UNIVERSITY OF
Southampton
School of Engineering Sciences

OpenFOAM Workshop

Programming session:

from the C++ basics to the compilation of user libraries

Daniele Trimarchi
daniele.trimarchi@soton.ac.uk

Southampton, 21 October 2011

Presentation overview:

30 m

- C++ basics
  - main, declarations, types
  - memory management: references and pointers
  - Object orientation:
    - classes
    - inheritance
  - header and source files
- Compiling applications
  - make, wmake, wmake libso. Linking libraries.

15 m

- Writing simple applications in OpenFOAM
  - general structure (includes, skim between time-dirs...)
  - examples: divergence, INDT

Presentation overview:

20 m

- Modifying existing libraries
  - overview of the force class
  - modifying the class
    - Change the source and modify the names
    - compile the library
    - call the library during the execution

25 m

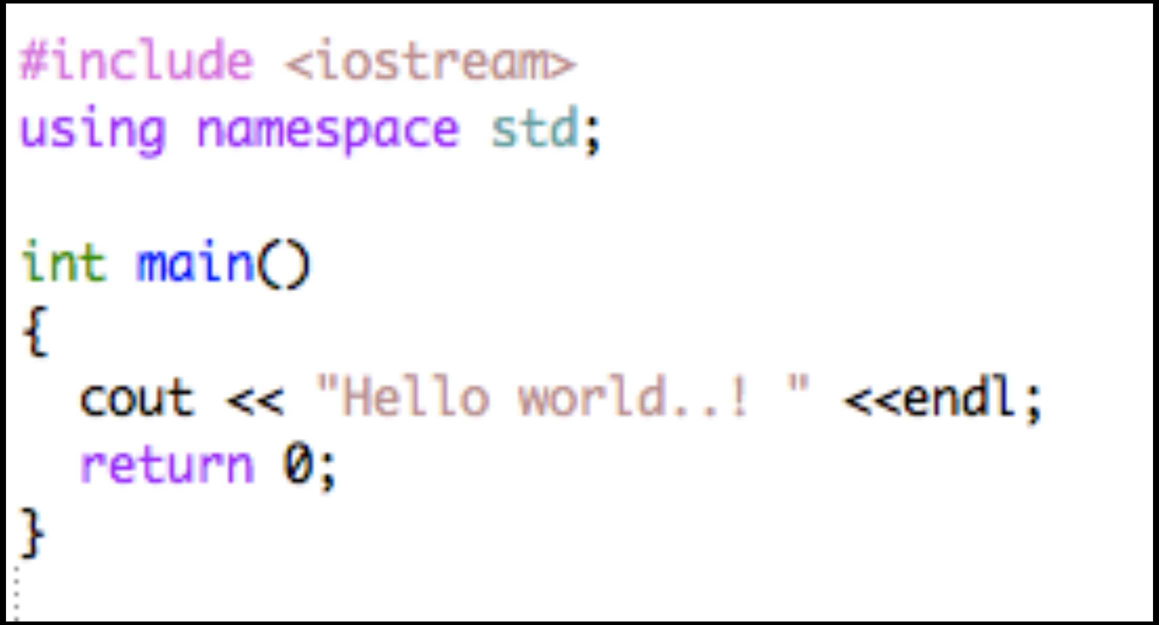
- Adding classes
  - FSInterface class
  - Linking classes trough pointers

PART 1

C++ basics

C++ basics

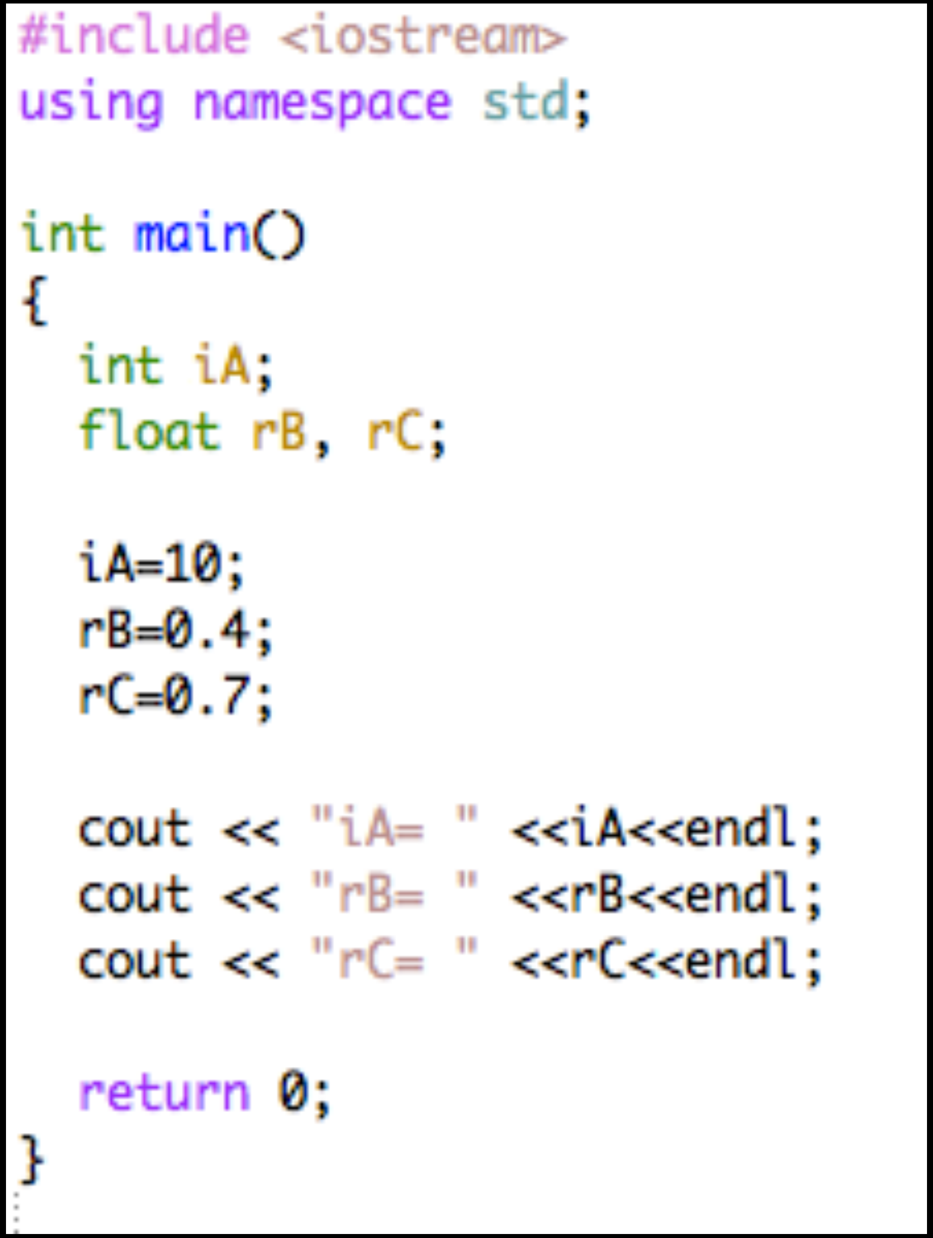
Hello world code... it only prints a message on screen



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

int main()
{
   cout << "Hello world..! " <<endl;
   return 0;
}
```

C++ basics

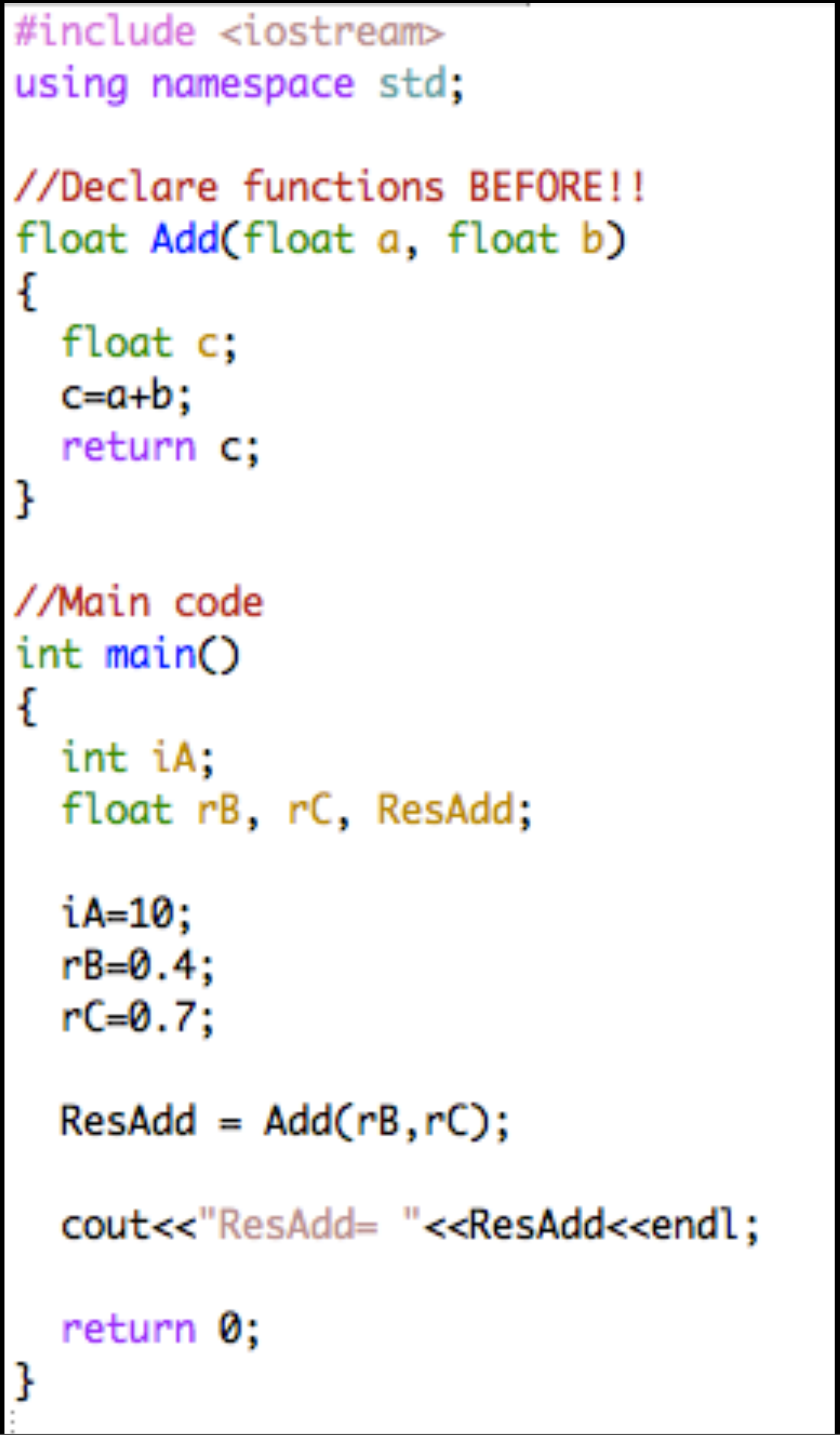
Declare variables, assign values and output the results



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

int main()
{
  int iA;
  float rB, rC;

  iA=10;
  rB=0.4;
  rC=0.7;

  cout << "iA= " <<iA<<endl;
  cout << "rB= " <<rB<<endl;
  cout << "rC= " <<rC<<endl;

  return 0;
}
```

C++ basics

Define and use of the functions

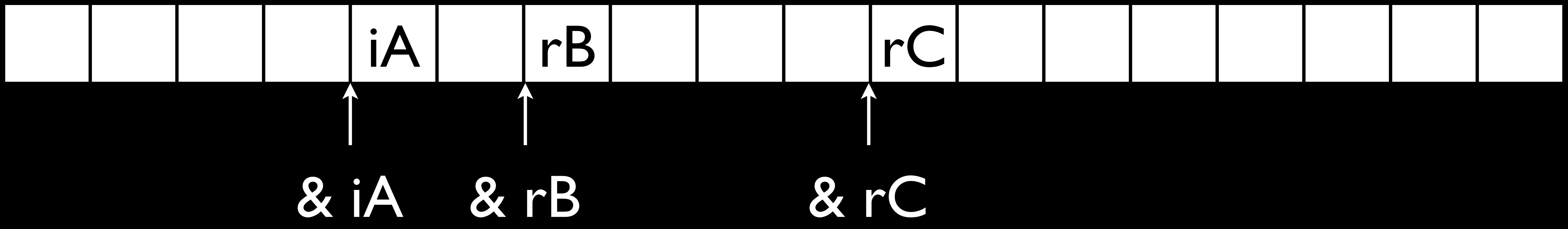


```cpp
#include <iostream>
using namespace std;

//Declare functions BEFORE!!
float Add(float a, float b)
{
  float c;
  c=a+b;
  return c;
}

//Main code
int main()
{
  int iA;
  float rB, rC, ResAdd;

  iA=10;
  rB=0.4;
  rC=0.7;

  ResAdd = Add(rB,rC);

  cout<<"ResAdd= "<<ResAdd<<endl;

  return 0;
}
```

C++ basics

References and pointers

Computer memory: sequential representation



- References (&) are the addresses of the variables in the computer memory. References are constant.
- The value of a reference (ex: 0x7fff5fbfb604) can be stored in a particular type of variable, called pointer.
- The pointer is a variable of the same type of the variable it points to. Integer pointer points to integer variable!

