# Object orientation in C++ (and OpenFOAM)

# Object orientation in C++ (and OpenFOAM)

## Prerequisites

- You have a very basic knowledge in C programming, and C syntax (to the point where it is the same as in C++).
- You know what it means to construct an object of a class, and how to call a function of that class in the top-level code (if you know that the function exists).
- You know the difference between a declaration and a definition of a function in the top-level code.
- You know how to compile top-level C++ codes using both `g++` and `wmake` (not necessarily OpenFOAM code).

## Learning outcomes

- You will be familiar with the concepts of object orientation in C++
- You will be able to read and understand most features of OpenFOAM classes.
- You will be able to figure out how OpenFOAM classes are related.

Note that you will be asked to pack up your final cleaned-up directories and submit them for assessment of completion.

# 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
- A link:
  - `https://www.geeksforgeeks.org/c-plus-plus/`

# C++ classes and objects

- C++ code can define *classes*, and the 'variables' we assign to a class are *objects* of that class.
- Object orientation focuses on those *objects*.
- An *object* thus 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)
- 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 declaration and definition

- The following structure defines the class with name `myClass` and its public and private member functions and member data.

```
class myClass {
public:
    //declarations of public member functions and member data
protected:
    //declarations of protected member functions and member data
private:
    //declarations of private member functions and member data
};
```

- `public` attributes are visible from outside the class.
- `protected` attributes are visible in the class and sub-classes.
- `private` attributes are only visible in the class.
- If neither `public, procected, private` are specified, all attributes will be `private`.
- Declarations of member functions and member data inside the class are done just as functions and variables are declared in a top-level code.

# C++ class declaration and definition

Let's start building an application with a class.

Put in `appWithClass.C`:

```
class myClass
{
private:
protected:
public:
};

int main()
{
    myClass myClassObject;
    return 0;
}
```

The necessary `main()` function knows about a class named `myClass`, and it is thus possible to construct an object of that class.

Compile and run with (`rm` so any error messages can be seen when the output grows):

```
rm appWithClass; g++ -o appWithClass appWithClass.C; ./appWithClass
```

# C++ class declaration and definition

We will now add a member data to the class and write it out in two ways. For this we need the `<iostream>` library, in `namespace std`. Add the lines in red:

```
#include <iostream> //Just for cout
using namespace std; //Just for cout
class myClass
{
private:
protected:
public:
    int i_=19; //Member data (underscore is OpenFOAM convention)
};

int main()
{
    myClass myClassObject;
    cout << "myClassObject.i_: " << myClassObject.i_ << endl;
    return 0;
}
```

Try moving the member data to `private` or `protected`, and it will not compile. Why?

# C++ class declaration and definition

When we construct an object of a class we do in fact use a *constructor*. The constructor defines how an object of the class should be constructed, and if/how the member data should be initialized. We haven't declared any constructor in our code, but the compiler has declares one for us. The compiler has also declared a default *destructor* for us, which can be used to specify how an object should be destructed when the lifetime of the object expires. However, it is better to explicitly state the constructors and destructors.

Add a constructor and destructor below the declaration of `i_` in the `public` part of the class:

```
myClass()
{
    cout<< "i_ = " << i_ << endl;
};
~myClass()
{};
```

You can see that these are functions, with definitions in the curly brackets. They have the same name as the class, which makes them constructors.

The tilde (~) makes the second one a destructor. Read more about destructors at:
`https://en.cppreference.com/w/cpp/language/destructor`

# C++ class declaration and definition

At the moment the member data `i_` is given its value at declaration. However, it can also be given at definition/initialization.

Change from

```
int i_=19; //Member data (underscore is OpenFOAM convention)
myClass()
{
```

to

```
int i_; //Member data (underscore is OpenFOAM convention)
float j_;
myClass()
:
i_(20),
j_(21.5)
{
```

We have here as well added one more member data, which is a `float`.

