051176 - Computational Techniques for Thermochemical Propulsion
Master of Science in Aeronautical Engineering

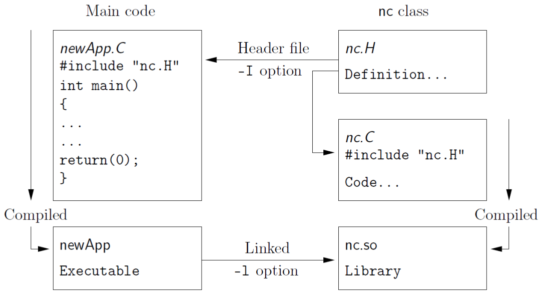
Compiling applications and libraries

Prof. **Federico Piscaglia**
Dept. of Aerospace Science and Technology (DAER)
Politecnico di Milano - Italy

federico.piscaglia@polimi.it

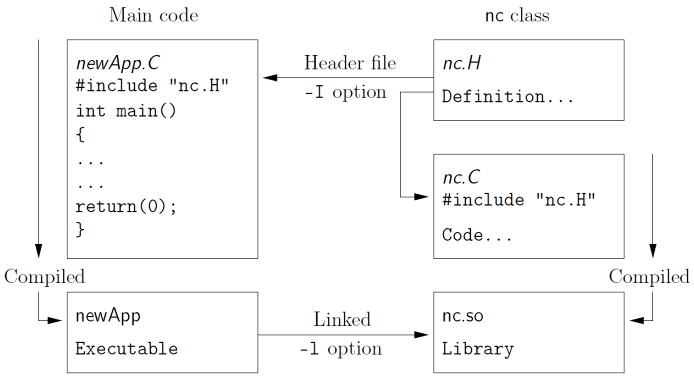
Compiling applications and libraries

Compilation is an integral part of application development that requires careful management since every piece of code requires its own set instructions to access dependent components of the OpenFOAM library. In `UNIX/Linux` systems these instructions are often organised and delivered to the compiler using the standard `UNIXmake` utility.



OpenFOAM uses its own `wmake` compilation script that is based on `GNU make` but is considerably easier to use. The application `wmake` can be used on any code, not only the OpenFOAM library.

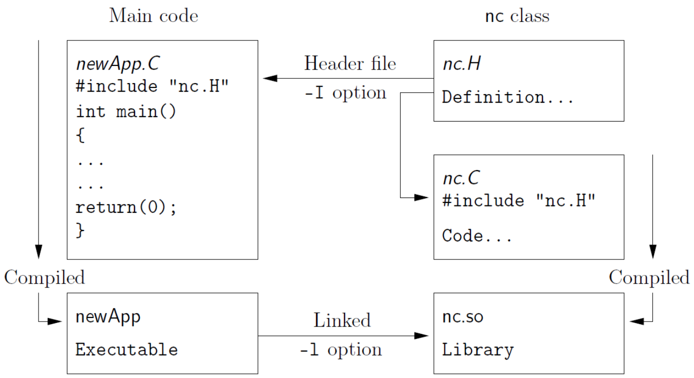
Compiling applications and libraries in OF



Source:https://cfd.direct/openfoam/user-guide/compiling-applications/

- A class is defined through a set of instructions such as object construction, data storage and class member functions. The file that defines these functions — the class definition — takes a .C extension, e.g. a class nc would be written in the file `nc.C`.

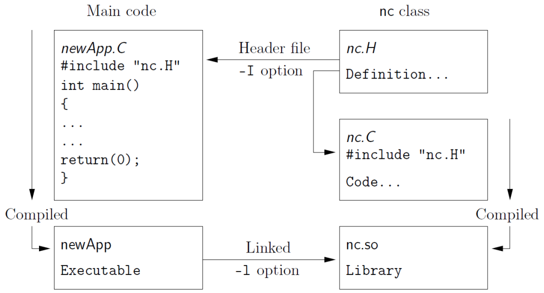
Compiling applications and libraries in OF

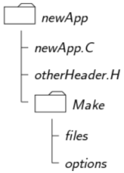


Source:https://cfd.direct/openfoam/user-guide/compiling-applications/

- This file can be compiled independently of other code into a binary executable library file known as a shared object library with the .so file extension, i.e. nc.so. When compiling a piece of code, say `newApp.C`, that uses the nc class, `nc.C` need not be recompiled, rather `newApp.C` calls the `nc.so` library at runtime. This is known as `dynamic linking`.