C++ basics

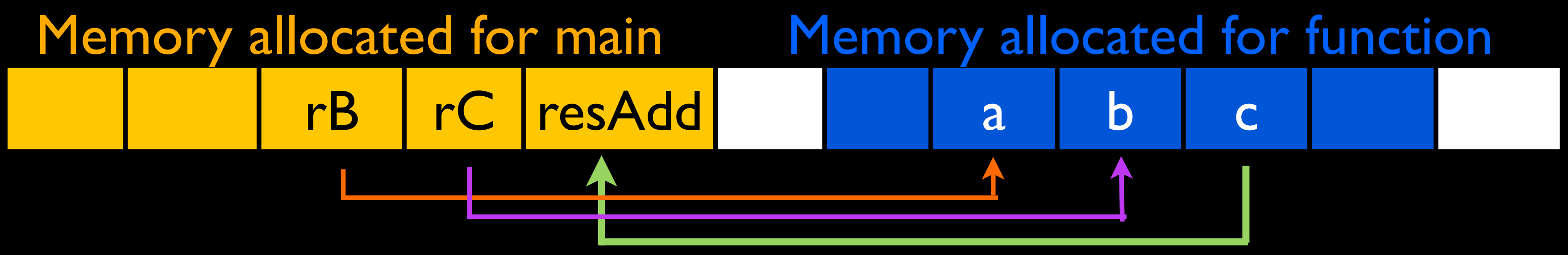
Declaration   Assignation

int * APtr   = & iA
float * rPtr = & rB

...

rPtr = & rC Re-Assignation

- Using references and pointers is convenient: it avoids the need of making copies when passing arguments to functions (for large arrays this is very convenient!)



main

```
float rB = 1.1;
float rC = 0.6;
ResAdd = Add(rB, rC);

cout << ResAdd << endl;
```

copy: a = rB
copy: b = rC

function

```
float c = a + b;
```

copy: ResAdd = c

C++ basics

- The deferencing operator * restitutes the value of the variable pointed by the pointer. So:

Variable declaration and assignation:

Pointer declaration and assignation:

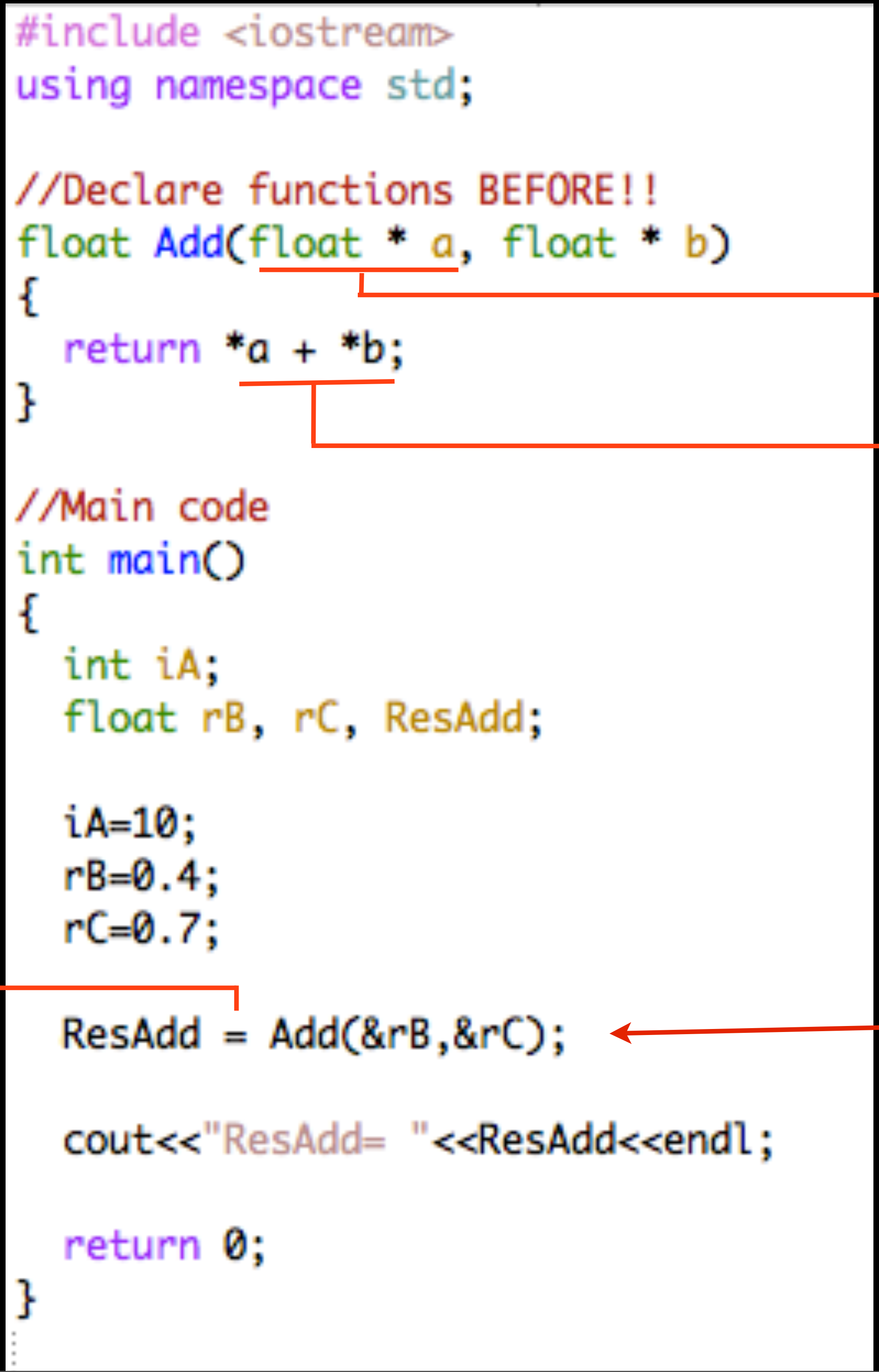
Pointer deferencing:

```
float rB =  1.1;
float * rPtr =  & rB

cout<<   rB <<endl;        ⟶   1.1
cout<< * rPtr <<endl;      ⟶   1.1

cout<< & rB<<endl;         ⟶   0x7fff5fbfb604
```

Similar syntax, but very different meaning!!

C++ basics

Optimizing the function



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

//Declare functions BEFORE!!
float Add(float * a, float * b)
{
   return *a + *b;
}

//Main code
int main()
{
   int iA;
   float rB, rC, ResAdd;

   iA=10;
   rB=0.4;
   rC=0.7;

   ResAdd = Add(&rB,&rC);

   cout<<"ResAdd= "<<ResAdd<<endl;

   return 0;
}
```

1 Pass references to the function (no copies!)

2 Assign the pointer: float * a = & rB

3 Deference and return therefore a float value

4 Assign the returned value to the float variable

C++ basics

- CRectangle class

Class declaration

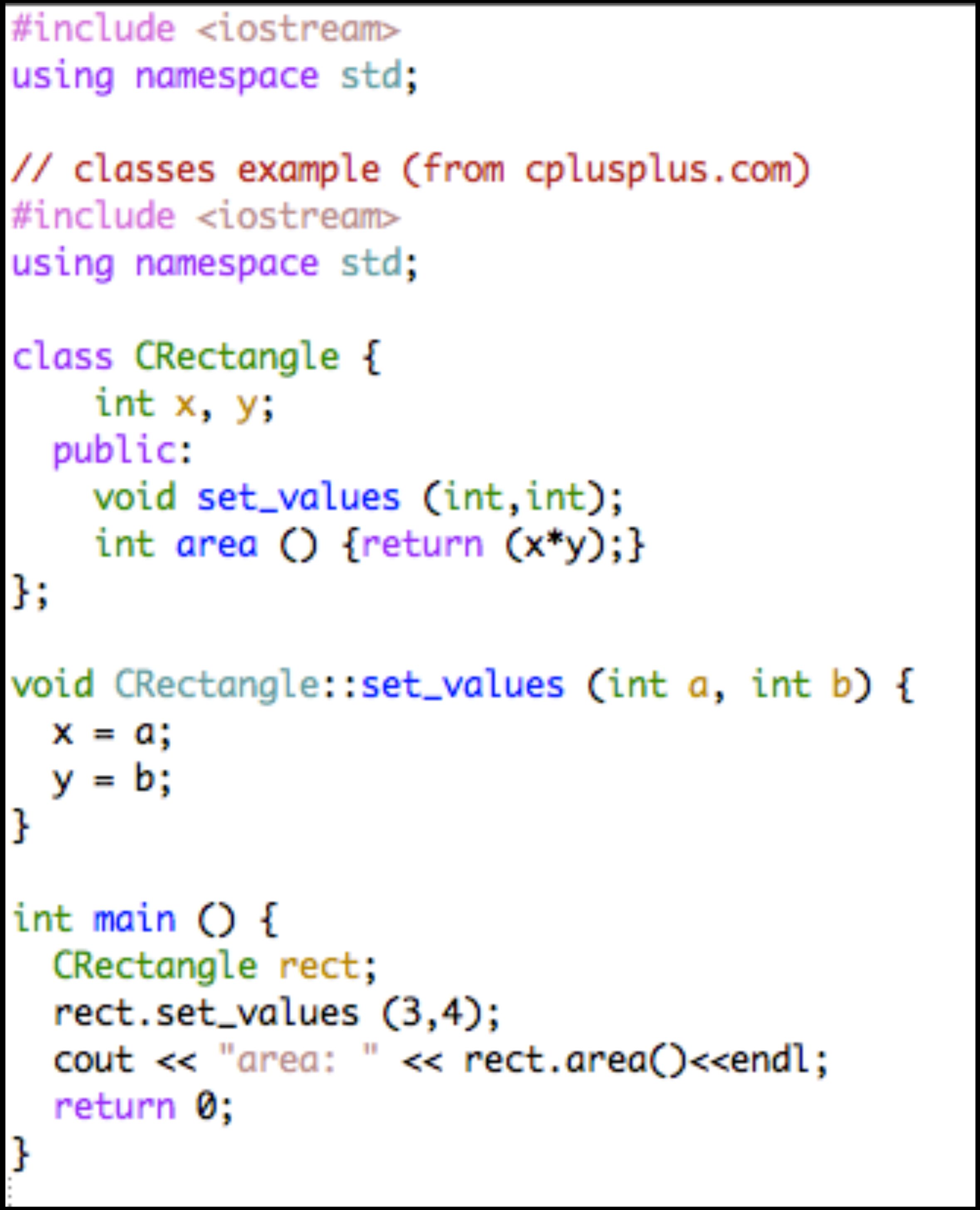
Private members

Public members and functions

Public function definition

Object declaration

Call the function member of the class



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

// classes example (from cplusplus.com)
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area()<<endl;
  return 0;
}
```

Declarations

Definition

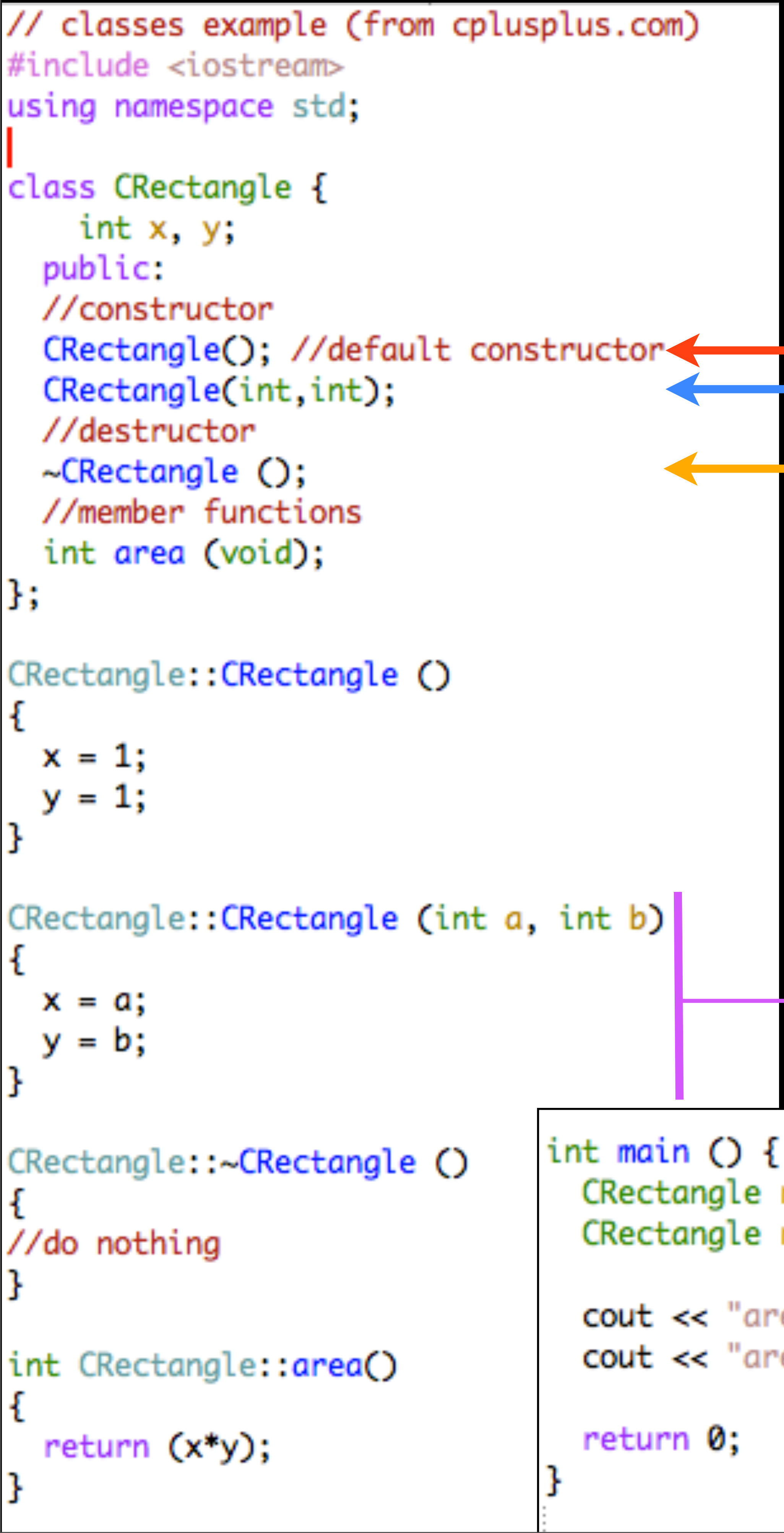
C++ basics

The function is called with the operator “ . ” : rect.setValues(3,4)
The function can also be called using a pointer:

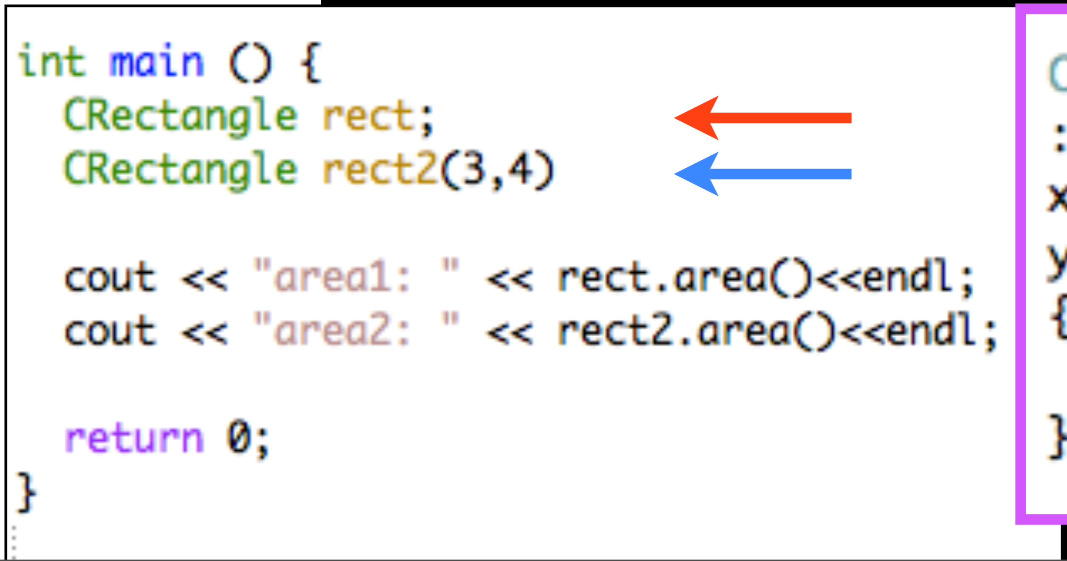
```
CRectangle tria                          →  declare the variable
CRectangle * triaPtr                     →  declare the pointer

triaPtr = & tria;                        →  assign the pointer

cout << tria.area() << endl;             →  call fcn using object
cout << * tria.area() << endl;           →  call fcn using object, by
                                            de-referencing pointer
cout << triaPtr -> area() << endl        →  call fcn using pointer
```

C++ basics

- The constructor is a class-member function called when the object is initially build
- Dual of the Constructor is the Destructor
- As for every function, different arguments define different constructors; the constructor can also call other functions

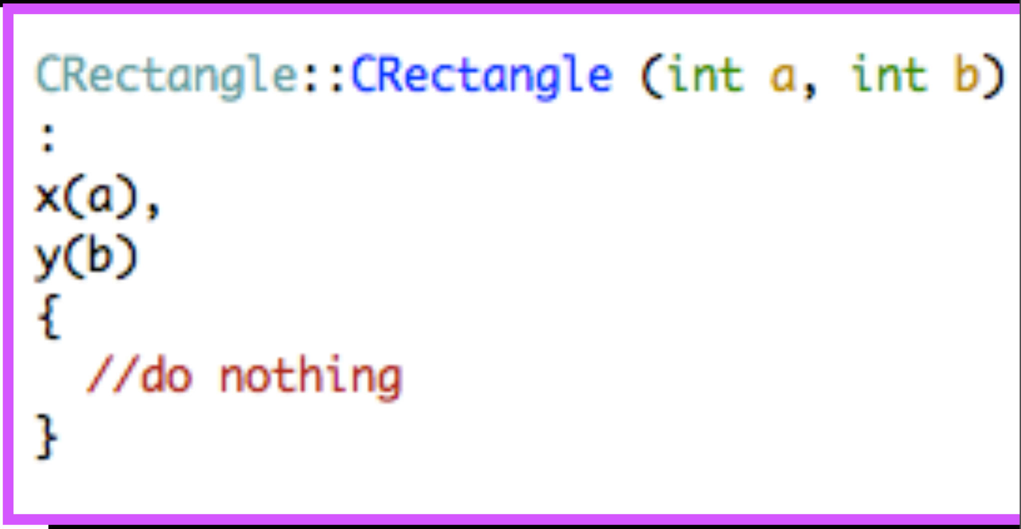


```
// classes example (from cplusplus.com)
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
  //constructor
  CRectangle(); //default constructor
  CRectangle(int,int);
  //destructor
  ~CRectangle ();
  //member functions
  int area (void);
};

CRectangle::CRectangle ()
{
  x = 1;
  y = 1;
}

CRectangle::CRectangle (int a, int b)
{
  x = a;
  y = b;
}

CRectangle::~CRectangle ()
{
//do nothing
}

int CRectangle::area()
{
  return (x*y);
}
```



```
int main () {
  CRectangle rect;
  CRectangle rect2(3,4)

  cout << "area1: " << rect.area()<<endl;
  cout << "area2: " << rect2.area()<<endl;

  return 0;
}
```

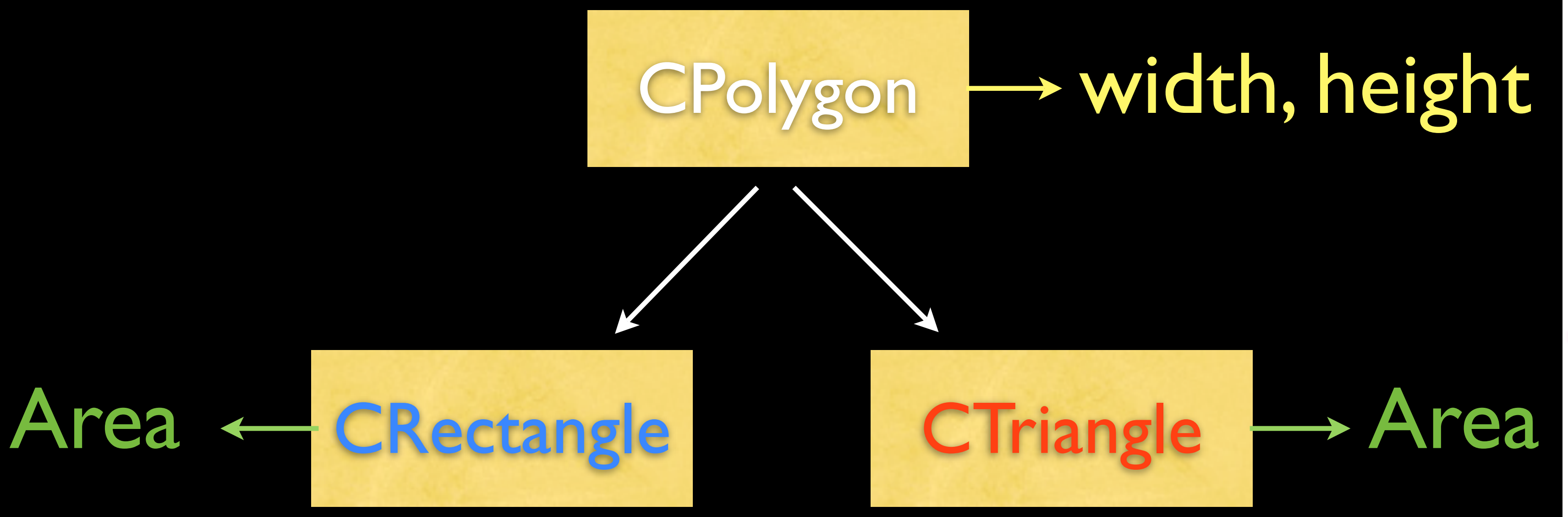
Standard syntax for the constructor



```
CRectangle::CRectangle (int a, int b)
:
x(a),
y(b)
{
  //do nothing
}
```

C++ basics

- Inheritance: Cpolygon class

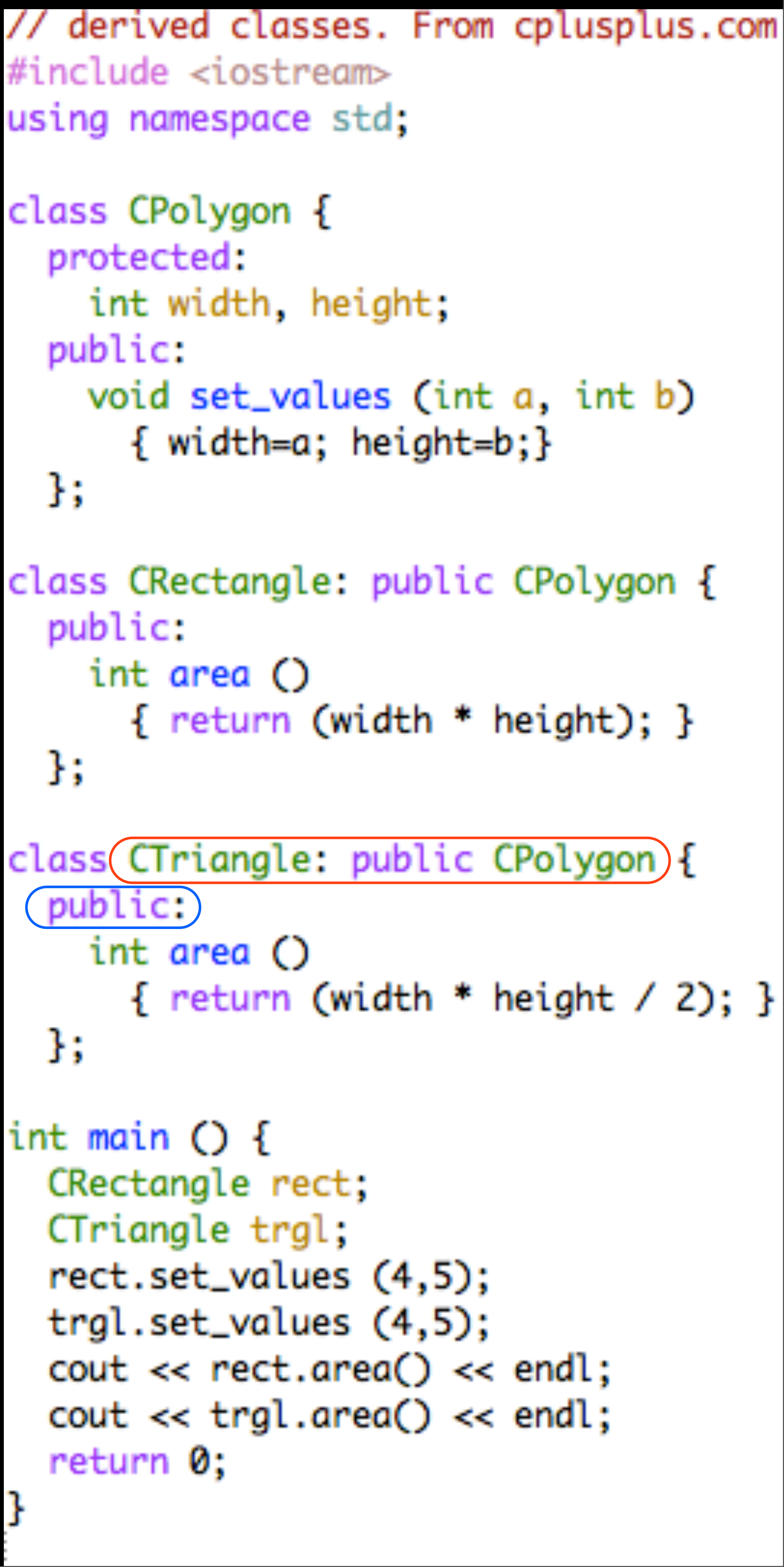


Inherited class:
class CRectangle : public CPolygon

... is a ...

!!