The lines between `:` and `{` is the initialization. The member data should be initialized in the same order as in the declaration, in a comma-separated list

# C++ class declaration and definition

The classes are most of the time implemented in libraries that are linked to at compile-time. The top-level code needs to know only the class declaration. The class definition is compiled into the library only. We start by separating the definition from the declaration:

```
class myClass
{
private:
protected:
public:
    int i_; //Member data (underscore is OpenFOAM convention)
    float j_;
    myClass();
    ~myClass();
};

myClass::myClass()
:
i_(20),
j_(21.5)
{
    cout<< "i_ = " << i_ << endl;
}
myClass::~myClass()
{}
```

We see that the definitions of the constructors are here outside the declaration, and we thus have to state the class name when they are defined (`myClass::`). We note again that the function name is the same as the class name for constructors and destructors.

# Use of classes through objects

Change the `main()` function to the following, and spend some time to understand what is happening:

```
int main()
{
    myClass myClassObject;
    cout<< "myClassObject.i_: " << myClassObject.i_ << endl;
    cout<< "myClassObject.j_: " << myClassObject.j_ << endl;
    myClass myClassObject2;
    cout<< "myClassObject2.i_: " << myClassObject2.i_ << endl;
    myClassObject2.i_=30;
    cout<< "myClassObject.i_: " << myClassObject.i_ << endl;
    cout<< "myClassObject2.i_: " << myClassObject2.i_ << endl;
    cout<< "========================================" << endl;
    return 0;
}
```

The last line is a delimiter so that it is easier to see the next section of output. Such a delimiter should be added manually after each addition to the `main()` function in the rest of these slides.

# References

There may be references to any object. Add before the `return` statement:

```
myClass& myClassObjectRef = myClassObject;
cout<< "myClassObjectRef.i_: " << myClassObjectRef.i_ << endl;
myClassObject.i_=42;
cout<< "myClassObject.i_: " << myClassObject.i_ << endl;
cout<< "myClassObjectRef.i_: " << myClassObjectRef.i_ << endl;
myClassObjectRef.i_=43;
cout<< "myClassObject.i_: " << myClassObject.i_ << endl;
cout<< "myClassObjectRef.i_: " << myClassObjectRef.i_ << endl;
```

The `&` at construction (first line) states that the object is a reference.

Compile, run, and understand how references work. A reference *refers* to an object of the same class. It is kind of an *alias*.

# Pointers (1/2)

There may be pointers to any object. Add before the `return` statement:

```
myClass* myClassObjectPntr = &myClassObject;
cout<< "myClassObjectPntr->i_: " << myClassObjectPntr->i_ << endl;
myClass* myClassPntr = new myClass;
cout<< "myClassPntr->i_: " << myClassPntr->i_ << endl;
myClassObjectPntr->i_=3;
myClassPntr->i_=4;
cout<< "myClassObjectPntr->i_: " << myClassObjectPntr->i_ << endl;
cout<< "myClassPntr->i_: " << myClassPntr->i_ << endl;
myClass* generalPntr;
generalPntr = &myClassObject;
cout<< "generalPntr->i_: " << generalPntr->i_ << endl;
generalPntr = &myClassObject2;
cout<< "generalPntr->i_: " << generalPntr->i_ << endl;
```

- The `*` after the class name says that a pointer should be constructed.
- The first pointer is constructed using a reference to the object myClassObject+ (the `&` sign). The second pointer is constructed using the class itself (i.e. there is no object, only a memory location that allocates a `new` object!!!). Note in the output that the constructor of the class is called!!! The third pointer is just constructed as a pointer of the class `myClass`, without pointing at anything. Note that the constructor is not called!!! It is then used to show that pointers can be made to point at different objects of that class.

# Pointers (2/2)

The code again, for reference:

```
myClass* myClassObjectPntr = &myClassObject;
cout<< "myClassObjectPntr->i_: " << myClassObjectPntr->i_ << endl;
myClass* myClassPntr = new myClass;
cout<< "myClassPntr->i_: " << myClassPntr->i_ << endl;
myClassObjectPntr->i_=3;
myClassPntr->i_=4;
cout<< "myClassObjectPntr->i_: " << myClassObjectPntr->i_ << endl;
cout<< "myClassPntr->i_: " << myClassPntr->i_ << endl;
myClass* generalPntr;
generalPntr = &myClassObject;
cout<< "generalPntr->i_: " << generalPntr->i_ << endl;
generalPntr = &myClassObject2;
cout<< "generalPntr->i_: " << generalPntr->i_ << endl;
```

- We see that we have to use -> to call a member data (or functions) through a pointer.
- Pointers are often confused with references. A pointer holds a memory address of another object. A reference can be seen as an alias of another object. A pointer can be re-assigned, while a reference must be assigned at initiation.
- OpenFOAM uses the pointer functionality to make run-time choises possible, such as the choise of turbulence model.