Header .H files

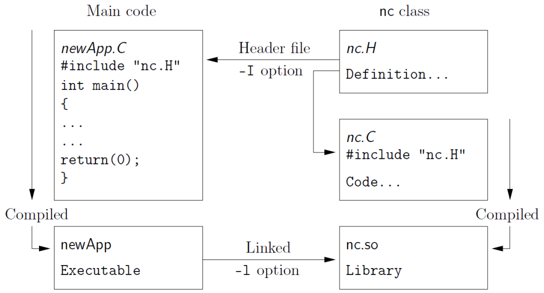


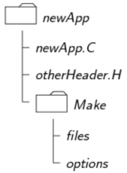


Source:https://cfd.direct/openfoam/user-guide/compiling-applications/

- As a means of checking errors, the piece of code being compiled must know that the classes it uses and the operations they perform actually exist.
- Each class requires a class declaration, contained in a header file with a .H file extension that includes the names of the class and its functions. This file is included at the beginning of any piece of code using the class, using the #include directive, including the class declaration code itself.

Header .H files





Source:https://cfd.direct/openfoam/user-guide/compiling-applications/

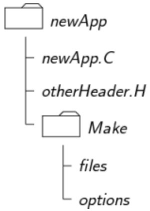
- Any piece of .C code can resource any number of classes and must begin by including all the .H files required to declare these classes. Those classes in turn can resource other classes and so also begin by including the relevant .H files, that are known as the dependencies;
- by searching recursively down the class hierarchy we can produce a complete list of header files for all the classes on which the top level .C code ultimately depends.

OpenFOAM: Compilation and Linking

Header files are included in the code using the `#include` directive:

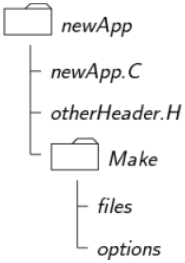
```
# include "otherHeader.H";
```

This causes the compiler to suspend reading from the current file, to read the included file. This mechanism allows any self-contained piece of code to be put into a header file and included at the relevant location in the main code in order to improve code readability.



EXAMPLE: in most OpenFOAM applications, the code for creating fields and reading field input data is included in a file `createFields.H` which is called at the beginning of the code. In this way, header files are not solely used as class declarations.

Compiling with `wmake`



- OpenFOAM applications are organised using a standard convention that the source code of each application is placed in a directory whose name is that of the application. The top level source file then takes the application name with the .C extension.
- For example, the source code for an application called newApp would reside is a directory newApp and the top level file would be `newApp.C`.
- `wmake` then requires the directory must contain a Make subdirectory containing 2 files, `options` and `files`.

Compiling with `wmake`

The compiler searches for the included header files in the following order, specified with the `-I` option in wmake:

- the `$WM_PROJECT_DIR/src/OpenFOAM/lnInclude` directory;
- a local `lnInclude` directory, i.e. `newApp/lnInclude`;
- the local directory, i.e. `newApp`;
- platform dependent paths set in files in the

  `$WM_PROJECT_DIR/wmake/rules/$WM_ARCH/`

  directory, e.g. `/usr/X11/include` and `$(MPICH_ARCH_PATH)/include`; other directories specified explicitly in the `Make/options` file with the `-I` option.

The `Make/options` file contains the full directory paths to locate header files using the syntax:

```
EXE_INC = \
        -I<directoryPath1> \
        -I<directoryPath2> \
        ...                \
        -I<directoryPathN>
```

Note that the directory names are preceeded by the `-I` flag and that the syntax uses the \ to continue the `EXE_INC` across several lines, with no \ after the final entry.

Linking to libraries

The `Make/options` file contains the full directory paths and library names using the syntax:

```
EXE_LIBS =
    -L<libraryPath>
    -l<library1>
    -l<library2>
    ...
    -l<libraryN>
```

The actual library files to be linked must be specified using the `-l` option and removing the lib prefix and .so extension from the library file name, e.g. `libnew.so` is included with the flag `-lnew`.

The compiler links to shared object library files in the following directory paths, specified with the -L option in wmake:

- the `$FOAM_LIBBIN` directory;
- platform dependent paths set in files in the `$WM_DIR/rules/$WM_ARCH/` directory, e.g. `/usr/X11/lib` and `$(MPICH_ARCH_PATH)/lib`;
- other directories specified in the Make/options file.