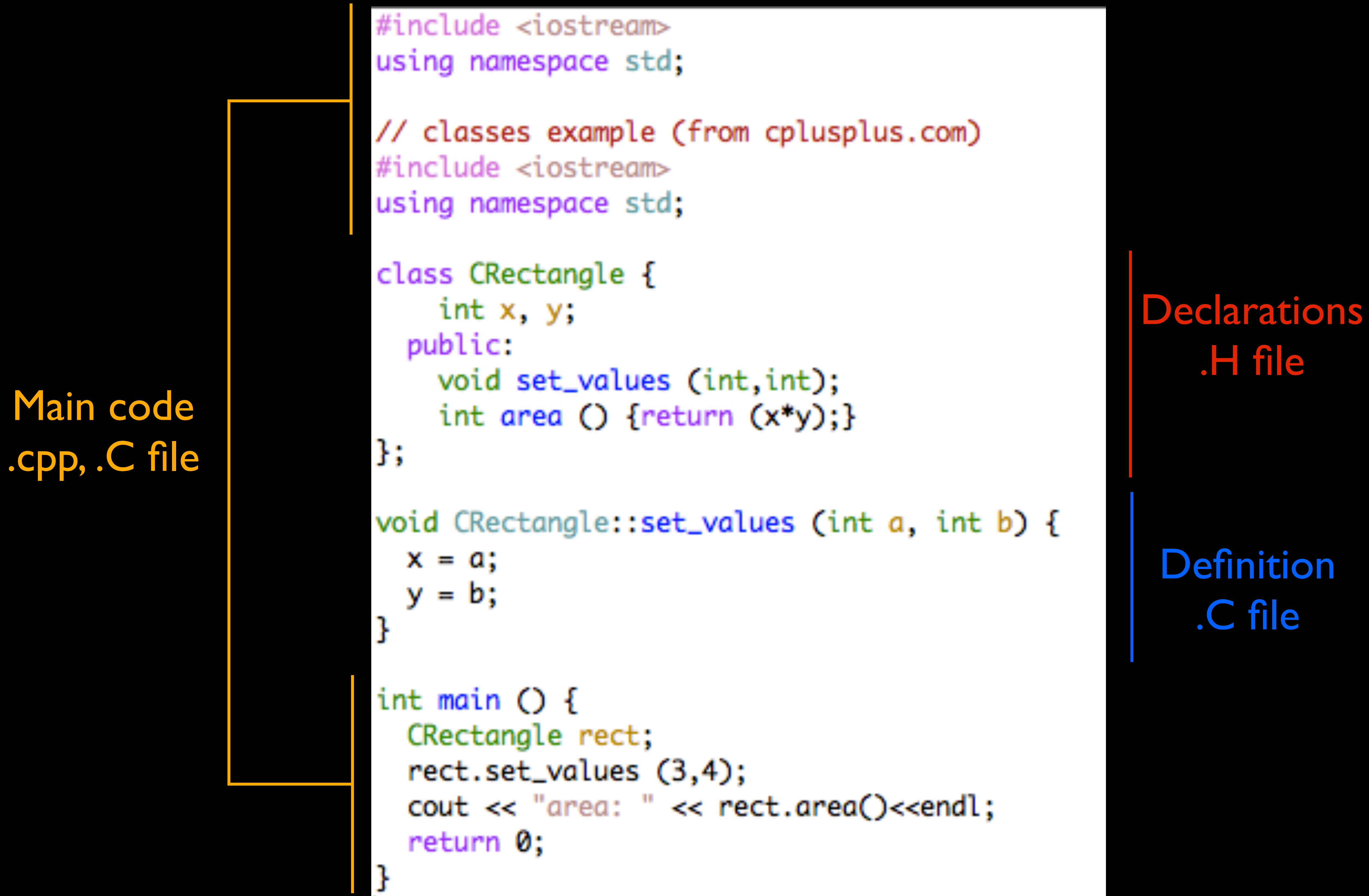
Every object has its own function definition. The function behaves differently accordingly to the type of the object



```
// derived classes. From cplusplus.com
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

C++ basics

- Main (.cpp; .C), Header (.H) and Source (.C) files

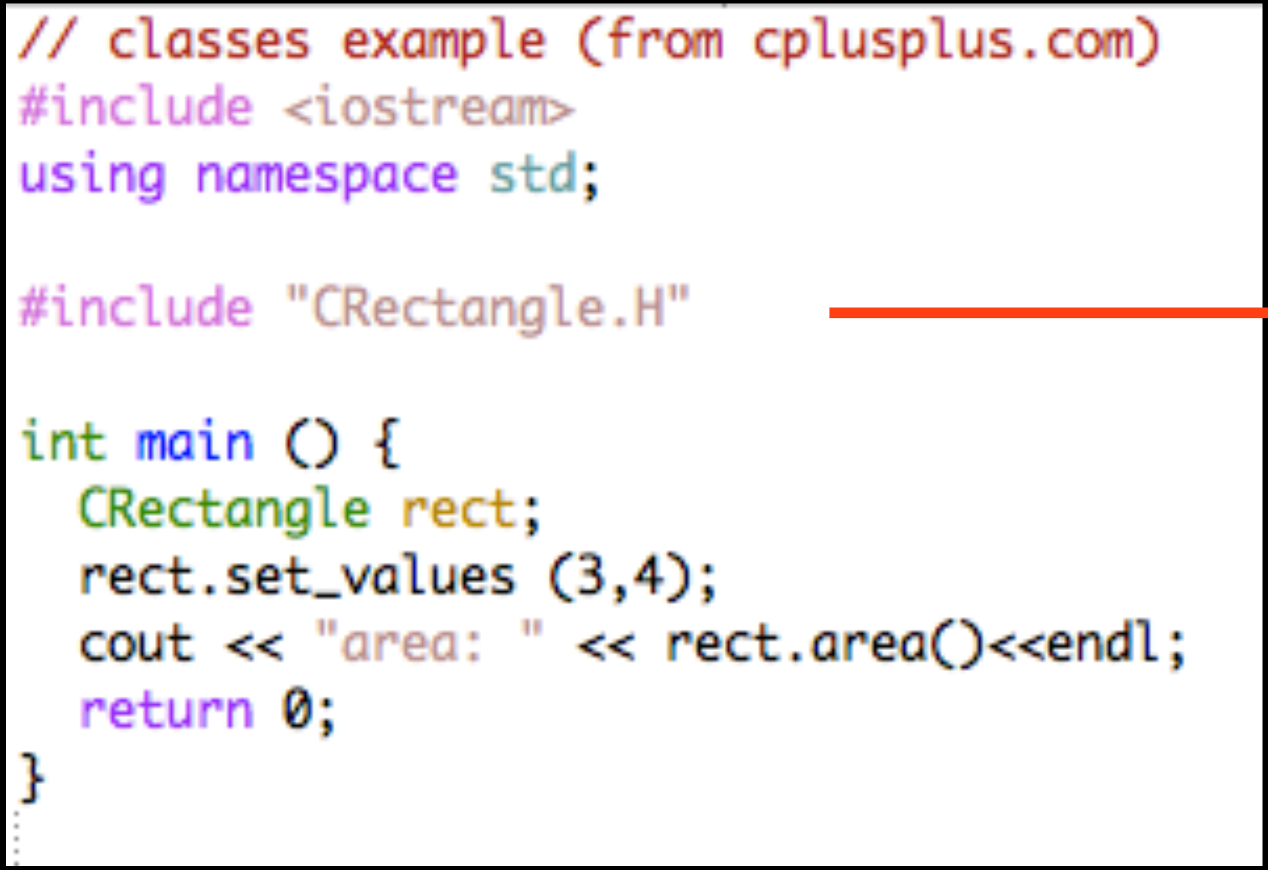


```
#include <iostream>
using namespace std;

// classes example (from cplusplus.com)
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area()<<endl;
  return 0;
}
```

C++ basics

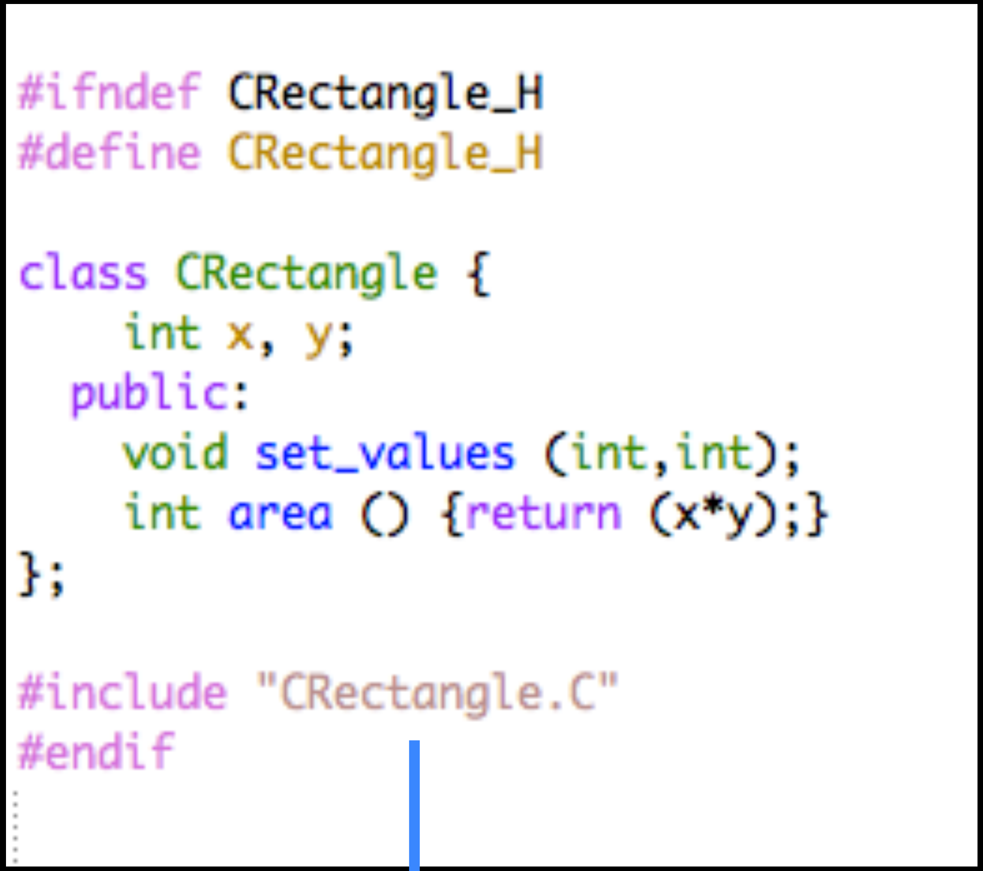
- Main (.cpp; .C), Header (.H) and Source (.C) files

Main code
.cpp, .C file



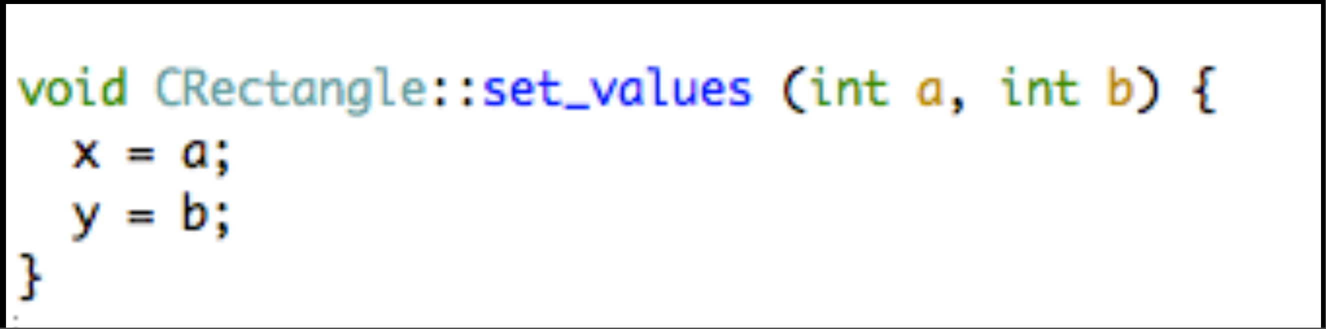
```
// classes example (from cplusplus.com)
#include <iostream>
using namespace std;

#include "CRectangle.H"

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area()<<endl;
  return 0;
}
```

Declarations
.H file