# Static members (data and functions)

- 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.

Add after the declaration of `j_` (under `public`):

```
static int m_;
```

Add before `main()` (it is not allowed to set the value inside the class - why?):

```
int myClass::m_ = 9;
```

Add before `return 0;`:

```
cout<< "myClassObject.m_: " << myClassObject.m_ << endl;
cout<< "myClassObject2.m_: " << myClassObject2.m_ << endl;
myClass::m_=30; //Or: myClassObject2.m_=30;
cout<< "myClassObject.m_: " << myClassObject.m_ << endl;
cout<< "myClassObject2.m_: " << myClassObject2.m_ << endl;
```

We see that all objects are affected by the center line. It would be less clear to write the alternative that is commented at the center line (why?), although it will have the same effect!

# Member functions

Until now our class only has member data (except for the constructor and destrucor, which are in fact special member functions). Add the declaration of a member function in the public part of the declaration (after the destructor):

```
    void write();
```

Add the definition of the member function (after defining the destructor):

```
void myClass::write()
{
    cout<< "My member data i_ = " << i_ << endl;
    cout<< "My member data j_ = " << j_ << endl;
}
```

Add a call to the function, before `return 0;` in the `main()` function:

```
    myClassObject.write();
    myClassObject2.write();
    generalPntr->write(); //Use "->" for pointers!
```

Compile and test.

We see that the member functions have direct access to *all* the member data and member functions of the class (`private`, `protected`, `public`).

# Inlining of member functions

The member functions may be *inline*d, to avoid an overhead in the function call for small functions. 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.

You can try this by adding `inline` at the beginning of the declaration and definition of the `write()` function. Note however that `inline` functions must be both *declared and defined* in the header file when doing it the OpenFOAM way (why?).

# Constant member functions

An *object of a class* can be constant (`const`). Such objects are only allowed to call *constant member functions*, that promise not to change the object itself (the values of its member data). Some member functions might in fact not change the object, but we need to tell the compiler that it doesn't. That is done by adding `const` after the parameter list in the function declaration/definition. Then the function can be used for constant objects.

Add before `return 0;`:

```
const myClass myClassObject3;
myClassObject3.write();
```

Try to compile and see it fail, since the `write()` function is not a constant function (although it does not change the object). Make it work by changing the function declaration/definition to:

```
inline void write() const;
...
inline void myClass::write() const
```

# Reference to member data through member functions (1/2)

We have previously seen that *private* member data can only be used inside the class. However, *public* member functions can be used to make *private* member data available outside the class.

Add after `private:`

```
    int k_=7;
```

Add after the declaration of the `write()` function (note the `&` sign):

```
    int& k();
```

Add after the definition of the `write()` function (note the `&` sign):

```
int& myClass::k()
{
    return k_;
}
```

Add before the `return 0;` statement (note the `&` sign):

```
    int& k=myClassObject.k();
    cout<< "k: " << k << endl;
    k=5; //Or directly: myClassObject.k()=5;
    cout<< "k: " << k << endl;
    cout<< "myClassObject.k(): " << myClassObject.k() << endl;
```

# Reference to member data through member functions (2/2)

We might only want to make the member data available, but not modifiable.