Linking to libraries

The Make/options file contains the full directory paths and library names using the syntax:

```
EXE_LIBS =
    -L<libraryPath>
    -l<library1>
    -l<library2>
    ...
    -l<libraryN>
```

The actual library files to be linked must be specified using the -l option and removing the lib prefix and .so extension from the library file name, e.g. libnew.so is included with the flag -lnew.

By default, wmake loads the following libraries:

- the libOpenFOAM.so library from the $FOAM_LIBBIN directory;
- platform dependent libraries specified in set in files in the folder $WM_DIR/rules/$WM_ARCH/, e.g. libm.so from /usr/X11/lib and liblam.so from $(LAM_ARCH_PATH)/lib;
- other libraries specified in the Make/options file.

Make/options: **example**

From `$FOAM_SOLVERS/incompressible/pisoFoam/Make/options`

```
EXE_INC = \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/incompressible/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude

EXE_LIBS = \
    -lturbulenceModels \
    -lincompressibleTurbulenceModels \
    -lincompressibleTransportModels \
    -lfiniteVolume \
    -lmeshTools \
    -lfvOptions \
    -lsampling
```

By default, `wmake` loads the following libraries:

- the `libOpenFOAM.so` library from the `$FOAM_LIBBIN` directory;
- platform dependent libraries specified in set in files in the folder `$WM_DIR/rules/$WM_ARCH/`, e.g. `libm.so` from `/usr/X11/lib` and `li-blam.so` from `$(LAM_ARCH_PATH)/lib`;
- other libraries specified in the `Make/options` file.

Make/files

The compiler requires a list of `.C` source files that must be compiled.

- The list must contain the main `.C` file but also any other source files that are created for the specific application but are not included in a class library.
- The full list of `.C` source files must be included in the `Make/files` file. For many applications the list only includes the name of the main `.C` file, e.g. `newApp.C` in the case of our earlier example.

```
    newApp.C
    newFunctionality.C
    anyCodeToCompileUsedByNewApp.C

    EXE = $(FOAM_USER_APPBIN)/newApp
```

In the `Make/files` file, it must be included:

- the full list of `.C` source files;
- a full path and name of the compiled executable, specified by the `EXE =` syntax. Standard convention stipulates the name is that of the application, i.e. newApp in our example.

OpenFOAM offers two useful choices for path: standard release applications are stored in `$FOAM_APPBIN`; applications developed by the user are stored in `$FOAM_USER_APPBIN`.

Source files to be compiled - best practice

If the user is developing his own applications, it is reccomended to:

- create an applications subdirectory in `$WM_PROJECT_USER_DIR` directory with the source code of personal OpenFOAM applications;
- in the `Make/files` file of a user application, the user should specify where the user's executables are written (usually `$FOAM_USER_APPBIN` directory).

The `Make/files` file for our example would appear as follows:

```
newApp.C
EXE = $(FOAM_USER_APP-
BIN)/newApp
```

Running `wmake`

The wmake script is generally executed by typing:

```
wmake <optionalDirectory>
```

The `<optionalDirectory>` is the directory path of the application that is being compiled. Typically, `wmake` is executed from within the directory of the application being compiled, in which case `<optionalDirectory>` can be omitted.

Compiling libraries (**`LIBS`**)

When compiling a library, there are 2 critical differences in the configuration of the file in the `Make` directory:

- in the `files` file, `EXE =` is replaced by `LIB =` and the target directory for the compiled entity changes from `$FOAM_APPBIN` to `$FOAM_LIBBIN` (and an equivalent `$FOAM_USER_LIBBIN` directory); in the options file, `EXE_LIBS =` is replaced by `LIB_LIBS =` to indicate libraries linked to library being compiled.
- when `wmake` is executed it additionally creates a directory named `lnInclude` containing soft links to all the files in the library. The `lnInclude` directory is deleted by the `wclean` script when cleaning library source code.