```
#ifndef CRectangle_H
#define CRectangle_H

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

#include "CRectangle.C"
#endif
```

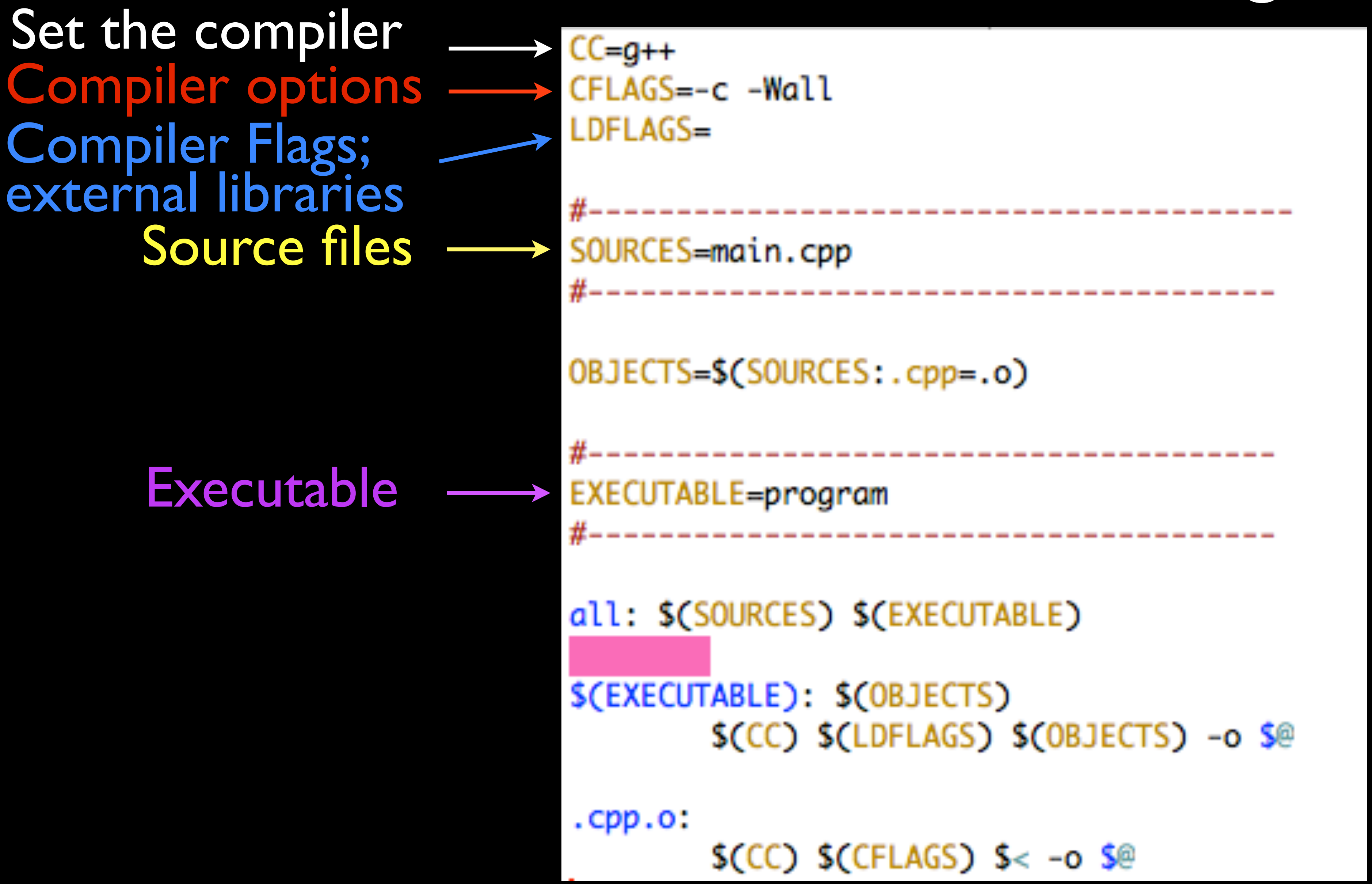
Definition
.C file



```
void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}
```

C++ basics

- Compiling applications: passing from human readable instructions to binaries
- Unix environment: call makefile trough “make”



```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=

#---------------------------------------
SOURCES=main.cpp
#---------------------------------------

OBJECTS=$(SOURCES:.cpp=.o)

#---------------------------------------
EXECUTABLE=program
#---------------------------------------

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
	$(CC) $(CFLAGS) $< -o $@
```

- The “make” command is overwritten in OpenFOAM by “wmake” and “wmake libso”

References

Several books are available on C++. For example:

Deitel, Deitel
C++, How to program
ed. Prentice Hall

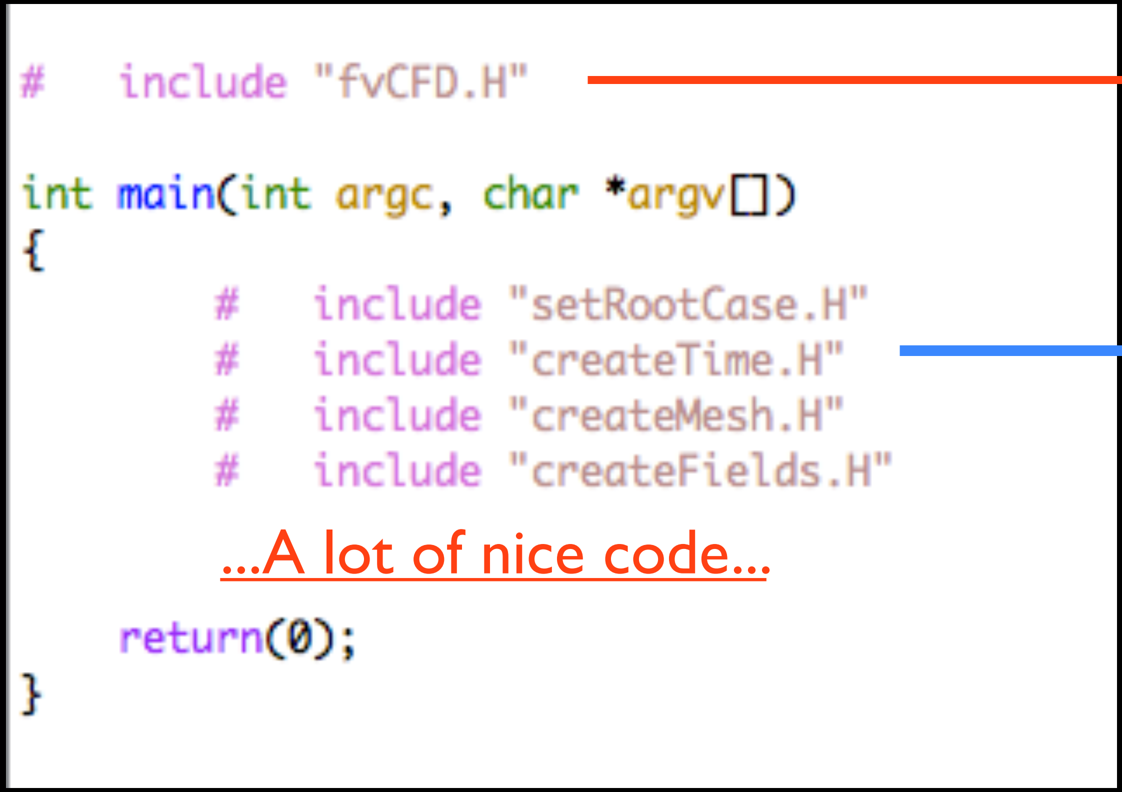
A book is generally better (expecially for C++)...
but on-line useful guidance can be also found:

http://www.cplusplus.com/doc/tutorial/

PART 2

Writing simple applications in OpenFOAM

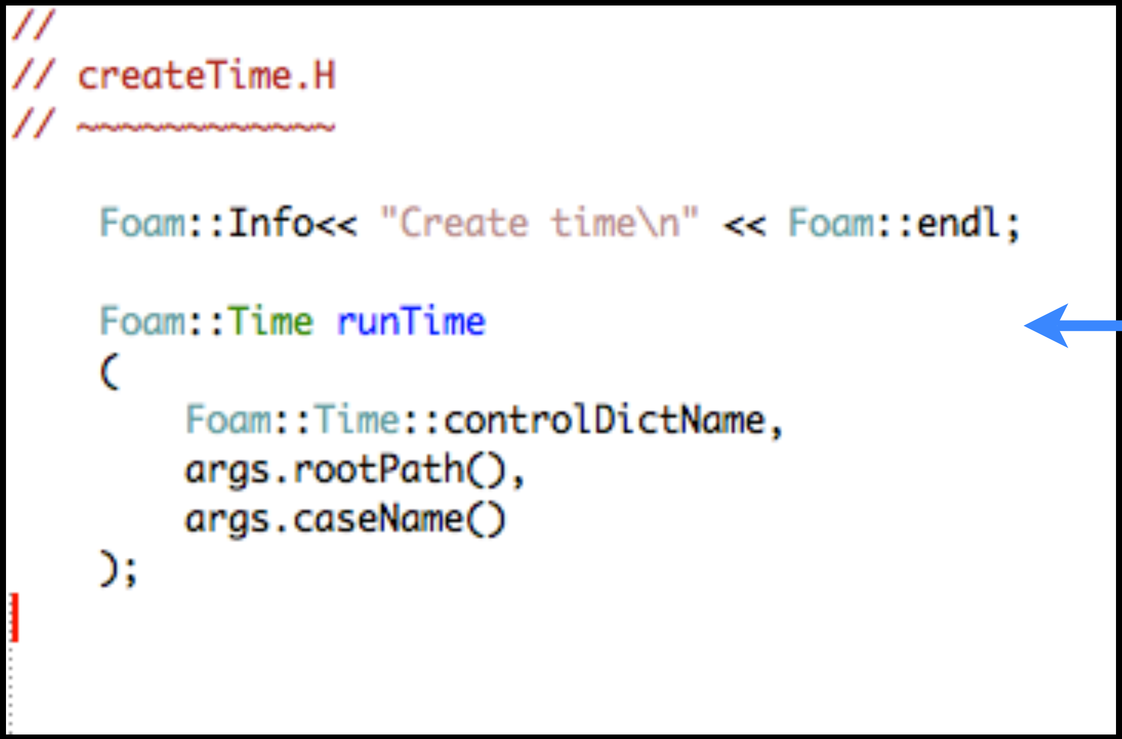
General structure of an OpenFOAM application



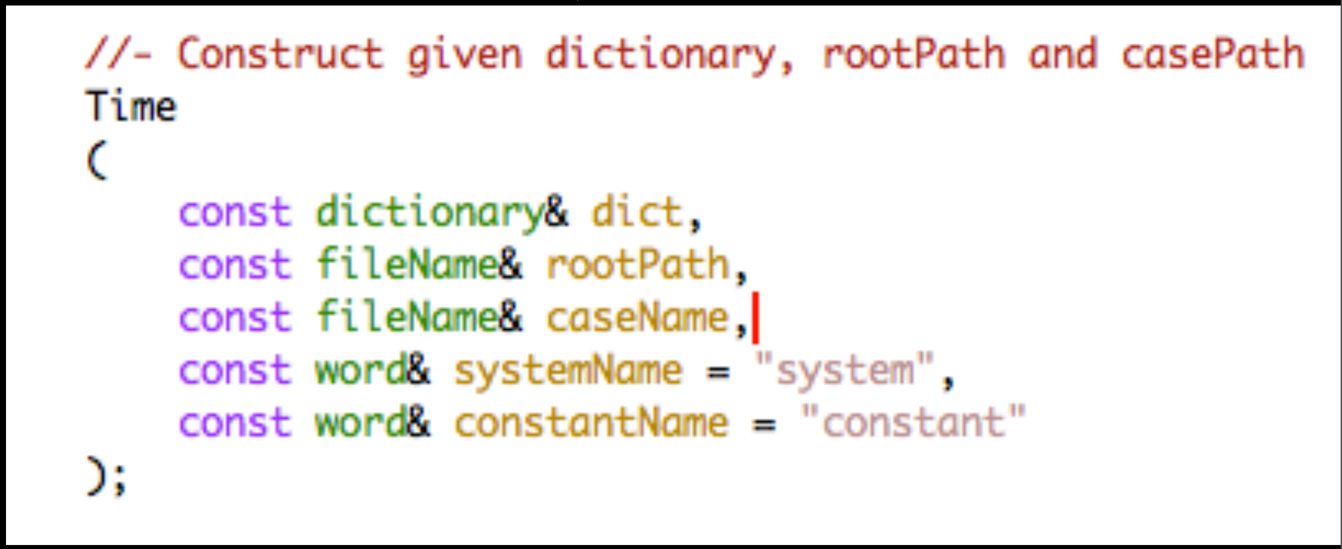
```
#   include "fvCFD.H"

int main(int argc, char *argv[])
{
        #   include "setRootCase.H"
        #   include "createTime.H"
        #   include "createMesh.H"
        #   include "createFields.H"

          ...A lot of nice code...

    return(0);
}
```

src/finiteVolume/cfdTools/general/include/fvCFD.H
several other include: classes for time, mesh, geometry, math constants...

src/OpenFOAM/include/createTime.H
Declares runTime, object of the class Foam::Time.
Constructor defined in
src/OpenFOAM/db/Time/Time.H, line193

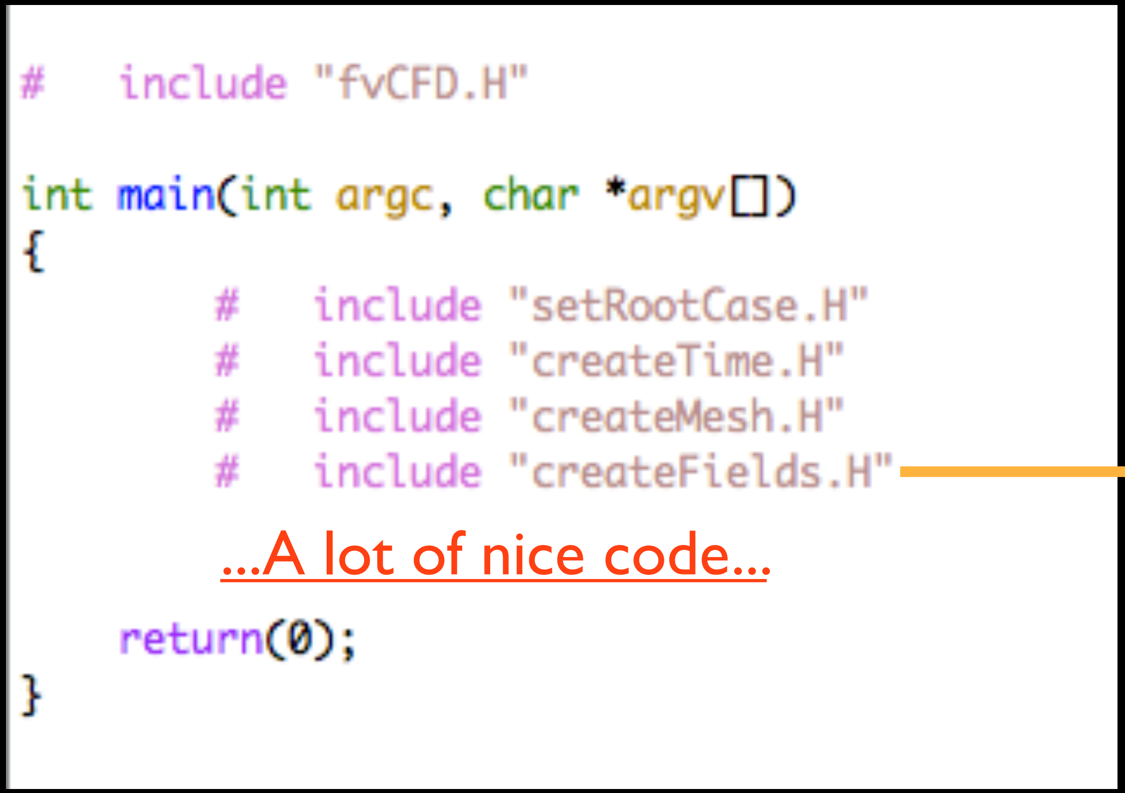