Add after the `k_` member data under `private:`

```
    const int l_=17;
```

Add after the declaration of the `k()` function:

```
    const int& l();
```

Add after the definition of the `k()` function:

```
const int& myClass::l()
{
    return l_;
}
```

Add before the `return 0;` statement:

```
    const int& l=myClassObject.l();
    cout<< "l: " << l << endl;
```

Note that the `const` has its origin in the declaration of the member data, so it can't be removed.
Since the function can't change the member data, we might consider making it constant:
Declaration: `const int& l() const;`
Definition: `const int& myClass::l() const`

# More about constructors - special member functions

Constructors are special member functions that are only used when an object is constructed. They are named the same as the class itself. As for any function there may be many constructors, depending on how an object of the class can be constructed. Let's add a constructor for which we supply the values of the member data. After the declaration of the constructor, add:

```
    myClass(int,int);
```

After the definition of the constructor, add:

```
myClass::myClass(int i, int j)
:
i_(i),
j_(j)
{
    cout<< "i_ = " << i_ << endl;
    cout<< "j_ = " << j_ << endl;
}
```

Before the `return` statement, add:

```
    myClass myClassObject4(23,56);
```

There can of course only be one constructor of `myClass` that takes two integer arguments.

# Operators

At the moment we can't say `myClass mySum=myClassObject+myClassObject2;`, since it is not obvious how the summation should be done.

- Operators define how to manipulate objects (such as to do the above summation).
- 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 `>>`

We will introduce a summation operator...

# Operators

Add before `main()`:

```
inline myClass operator+(const myClass& mC1, const myClass& mC2)
{
    return myClass
    (
        mC1.i_ + mC2.i_,
        mC1.j_ + mC2.j_
    );
}
```

Add before `return 0;`:

```
    myClassObject.write();
    myClassObject2.write();
    myClass mySum=myClassObject+myClassObject2;
    mySum.write();
```

We note here, and from the output, that we use the constructor that takes two integer arguments.

# Inheritance

- A class can inherit attributes from already existing classes, and extend with new attributes.
- Syntax, when defining the new class:

```
class subClass
: public baseClass
{ ...subClass attributes... }
```

  where `subClass` will inherit all the attributes from `baseClass`.
  `subClass` is now a *sub-class* to `baseClass`.
  We will discuss `public` later.

- An attribute of `subClass` may have the same name as one in `baseClass`. Then the `subClass` attribute will be used for `subClass` objects and the `baseClass` 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.

# Inheritance

Add before main:

```
class mySubClass
: public myClass
{
public:
        mySubClass();
        mySubClass(int,int);
};
mySubClass::mySubClass()
:
myClass()
{}
mySubClass::mySubClass(int i, int j)
:
myClass(i,j)
{}
```

As can be seen, it is necessary to redefine the constructors and call the base class constructors for their initializations.

Continued...

# Inheritance

Add before `return 0;`:

```
mySubClass mySubClassObject(45,90);
cout<< "mySubClassObject.i_: " << mySubClassObject.i_ << endl;
cout<< "mySubClassObject.j_: " << mySubClassObject.j_ << endl;
mySubClassObject.write();
```

You should now be able to change the class of your first `myClassObject` to `mySubClass`, and the constructors, member data, member functions, references and pointers should work as before.

You can then add *additional* attributes to the sub class.

The sub classes are in OpenFOAM implemented in a separate directories, and thus in separate files. The sub class thus needs to `#include myClass.H` in its header of the `*.H` file.

Note that you can't do

```
cout<< "mySubClassObject.k_: " << mySubClassObject.k_ << endl;
```

since `k_` is `private` in the base class and not visible in the sub class. See next slide...

# Inheritance: visibility and multiple inheritance

Some general information about inheritance and visibility, which you can test yourself if you like:

- A member that is redefined in a sub class will hide the corresponding member from the base class. *All* the alternatives for a function with a specific name from the base class will be hidden in the sub class if one or more of them is redefined in the sub class.
- A hidden member of a base class can be reached in the sub class by `baseClass::member`
- Members of a class can (as known) 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 or top-level code).
  - The visibility of the members inherited from a base class to a sub class can be stated/-modified in the sub class using the reserved words `public`, `private` or `protected` when declaring the class. (`public` in the previous example). It is only possible to make each member 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!

# 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. It is not mutal.

Add at the end of `public` in the declaration of `myClass`:

```
friend class myClassFriend;
```

Remember that `l_` is private data of `myClass`, and add before `main()`:

```
class myClassFriend
{
public:
        myClassFriend();
};
myClassFriend::myClassFriend()
{
    myClass testMyClassFriend;
    cout << "testMyClassFriend.l_ = " << testMyClassFriend.l_ << endl;
}
```

Add before `return 0;`:

```
myClassFriend myClassFriendObject;
```

# Templates

The most obvious way to define a class (or function) 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. OpenFOAM convention is that templated classes (and their file and directory names) start with a capital letter.

