Library and class organization in OpenFOAM

Library and class organization in OpenFOAM

Prerequisites

- You have a basic knowledge in object oriented C++ programming.
- 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 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 there is an extended task related to these procedures!

Library and class organization in OpenFOAM

We just went through many aspecs of object orientation in C++, in a single file.
You will be asked to set up the same thing the OpenFOAM way, using separate directories and files for the class(es).
Now we will go through the basic procedure of doing that.
We will do this in a new directory:

```
cd ..
mkdir myClassApp
cd myClassApp
```

Library and class organization in OpenFOAM

We know from OpenFOAM that classes are 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.

Library and class organization in OpenFOAM

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).

Library and class organization in OpenFOAM

Put in `myClassApp.C`:

```
#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!

Library and class organization in OpenFOAM

Put in `Make/files`:

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

Create an empty `Make/options` file.

Compile with `wmake`, and realize that there are three `gcc` lines in the output. The first is for `myClass.C`, generating the object file `myClass.o`. The second is for `myClassApp.C`, generating `myClassApp.o`. The third is for the linking, including the class compiled right now and any dynamic linking.

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

Library and class organization in OpenFOAM

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

```
ldd `which myClassApp`
```

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

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

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

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 solver can not link to the class, at the second stage of compilation. We will make sure that it can link to it dynamically instead.

Library and class organization in OpenFOAM

Create a directory `myClass`, and put the `myClass.*` files in that directory:

```
mkdir myClass
mv myClass.* myClass
```

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

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

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

Compile the library separately:

```
wmake myClass
```

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

```
EXE_INC = \
    -ImyClass/lnInclude
EXE_LIBS = \
    -L$(FOAM_USER_LIBBIN) \
    -lmyClass
```

Compile, test, and see that ``ldd `which myClassApp` `` shows that the file `libmyClass.so` is now dynamically linked to.

Library and class organization in OpenFOAM

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.

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 mysterios 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!