```
//
// createTime.H
// ~~~~~~~~~~~~

    Foam::Info<< "Create time\n" << Foam::endl;

    Foam::Time runTime
    (
        Foam::Time::controlDictName,
        args.rootPath(),
        args.caseName()
    );
```

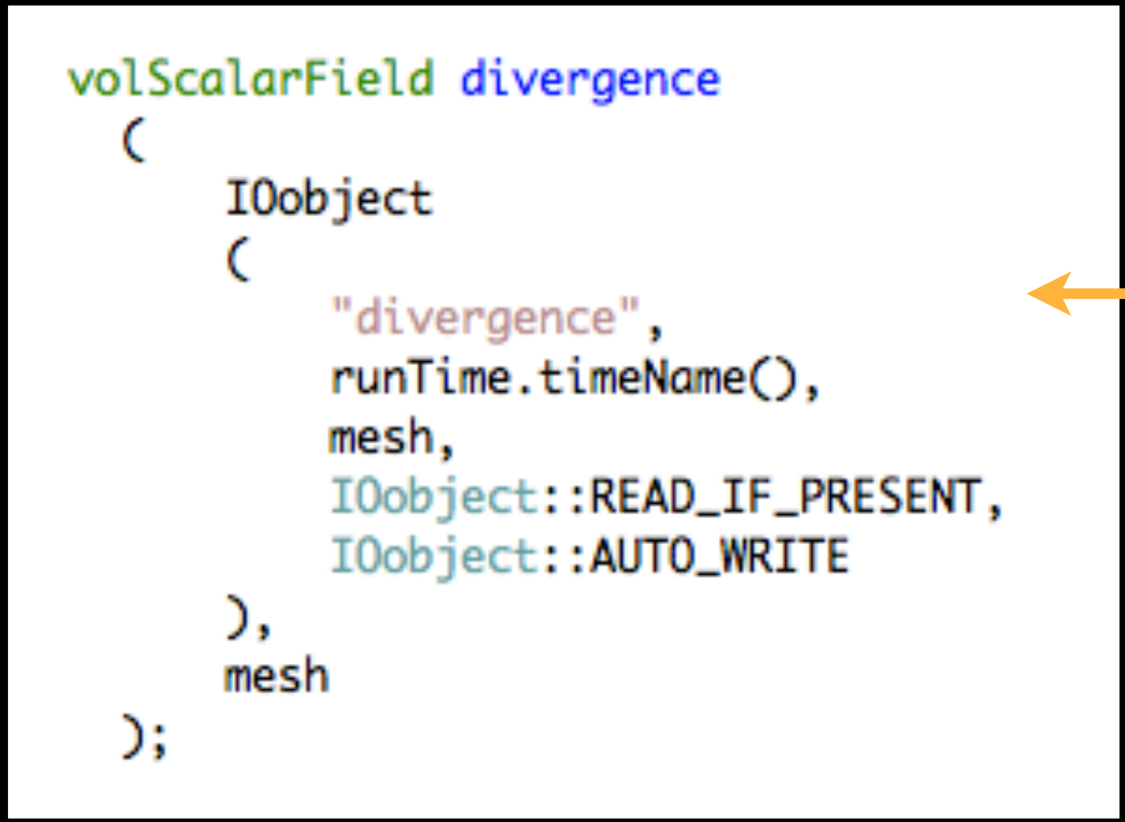


```
    //- Construct given dictionary, rootPath and casePath
    Time
    (
        const dictionary& dict,
        const fileName& rootPath,
        const fileName& caseName,
        const word& systemName = "system",
        const word& constantName = "constant"
    );
```

General structure of an OpenFOAM application



```
#   include "fvCFD.H"

int main(int argc, char *argv[])
{
        #   include "setRootCase.H"
        #   include "createTime.H"
        #   include "createMesh.H"
        #   include "createFields.H"

        ...A lot of nice code...

    return(0);
}
```



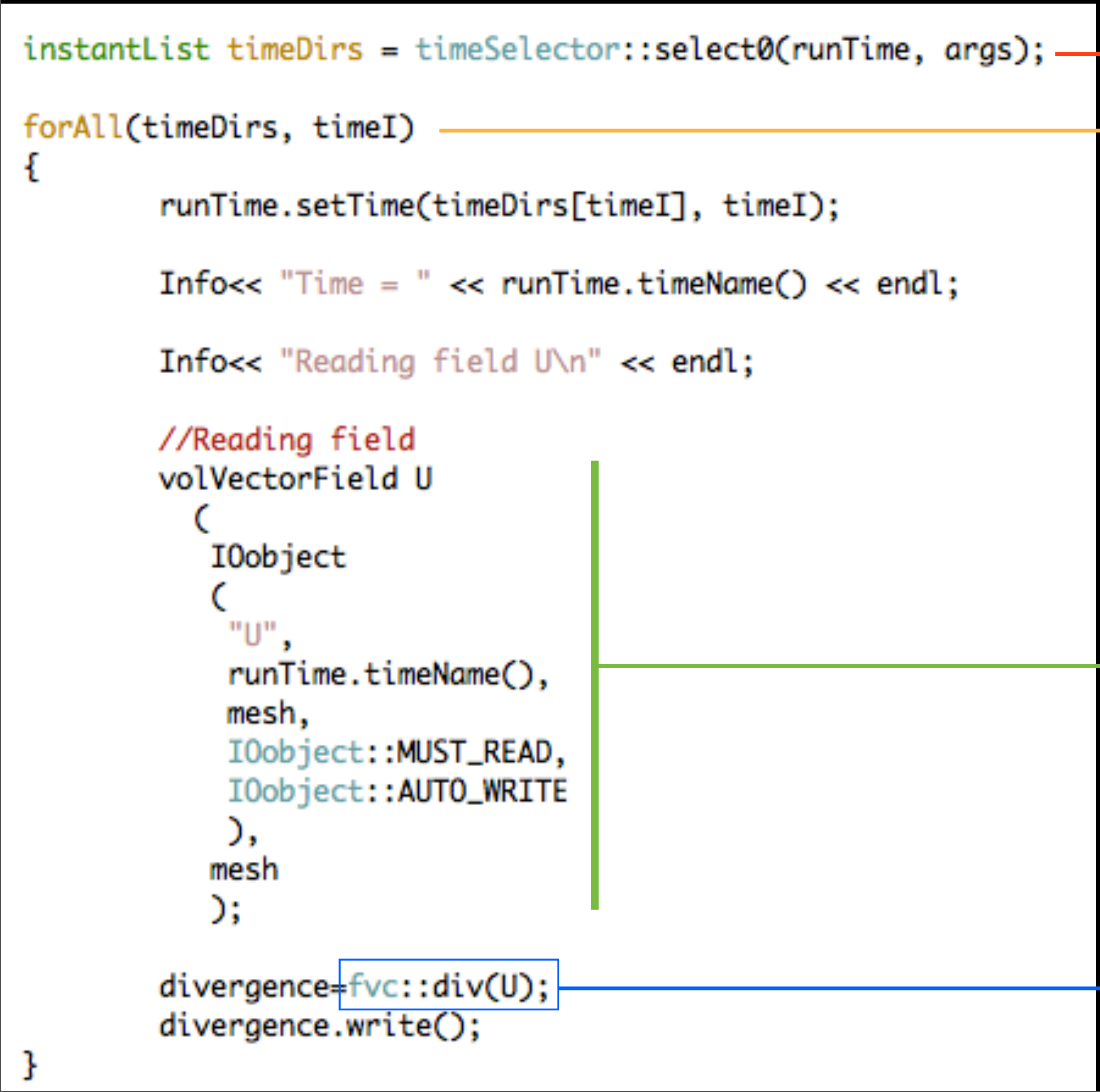
```
  volScalarField divergence
    (
        IOobject
        (
            "divergence",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh
    );
```

File in the source directory

Declares a VolScalarField called *divergence* to be written in every time-step folder

Calculating the divergence $\nabla\cdot$

$$\nabla \cdot \vec{v} = \frac{\partial v_1}{x_1} + \frac{\partial v_2}{x_2} + \frac{\partial v_3}{x_3} = \delta_{ik} v_{i,k}$$



```
instantList timeDirs = timeSelector::select0(runTime, args);

forAll(timeDirs, timeI)
{
        runTime.setTime(timeDirs[timeI], timeI);

        Info<< "Time = " << runTime.timeName() << endl;

        Info<< "Reading field U\n" << endl;

        //Reading field
        volVectorField U
          (
           IOobject
           (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ,
            IOobject::AUTO_WRITE
            ),
           mesh
           );

        divergence=fvc::div(U);
        divergence.write();
}
```

Check for existing time dirs

OpenFOAM version of the for loop. Equivalent to:

```
for(int timeI=0; i<timeDirs.size(); timeI++)
{...}
```

Declares and reads the field U from the selected time directory

calculates the field

Compiling the application

Working directory:

```
SolutionDivergence.C
createFields.H
Make
    files
    options
```

```
solutionDivergence.C
EXE = $(FOAM_USER_APPBIN)/DivU
```

→ Source file

→ Compile application

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude

EXE_LIBS = -lfiniteVolume
```