Add before `main()`:

```
template<typename T>
class MyTemplatedClass
{
public:
        T x_;
        MyTemplatedClass(T);
};
template<typename T>
MyTemplatedClass<T>::MyTemplatedClass(T x)
:
x_(x)
{cout << "x_ = " << x_ << endl;}
```

Add before `return 0;`:

```
    MyTemplatedClass<int> myTemplatedClassIntObject(4.6);
    cout<< "myTemplatedClassIntObject.x_: " << myTemplatedClassIntObject.x_ << endl;
    MyTemplatedClass<float> myTemplatedClassFloatObject(4.6);
    cout<< "myTemplatedClassFloatObject.x_: " << myTemplatedClassFloatObject.x_ << endl;
```

# Typedef

- OpenFOAM is full of templates, and sometimes several layers of templates.
- A code can be easier to read if e.g. complex templates are renamed with `typedef`. Add before `main()`:

```
typedef MyTemplatedClass<int> myTemplInt;
typedef MyTemplatedClass<float> myTemplFloat;
```

Add before `return 0;`:

```
myTemplInt myTemplIntObject(4.6);
cout<< "myTemplIntObject.x_: " << myTemplIntObject.x_ << endl;
myTemplFloat myTemplFloatObject(4.6);
cout<< "myTemplFloatObject.x_: " << myTemplFloatObject.x_ << endl;
```

# Virtual member functions (1/4)

- Virtual member functions are used for dynamic binding, i.e. the function will work differently depending on how it is called, and it can be determined at run-time using pointers.
- The reserved word `virtual` is used in front of the member function declaration to declare it as virtual.
- Variants of the virtual member function can be realized by sub classes that re-implement the member function. The sub class member functions corresponding to the base class virtual 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 of the base class, and the virtual functions will operate according to that specific sub class.
- The difference compared to a standard sub class is that for a standard sub class an object must be constructed for *either* the base class *or* the sub class. Then the object will always correspond to that class. When using virtual functions, the pointer can be constructed for the base class and point at one of the sub classes. It is also possible to dynamically change so that it points at any of the other sub classes.

# Virtual member functions (2/4)

- Let us modify the `write()` function of the original class `myClass`. Just add the word `virtual` at the beginning of the declaration of the `write()` function. It is also necessary to add that word to the destructor, i.e.:

```
...
virtual ~myClass();
...
virtual inline void write() const;
...
```

- Compile and run, and you see that the class works as before. But we now have a new option...

# Virtual member functions (3/4)

- Add two sub classes that re-implement the `write()` function (before the `main()` function):

```
class myClassVirtualInstance1
: public myClass
{
public:
    inline void write() const;
};
inline void myClassVirtualInstance1::write() const
{
    cout<< "In myClassVirtualInstance1" << endl;
}
class myClassVirtualInstance2
: public myClass
{
public:
    inline void write() const;
};
inline void myClassVirtualInstance2::write() const
{
    cout<< "In myClassVirtualInstance2" << endl;
}
```

# Virtual member functions (4/4)

- Add before `return 0;`:

```
myClassVirtualInstance1 myClassVirtualInstance1Object;
myClassVirtualInstance2 myClassVirtualInstance2Object;
myClass* myClassPtr;
myClassPtr = &myClassObject;
myClassPtr->write();
myClassPtr = &myClassVirtualInstance1Object;
myClassPtr->write();
myClassPtr = &myClassVirtualInstance2Object;
myClassPtr->write();
```

- Compile, run, and check that the output from `write()` is given correctly.
- You can see that the other member functions work as for the base class.
- Try to remove the word `virtual` in the base class, and see that the dynamic binding is broken - i.e. the base class `write()` function is executed. This means that the base class determines which functions are virtual and can be modified through pointers.
- The objects constructed at the first rows above belong to the sub-classes. They work the same both with and without the word `virtual` in the base class. Check yourself!
- Note that the base class works as before (it is still being used by objects in our code).

