`
  The reference parameter `x1` will now be a reference to the argument to the function instead of a local variable in the function. (standard arrays are always treated as reference parameters).

- Reference parameters can also be used to avoid copying of large fields when calling a function. To avoid changing the parameter in the function it can be declared as `const`, i.e.
  `void checkWord(const string& s)`
  This often applies for parameters of class-type, which can be large.

- Default values can be specified, and then the function may be called without that parameter, i.e.
  `void checkWord(const string& s, int nmbr=1)`

C++ basics, Example code

```
#include <iostream>
using namespace std;

double average (double& x1, double& x2, int nvalues=2)
{
  x1 = 7.5;
  return (x1+x2)/nvalues;
}

main()
{
double d1=2.1;
double d2=3.7;
cout << "Modified average: " << average(d1,d2) << endl;
cout << "Half modified average: " << average(d1,d2,4) << endl;
cout << "d1: " << d1 << ", d2: " << d2 << endl;
}
```

C++ basics, Types

- *Types* define what values a variable may obtain, and what operations may be made on the variable.
- Pre-defined C++ types are:

```
signed char
short int
int
unsigned char
unsigned short int
unsigned int
unsigned long int
float
double
long double
```

- User defined types can be defined in *classes*. OpenFOAM provides numerous types that are useful for solving partial differential equations.

C++ basics, Example code

```
#include <iostream> //Just for cout
using namespace std; //Just for cout
#include "tensor.H"
#include "symmTensor.H"
//#include "IOstreams.H" //For Info. Add compilation flags to make it work!
using namespace Foam;

int main()
{   tensor t1(1, 2, 3, 4, 5, 6, 7, 8, 9);
    //Info << "t1: " << t1 << endl; //Add compilation flags!
    //cout << "t1: " << t1 << endl; //Not implemented for cout!
    cout << "t1[0]: " << t1[0] << endl;
    symmTensor st1(1, 2, 3, 4, 5, 6);
    cout << "st1[5]: " << st1[5] << endl;
    return 0;
}
```

Compile with (some trial-and-error, looking at output from wmake for test/tensor):

```
g++ basic.C -DWM_DP -I$WM_PROJECT_DIR/src/OpenFOAM/lnInclude \
            -L $WM_PROJECT_DIR/lib/$WM_OPTIONS/libOpenFOAM.so -o basic
```

C++ basics, scope of variables

- 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 (arguments) is visible in the entire function.
  Other scopes are classes and name spaces, which we will discuss later.
  There can be several variables with the same name, but only one in each block.
  It is possible to use a global variable even if there is a local variable with the same name in the current block. The operator `::` is then used (`::x`)
  Example:

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

C++ basics, Pointers

- Pointers point at a memory location.
- A pointer is recognized by its definition (*):

```
int *pint;
double *pdouble;
char *pchar;
```

- Turbulence models are treated with the `turbulence` pointer in OpenFOAM.
- We will not discuss pointers any further at the moment.

C++ basics, typedef

- Type declarations in C++ may be quite complicated. By using `typedef`, the type can be given a new name, i.e.
  `typedef vector<int> integerVector;`
  An integer vector can then simply be defined 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. **Example code:**

```
#include <iostream>
#include <vector>
using namespace std;
typedef vector<double> doubleVector;

main()
{
vector<double> v2(3);
doubleVector v3(4, 1.5);
cout << "v2: (" << v2[0] << "," << v2[0] << "," << v2[0] << ")" << endl;
cout << "v3: (" << v3[0] << "," << v3[0] << "," << v3[0] << ")" << endl;
}
```

C++ basics, object orientation

- Object orientation focuses on the *objects* instead of the functions.
- The *types* that we have just had a look at are in fact *classes*, and the variables we assign to a *type* are objects of that class.
- 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 of the object and the functions that can manipulate the object. 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.
- 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.
- The classes can be seen as new types that are designed for specific tasks.
- In OpenFOAM, the classes are designed to define, discretize and solve PDE's.

C++ basics, class definition

- The following structure defines the class `name` 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 name {
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++ basics, using a class

- An object of a class `name` is defined in the main code as:
  `name nameObject;` (c.f. `int i`)
- `nameObject` will then have all the attributes defined in the class `name`.
- 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 `name`, it is called in the main code as:
  `nameObject.write();`
- When using the memberfunctions through a pointer, the syntax is slightly different (here `p1` is a pointer to the object `nameObject`, and `p2` is a pointer to a nameless new `name`:

```
p1 = &nameObject;
p2 = new name;
p1->write();
p2->write();
```

C++ basics, member functions

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

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

  where `name::` tells us that the member function `write` belongs to the class `name`., `void` tells us that the function does not return anything, 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 if possible.

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

C++ basics, organization of classes

- A good programming standard is to make the class files in pairs, one with the class definition, and one with the function declarations.
- Classes that are closely related to each other can share files, but keep the class definitions and function declarations separate. This is done throughout OpenFOAM.
- The class definitions must be `include`d in the object file that will use the class, and in the function declarations file. The object file from the compilation of the declaration file is statically or dynamically linked to the executable by the compiler.
- Inline functions must be implemented in the class definition file, since they must be inlined without looking at the function declaration 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`

C++ basics, Constructors

- 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 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++ basics, 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++ basics, Constructors

- A copy constructor has a parameter that is a reference to another object of the same class. (`className(const className&);`). 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. There are however exceptions.
- 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++ 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 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++ 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 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++ 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.
- `friends` are defined in the definition of the class using the special word `friend`.

C++ basics, Operators

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

```
new   delete     new[]      delete[]
+     -     *     /     %     ˆ     &     |     ˜
!     =     <     >     +=    -=    *=    /=    %=
ˆ=    &=    |=    <<    >>    >>=   <<=   ==    !=
<=    >=    &&    ||    ++    --    ,     ->*   ->
()    []
```

  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++ basics, 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 data members or member functions.
- Static members do not belong to any particular object, but to a particular class, so they are used as:

  `className::staticFunction(parameters);`

  (They can actually also be used as `object.staticFunction(parameters)`, but that would be a bit mis-leading since nothing happens explicitly to `object`, and that all objects of the class will notice the effect of the `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++ basics, 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 `Vector` is a sub class to `VectorSpace`.
- A member of `newClass` may have the same name as one in `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 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++ basics, 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++ 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 can not be created for an abstract class.
- The OpenFOAM `RASModel` is such an abstract class since it has a number of pure member functions, such as (see `$FOAM_SRC/turbulenceModels/incompressible/RAS/RASModel/RASModel.H`)

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

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

C++ basics, 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. Search OpenFOAM for examples of how to use `forAll`.

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 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, 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;
```

  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 starting with:

  ```
  using namespace std;
  ```

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

  ```
  using namespace Foam;
  ```

  to make all declarations in namespace `Foam` visible.

- 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
  }
  ```

  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.