General information

OpenFOAM is a library of tools, not a monolithic single-executable

- Most changes do not require surgery on the library level: code is developed in local work space for results and custom executables
- Environment variables and library structure control the location of the library, external packages (e.g. `gcc`, `Paraview`) and work space
- For model development, start by copying a model and changing its name: library functionality is unaffected
- Local workspace:
  - **Run directory**: `$FOAM_RUN`
    Ready-to-run cases and results, test loop etc. May contain case-specific setup tools, solvers and utilities.
  - **Local work space**: `$WM_PROJECT_INST_DIR/<userName>-4.x/`
    Contains applications, libraries and personal library and executable space

Debugging OpenFOAM

Build and Debug Libraries

- Release build optimised for speed of execution;
- Debug build provides
  - additional run-time checking and detailed trace-back capability
  - trace-back on failure
  - Once the code is compiled in Debug mode, you can use **`gdb` debugger**

NOTE: to compile the code in Debug mode, you need to set the environment variable `$WM_COMPILE_OPTION=Debug` in `$WM_PROJECT_DIR/etc/bashrc`

Similar tricks for debugging: DEBUG SWITCHES

- Each set of classes or class hierarchy provides its own debug stream
- . . . but complete flow of messages would be overwhelming!
- You can activate switch messages from:
  - `$FOAM_CASE/system/controlDict`
  - `$WM_PROJECT_DIR/etc/controlDict`

myFirstApp.C

Creating your application in OpenFOAM

EXAMPLE:

- We want to write `icoScalarTransportFoam`, an incompressible solver with a scalar transport equation (specie mass fraction, temperature,...)

- To do this, we need to create a new application based on the `icoFoam` code

IMPORTANT NOTES:

- an `application` in OpenFOAM is an executable file and may be:
  - a new solver. Before writing your code, please check what is available in `$FOAM_SOLVERS`.
  - an utility for data pre/post processing or for mesh manipulation. Before writing your code, please check what is available in `$FOAM_UTILITIES`.

- To do this, we need to create a new application based on the `icoFoam` code

Creating your application in OpenFOAM

... **AND MOST IMPORTANTLY**:

- always check if something similar to what you need is already available in the official distribution. Avoid to double the code.

  `More code = more maintainance`

- be careful in programming: object-oriented programming philosophy is based on splitting of different tasks:
  - a solver is demanded only to solve the equations;
  - a `utility/functionObject/application` is performing additional operations and it is NOT included in the solver. It must be LINKED to the solver;
  - physical models must be included in classes and dynamically linked to the solvers.

Creating your application in OpenFOAM

- Find appropriate code in OpenFOAM which is closest to the new use or provides a starting point

- Copy into local work space and rename

- Change file name and location of library/executable: `Make/files`

- Environment variables point to local work space applications and libraries:

```
$FOAM_PROJECT_USER_DIR
$FOAM_USER_APPBIN
$FOAM_USER_LIBBIN
```

- Change the code to fit your needs

Creating your solver in OpenFOAM

- applications are located in `$WM_PROJECT_DIR/applications`:

```
cd $WM_PROJECT_DIR/applications/solvers/incompressible
```

- Copy the `icoFoam` solver and put it in the user application folder:

```
cp -r icoFoam $WM_PROJECT_USER_DIR/applications
```

- Rename the directory and the source file name, clean all the dependancies and:

```
mv icoFoam icoScalarTransportFoam
cd icoScalarTransportFoam
mv icoFoam.C icoScalarTransportFoam.C
wclean
```

- Go the the Make directory and edit the file `files` as follows:

```
EXE = $(FOAM_USER_APPBIN)/icoScalarTransportFoam
```

- Now compile the application by typing the command `wmake` in the application's folder.

icoScalarTransportFoam

- We want to solve the following transport equation for the scalar field T
- It is an unsteady, convection-diffusion transport equation:

$$\frac{\partial T}{\partial t} + \boldsymbol{\nabla} \cdot T - \boldsymbol{\nabla} \cdot (\nu \, \nabla T) = 0$$

ν is the kinematic viscosity.

WHAT TO DO:

- Create the geometric field `T` in the `createFields.H` file
- Solve the transport equation for `T` in the `icoScalarTransportFoam.C` file

icoScalarTransportFoam

Creating the field `T`

Modify the file `createFields.H` adding this `volScalarField` constructor:

```
Info<< "Reading field T\n" << endl;

volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
   mesh
);
```

icoScalarTransportFoam

Creating the `volScalarField T`

- We have created a `volScalarField` object called `T` by reading a file called `T` in the
  `runTime.timeName()` directory (option: `IOobject::MUST_READ`)

- At the beginning of the simulation, `runTime.timename()` is the `start-Time` value specified in the `controlDict` file.