# Abstract classes (1/3)

- A class with at least one virtual member function that is undefined (a *pure* virtual function) is an abstract class.
- The main difference from the discussion on virtual member functions is that an object can not be created for an abstract class. It is used in cases where the base class has no function itself, enforcing the use of one of the sub classes.
- The OpenFOAM `turbulenceModel` is such an abstract class since one must specify *which* turbulence model. It has a number of pure member functions, such as (see `turbulenceModel.H`)

```
//- Solve the turbulence equations and correct the turbulence viscosity
virtual void correct() = 0;
```

(you see that it is *pure* virtual by '`= 0`', and that the definition does not evaluate any specific turbulence model).

- A `turbulence` pointer of the class `turbulenceModel` is constructed in e.g.: `$FOAM_SOLVERS/incompressible/pimpleFoam/createFields.H`
- The `correct()` function is called through the pointer, in e.g.: `$FOAM_SOLVERS/incompressible/pimpleFoam/pimpleFoam.C`: `turbulence->correct();`

# Abstract classes (2/3)

- We can't change `myClass` to an abstract class, since we construct objects of that class in our code. Verify this by changing in the declaration:

```
virtual inline void write() const = 0;
```

- Create a new abstract class, with a sub class (before `main()`):

```
class myAbstractClass
{
public:
    virtual ~myAbstractClass();
    virtual inline void write() const = 0;
};
myAbstractClass::~myAbstractClass()
{}
class myAbstractClassInstance1
: public myAbstractClass
{
public:
    inline void write() const;
};
inline void myAbstractClassInstance1::write() const
{
    cout<< "In myAbstractClassInstance1" << endl;;
}
```

# Abstract classes (3/3)

- Add before `return 0;`:

```
myAbstractClass* myAbstractClassPtr = new myAbstractClassInstance1;
myAbstractClassPtr->write();
```

- Note that you can't create an object of the base class (try before `return 0;`):

```
myAbstractClass myAbstractClassObject;
```

The abstract class does not have any definition of the function `write()`, so it can't work.

# End of example implementations

Here we stop implementing examples.

Remember that nothing we have implemented in this single-file code is OpenFOAM. You should still be able to compile it with:

```
g++ -o appWithClass appWithClass.C
```

and run it with:

```
./appWithClass
```

We will in the following few slides only discuss some more advanced aspects in general terms, which we will see later in OpenFOAM.

# Container classes

- 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 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; }
```

# Forward declaration

We know that we introduce header files to let the compiler know about classes, functions, etc. that the code in a particular class needs to use. In some cases that is not possible. For instance, if two classes are mutual friends they both need to know about each other at compile time. That is not possible, since the compilation goes sequentially through the files. As one of the classes is declared, the other class has not yet been declared. The solution is to do a forward declaration.

A forward declaration tells the compiler about the existence of an identifier before declaring the identifier.

Forward declarations can also potentially speed up the compilation speed, since the compiler does not have to look up the other class. On the other hand, the class identifier must be repeated wherever it will be used.

In OpenFOAM you see this before the class declaration, inside `namespace Foam`, looking like class, function, or operator declarations.

Read more: `https://www.geeksforgeeks.org/what-are-forward-declarations-in-c/`

# Structure vs. class

A structure (`struct`) is the same as a `class`, except for the default visibility of the attributes. A `struct` by default has `public` attributes, while a class by default has `private` attributes.

Structures are used mostly to hold collections of data, and to access the data through the `public` member data (e.g. `myStructObject.i_`).

We can in fact change all `class` to `struct` in the code we developed in these slides.

Read more at:

`https://www.geeksforgeeks.org/structure-vs-class-in-cpp/`
`https://en.cppreference.com/w/c/language/struct`

# Error messages

List of common error messages, to be extended:

- Comment the definition of myClass::write(), and you get a common error message:

  ```
  myClassAppWithClass.C:(.text.startup+0x3b5): undefined reference to `myClass::write()'
  ```

  You get a similar message when linking to a library with pre-compiled definitions that do not correspond to the declarations in your code.