→ include headers for FV

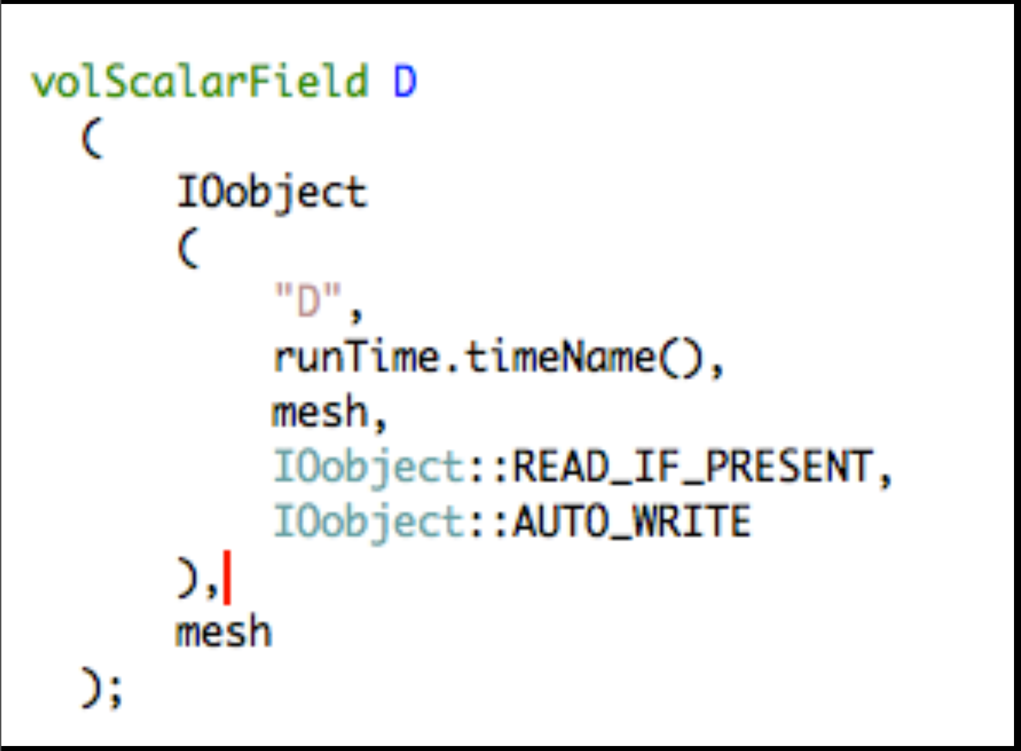
→ include FV library

The application is compiled typing at terminal the command: *wmake*

Calculating the Normalised invariant of the deformation tensor

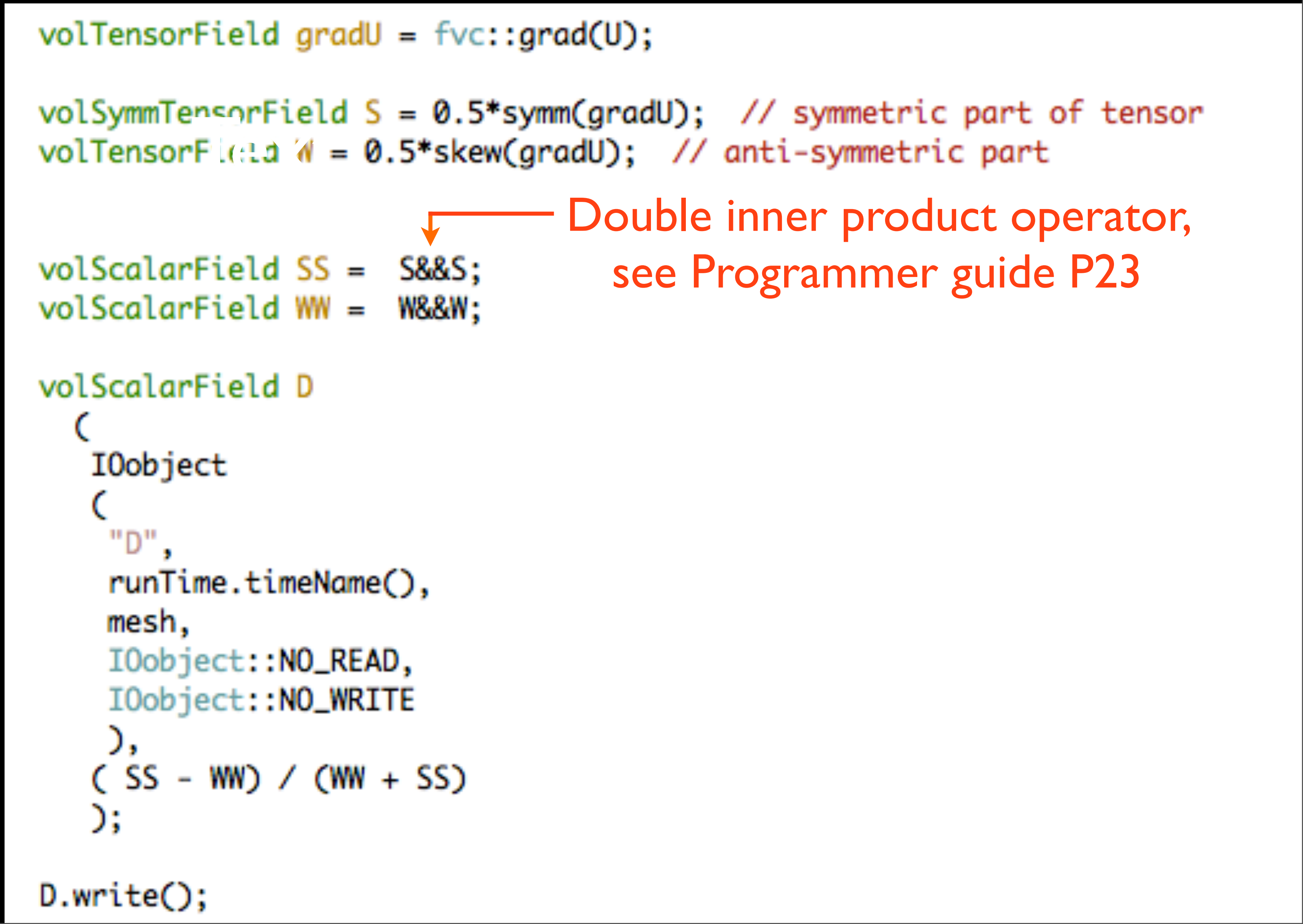
$$D = \frac{S_{ij}S_{ij} - W_{ij}W_{ij}}{S_{ij}S_{ij} + W_{ij}W_{ij}}; S_{ij} = \frac{1}{2}\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}; W_{ij} = \frac{1}{2}\frac{\partial u_i}{\partial x_j} - \frac{\partial u_j}{\partial x_i}$$

CreateFields.h



```
volScalarField D
  (
      IOobject
      (
          "D",
          runTime.timeName(),
          mesh,
          IOobject::READ_IF_PRESENT,
          IOobject::AUTO_WRITE
      ),
      mesh
  );
```

Core of the main code



```
volTensorField gradU = fvc::grad(U);

volSymmTensorField S = 0.5*symm(gradU);   // symmetric part of tensor
volTensorField W = 0.5*skew(gradU);   // anti-symmetric part


volScalarField SS =  S&&S;
volScalarField WW =  W&&W;

volScalarField D
  (
   IOobject
   (
    "D",
    runTime.timeName(),
    mesh,
    IOobject::NO_READ,
    IOobject::NO_WRITE
    ),
   ( SS - WW) / (WW + SS)
   );

D.write();
```

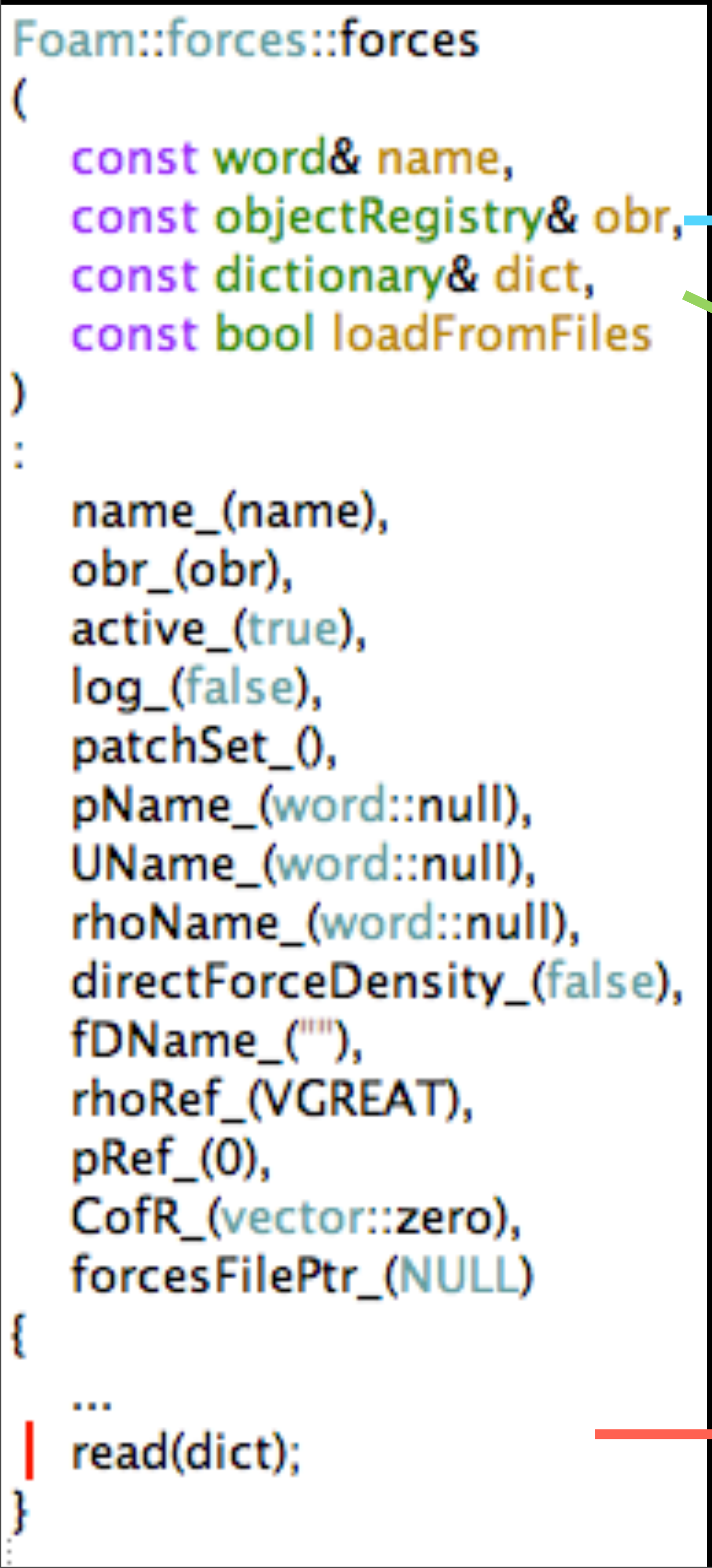
Double inner product operator, see Programmer guide P23

PART 3

Modifying existing libraries

Constructor of the class forces

Path:
OpenFOAM/src/postProcessing/functionObjects/forces/forces/Force.C



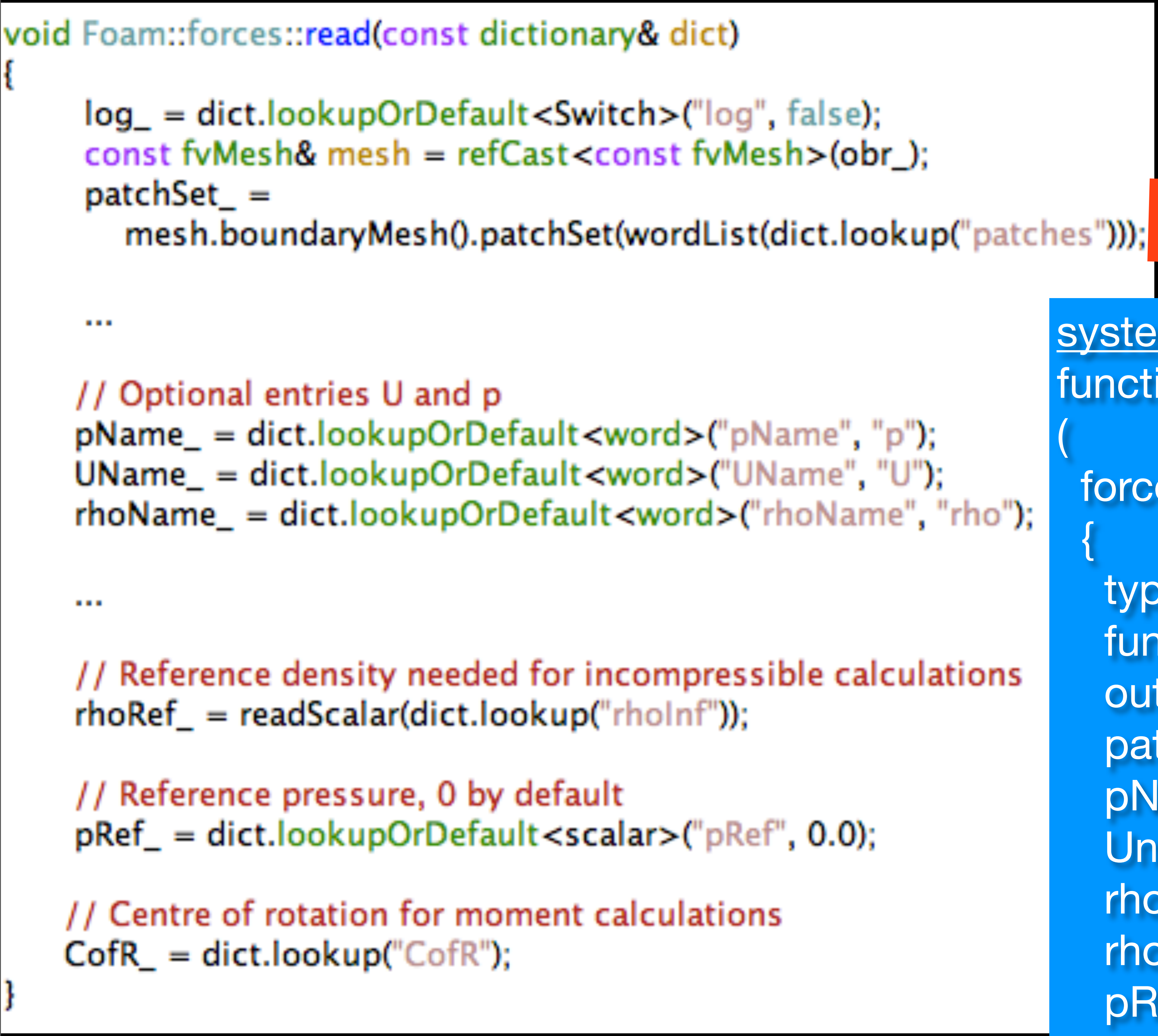
```
Foam::forces::forces
(
    const word& name,
    const objectRegistry& obr,
    const dictionary& dict,
    const bool loadFromFiles
)
:
    name_(name),
    obr_(obr),
    active_(true),
    log_(false),
    patchSet_(),
    pName_(word::null),
    UName_(word::null),
    rhoName_(word::null),
    directForceDensity_(false),
    fDName_(""),
    rhoRef_(VGREAT),
    pRef_(0),
    CofR_(vector::zero),
    forcesFilePtr_(NULL)
{
    ...
    read(dict);
}
```

Reference to the Object Registry. This is a list of the entities pertaining to an object

