Implement class in library (as in OpenFOAM)

Implement class in library (as in OpenFOAM)

Prerequisites

- You have a basic knowledge in object oriented C++ programming.
- You know how to implement and use a class in a single file.
- You know how to compile top-level C++ codes using both `g++` and `wmake` (not necessarily OpenFOAM code).

Learning outcomes

- You will be able to separate classes into separate directories and files, as it is done in OpenFOAM.
- You will know how to compile classes in separate directories and files both "hard-coded" into an application and as a dynamic library that can be linked to several applications.

Note that there is an extended task related to these procedures!

Create directories for the development

Create a directory for the development (same name as the top-level application):

```
mkdir appWithClassInLibrary
cd appWithClassInLibrary
```

Create a directory for the class (same name as the class):

```
mkdir myClass
cd myClass
```

Class declaration file

Classes should be coded in pairs of files, with a `*.H` file that contains the class *declaration* and a `*.C` file that contains the class *definition*.

Put the class *declaration* in `myClass.H`:

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

Here we have added `#ifndef/#define/#endif` to make sure that the class is not declared multiple times, in the case that we `#include "myClass.H"` several times.

Class definition file

Put the class *definition* in `myClass.C`:

```
#include "myClass.H"
#include <iostream> //Just for cout
using namespace std; //Just for cout
myClass::myClass()
:
i_(20),
j_(21.5)
{
    cout<< "i_ = " << i_ << endl;
}
myClass::~myClass()
{}
```

The first line above is necessary since we will compile only the `*.C` file of the class, and it needs to know about its own declaration.
We have also added two lines at the header for the `cout` output stream that is used in the class, since this file is now compiled separate to the top-level code (where it is also stated).

Top-level application file

Go to the application directory:
`cd ..`
Put in `appWithClassInLibrary.C` (same name as directory, as OpenFOAM convention):

```
#include <iostream> //Just for cout
using namespace std; //Just for cout
#include "myClass.H"

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;
    return 0;
}
```

The top-level solver needs to know about the class declaration (only), which is why we have added `#include "myClass.H"`. All pieces of code that use a class need to include the declaration of that class!

Create Make/files,option and compile

Put in `Make/files`:

```
myClass/myClass.C
appWithClassInLibrary.C
EXE = $(FOAM_USER_APPBIN)/appWithClassInLibrary
```

Put in `Make/options`:

```
EXE_INC = \
    -ImyClass
```

Compile with `wmake`, and realize that the compilation is done in three steps. The first is for `myClass.C`, generating the object file `myClass.o`. The second is for `appWithClassInLibrary.C`, generating `appWithClassInLibrary.o`. The third is for the linking, including the class compiled right now and any dynamic linking.

Make sure that the application runs and gives the same output as before!

Separate class to dynamic library (1/2)

At this point we have hard-coded a class into the top-level application. You do not see the `myClass` class when you issue the command

```
ldd `which appWithClassInLibrary`
```

Most of the time the classes are dynamically linked through libraries to the top-level applications. We will therefore put our class in a library and dynamically link to that library.

Start by changing the `Make/files` file to:

```
appWithClassInLibrary.C
EXE = $(FOAM_USER_APPBIN)/appWithClassInLibrary
```

Notice that after doing a `wclean` (necessary, since `wmake` does not realize that the `Make/files` file has changed), the `wmake` command will fail. The reason is that the top-level application can not link to the class, at the second stage of compilation. The application still has access to the class *declaration* (we still include the `myClass.H` file), but it no longer has access to the class *definition*. We will make sure that it can link to it dynamically instead.

Typical error message when a class *declaration* can't be found:

```
...: error: 'myClass' was not declared in this scope
```

Typical error message when a class *definition* can't be found:

```
...: undefined reference to `myClass::~myClass()'
```

Separate class to dynamic library (2/2)

Create a directory for the library and move the class directory there:

```
mkdir myLibrary
mv myClass myLibrary
```

Create a `myLibrary/Make/files` file with:

```
myClass/myClass.C
LIB = $(FOAM_USER_LIBBIN)/libmyLibrary
```

Create an empty `myLibrary/Make/options` file.

Compile the library separately:

```
wmake myLibrary
```

Modify the `Make/options` file of the top-level application to:

```
EXE_INC = \
    -ImyLibrary/lnInclude
EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
    -lmyLibrary
```

Compile and test (you probably have to first do `wclean` to remove the `*.dep` file).

Dynamic linking

We will now check the dynamic linking:

`ldd \`which appWithClassInLibrary\`` shows that the file `libmyLibrary.so` is dynamically linked to.

It is found since it is located in a directory that we pointed at during compilation (in `Make/options`: `-L$(FOAM_USER_LIBBIN) -lmyLibrary`)

Try renaming the file:

```
mv $FOAM_USER_LIBBIN/libmyLibrary.so $FOAM_USER_LIBBIN/libmyLibrary.so_tmp
```

`ldd \`which appWithClassInLibrary\`` now shows `libmyLibrary.so => not found`

`appWithClassInLibrary` gives:

```
appWithClassInLibrary: error while loading shared libraries:
libmyLibrary.so: cannot open shared object file: No such file or directory
```

Rename back:

```
mv $FOAM_USER_LIBBIN/libmyLibrary.so_tmp $FOAM_USER_LIBBIN/libmyLibrary.so
```

I.e., the compiled application only knows about the class *declaration*, but not the class *definition*. The class *definition* is provided by the compiled library that the solver links to at run-time.

Alignment with OpenFOAM directory organization

At this point it is just a matter of directory and file organization to move the library to an appropriate location in `$WM_PROJECT_USER_DIR/src`, and the application to an appropriate location in `$WM_PROJECT_USER_DIR/applications`.

The `Make/options` file of the application just has to be updated according to the location of the library header file.

Coding style

We have not bothered about indentation etc. Please read about (and apply) this at:

`https://openfoam.org/dev/coding-style-guide/`

A warning!

It is quite common to make backups of files when developing code. However, a warning is issued when developing OpenFOAM libraries:

All the files in the directory structure of a library will be linked to in the `lnInclude` directory, so if you put backup files in a backup directory in the directory structure of that library they will also end up linked to in `lnInclude`. It may be ok if they are not used, but it may also be dangerous if you use the same file names as the original files. The linking in `lnInclude` will only be done to one of the files with a particular name (the first or the last it finds, depending on how the linking is set up). If an active header file is changed during implementation it may mean that the old back-up header file is included in all files that depend on it. That may lead to mysterious problems.

If you have to make backups it is better to make tar archives of the backup directories. Then the original files are hidden in the archives. Just remember to remove the directory that you put in the tar archive, so that the files are not linked to!