- `T` will be automatically written (`IOobject::AUTO_WRITE`) in the `run-Time.timeName()` directory according to what is specified in the `control-Dict` file of the case.

- `T` is defined on the computational mesh (`mesh` object)
  - It has as many internal values (`internalField`) as the number of mesh cells
  - It needs as many boundary conditions (`boundaryField`) as the mesh boundaries specified in the `constant/polyMesh/boundary` file of the case.

icoScalarTransportFoam

Solving the transport equation for T.

- Create a new empty file, `TEqn.H`:

```
touch TEqn.H
```

- Include it in icoScalarTransportFoam.C at the end of the PISO loop:

```
....
turbulence->correct();

# include "TEqn.H"
.....
```

- Now we will implement the scalar transport equation for `T` in `icoScalarTransportFoam`...

icoScalarTransportFoam

Solving the transport equation for T.

- This is the transport equation:

$$\frac{\partial T}{\partial t} + \boldsymbol{\nabla} \cdot T - \boldsymbol{\nabla} \cdot (\nu \nabla T) = 0$$

- and here is how we implement it in OpenFOAM:

```
solve
(
     fvm::ddt(T)
   + fvm::div(phi, T)
   - fvm::laplacian(DT, T)
   ==
     fvOptions(T)
);
```

- Now we can compile the application by the command `wmake` in the application's folder

$FOAM_RUN/myFirstTutorial

icoScalarTransportFoam: setting up the case

- Now we want to create a tutorial case to test the functionality of the new solver `icoScalarTransportFoam`

- To do this, we create a new tutorial case based on the `icoFoam` code

- Copy the cavity tutorial case in your $FOAM_RUN directory and rename it

```
cp -r $FOAM_TUTORIALS/icoFoam/cavity $FOAM_RUN
mv cavity cavityScalarTransport
```

- Introduce the field `T` in `cavityScalarTransport/0` directory

```
cp 0/p 0/T
```

icoScalarTransportFoam: **initial conditions**

- Modify T as follows:

```
dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    movingWall
    {
        type            fixedValue;
        value           uniform 1;
    }

    fixedWalls
    {
        type            fixedValue;
        value           $internalField;

    }

    frontAndBack
    {
        type            empty;
    }
}
```

Case setup: system/fvSchemes

- Modify the subdictionary `divSchemes`, introducing the discretization scheme for `div(phi,T)`:

```
divSchemes
{
    default          none;
    div(phi,U)       Gauss linear;
    div(phi,T)       Gauss linear;
}
```

- Modify the subdictionary `laplacianSchemes`, introducing the discretization scheme for `laplacian(nu,T)`:

```
laplacianSchemes
{
    default          none;
    laplacian(nu,U) Gauss linear orthogonal;
    laplacian((1|A(U)),p) Gauss linear orthogonal;
    laplacian(nu,T) Gauss linear orthogonal;
}
```

Case setup: `system/fvSolution`

- Introduce the settings for `T` in the `solvers` subdictionary

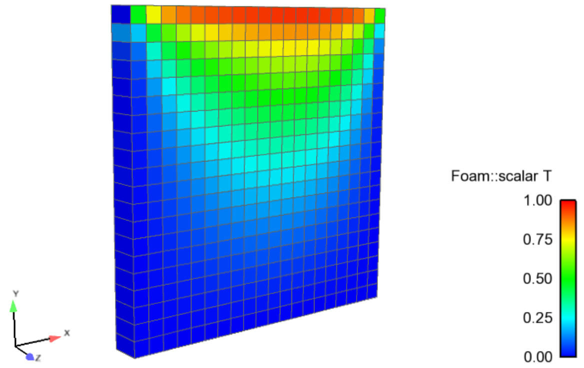
```
"(U|T)"
{
    solver          PBiCG;
    preconditioner  DILU;
    tolerance       1e-05;
    relTol          0;
}
```

NOTES:

- if you forget to insert some of the settings showed above, the solver returns an error.
- regular expressions (see "(U|T)" above!) are allowed in the input files

`icoScalarTransportFoam`: **post-processing**

- Run the case from its folder:
  - `icoScalarTransportFoam`
- post-process the data:



Thank you for your attention!

contact: federico.piscaglia@polimi.it