Reference to the controlDict

Call the member function forces::read
Read the entries in the controlDict

Reading entries from the controlDict



```
void Foam::forces::read(const dictionary& dict)
{
        log_ = dict.lookupOrDefault<Switch>("log", false);
        const fvMesh& mesh = refCast<const fvMesh>(obr_);
        patchSet_ =
            mesh.boundaryMesh().patchSet(wordList(dict.lookup("patches")));

        ...

       // Optional entries U and p
       pName_ = dict.lookupOrDefault<word>("pName", "p");
       UName_ = dict.lookupOrDefault<word>("UName", "U");
       rhoName_ = dict.lookupOrDefault<word>("rhoName", "rho");

       ...

       // Reference density needed for incompressible calculations
       rhoRef_ = readScalar(dict.lookup("rhoInf"));

       // Reference pressure, 0 by default
       pRef_ = dict.lookupOrDefault<scalar>("pRef", 0.0);

      // Centre of rotation for moment calculations
      CofR_ = dict.lookup("CofR");
}
```

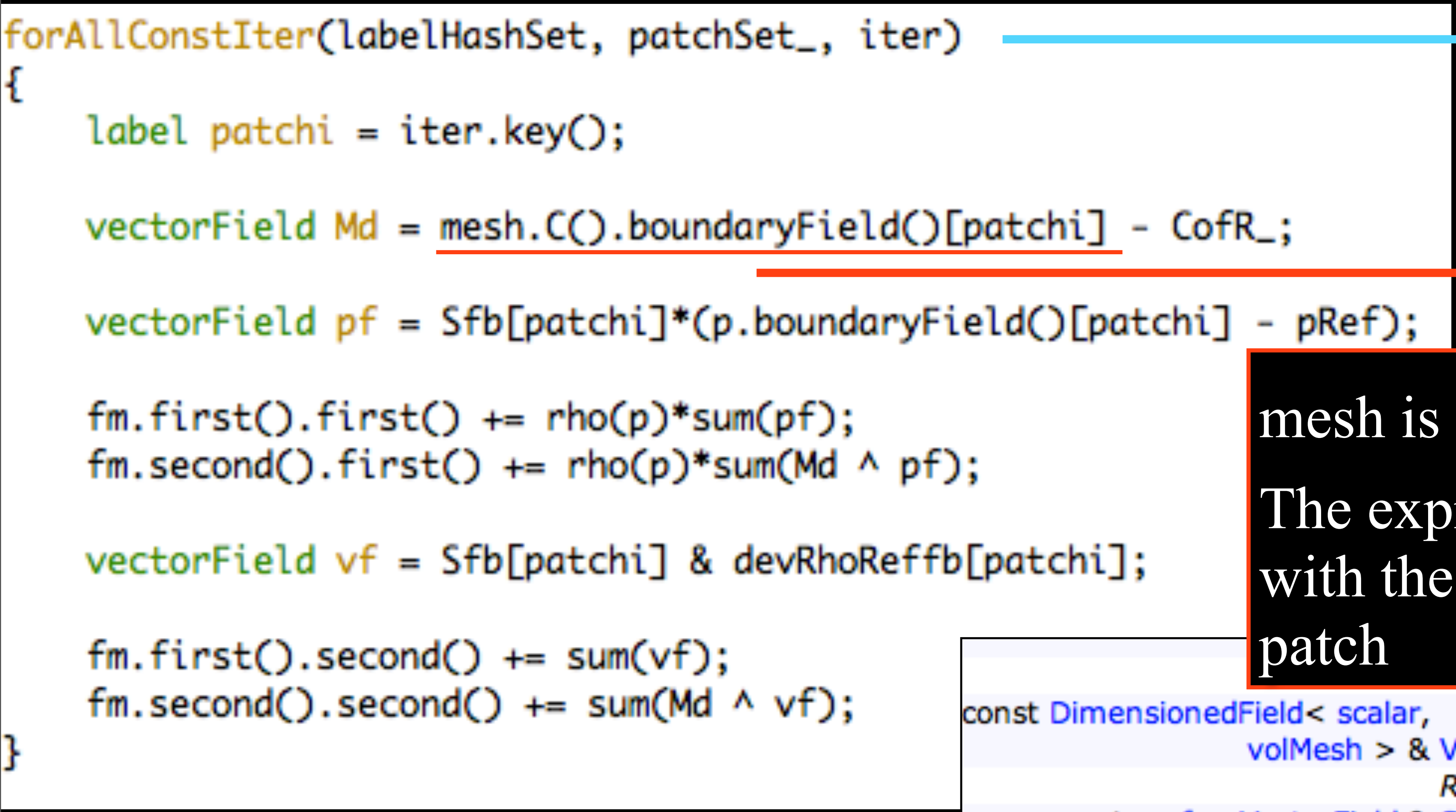
patches on which forces will be integrated

system/controlDict:

```
functions
(
  forces
  {
    type forces;
    functionObjectLibs ("libforces.dylib");
    outputControl outputTime;
    patches (wing);
    pName p;
    Uname U;
    rhoName rhoInf;
    rhoInf 1.2; //Reference density
    pRef 0;
    CofR (0 0 0); //Origin for moments
  }
)
```

Calculating the forces

The virtual function write() is called during the execution. This calls forces::calcForcesMoment(), where the calculation is performed

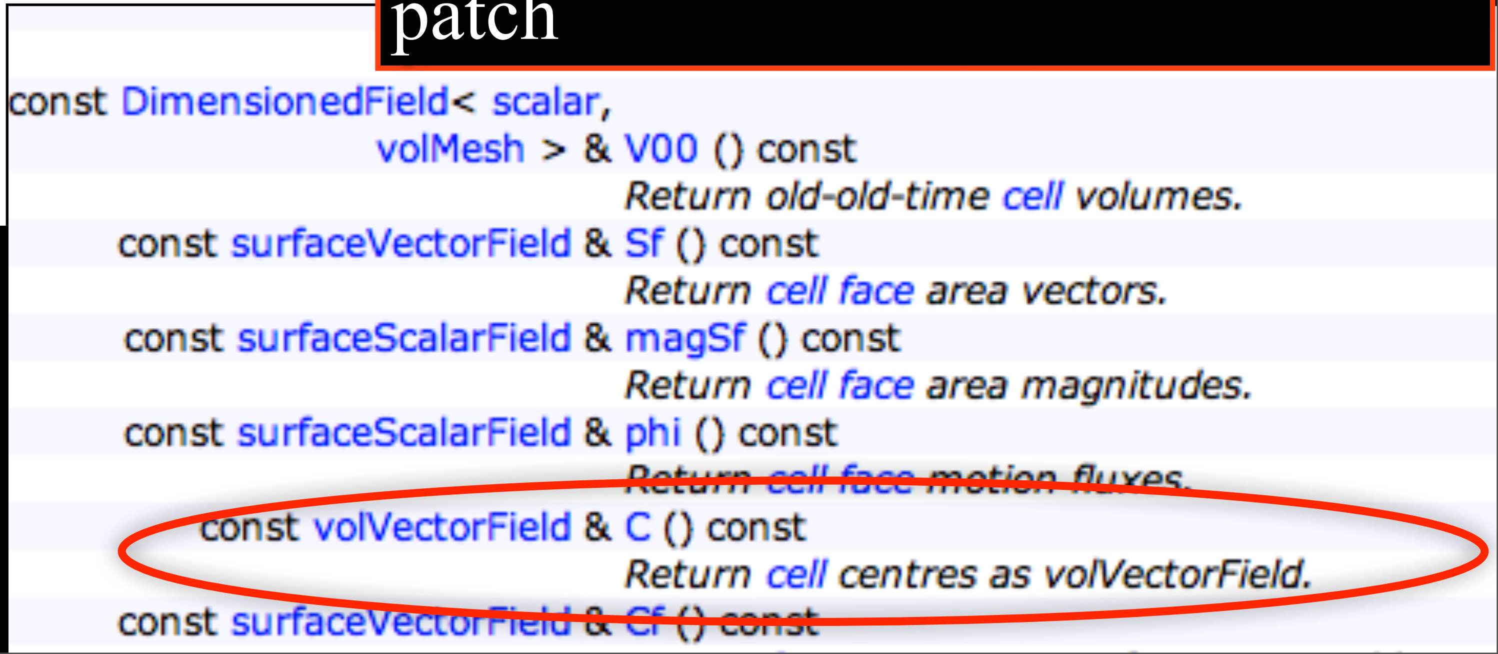


```
forAllConstIter(labelHashSet, patchSet_, iter)
{
    label patchi = iter.key();

    vectorField Md = mesh.C().boundaryField()[patchi] - CofR_;

    vectorField pf = Sfb[patchi]*(p.boundaryField()[patchi] - pRef);

    fm.first().first() += rho(p)*sum(pf);
    fm.second().first() += rho(p)*sum(Md ^ pf);

    vectorField vf = Sfb[patchi] & devRhoReffb[patchi];

    fm.first().second() += sum(vf);
    fm.second().second() += sum(Md ^ vf);
}
```

OpenFOAM iterator. it corresponds to a for cycle

mesh is object of the class fvMesh

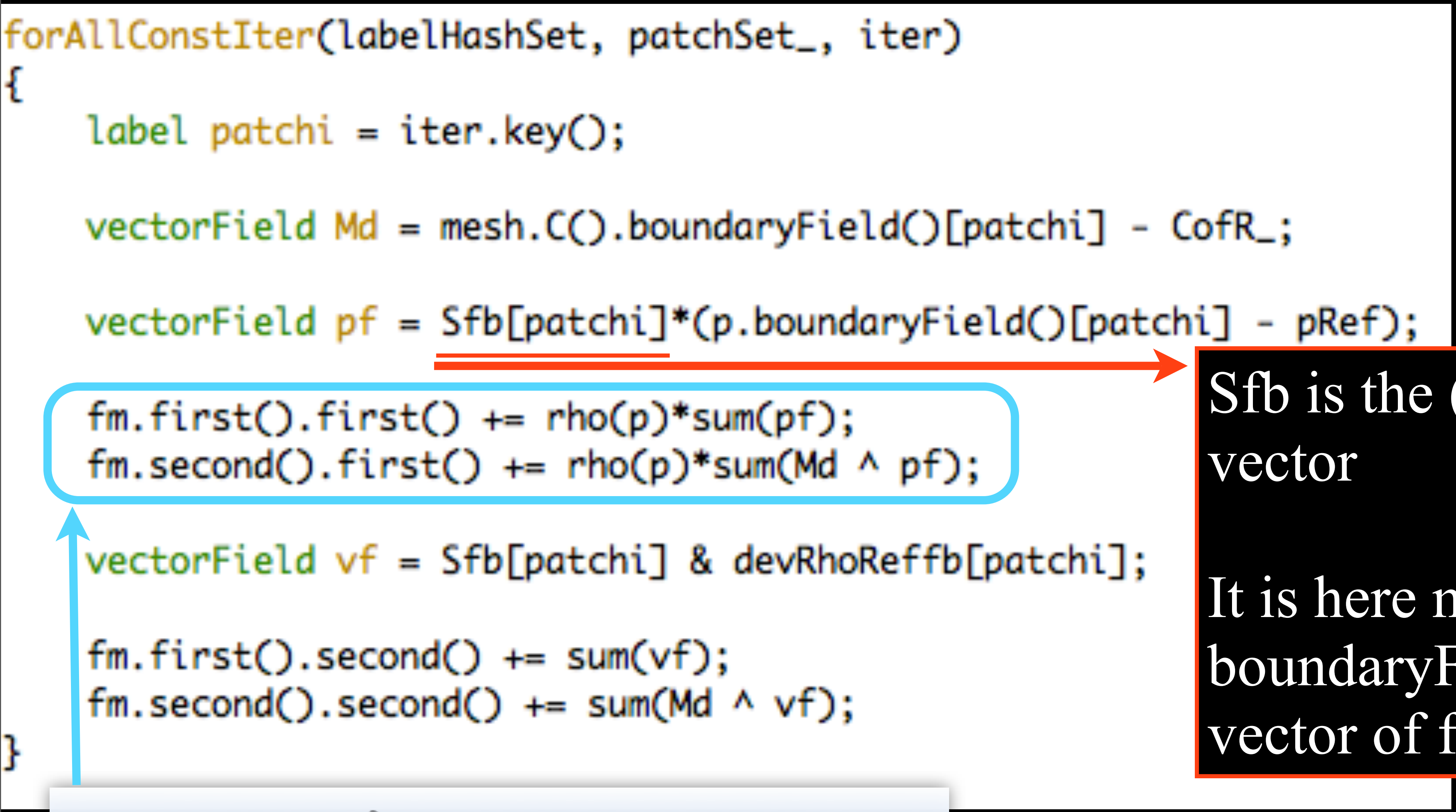
The expression returns a vector with the cell centres of the chosen patch



```
const DimensionedField< scalar,
                        volMesh > & V00 () const
                                     Return old-old-time cell volumes.
   const surfaceVectorField & Sf () const
                                     Return cell face area vectors.
   const surfaceScalarField & magSf () const
                                     Return cell face area magnitudes.
   const surfaceScalarField & phi () const
                                     Return cell face motion fluxes.
        const volVectorField & C () const
                                     Return cell centres as volVectorField.
   const surfaceVectorField & Cf () const
```

Calculating the forces

The virtual function write() is called during the execution. This calls forces::calcForcesMoment(), where the calculation is performed



```
forAllConstIter(labelHashSet, patchSet_, iter)
{
    label patchi = iter.key();

    vectorField Md = mesh.C().boundaryField()[patchi] - CofR_;

    vectorField pf = Sfb[patchi]*(p.boundaryField()[patchi] - pRef);

    fm.first().first() += rho(p)*sum(pf);
    fm.second().first() += rho(p)*sum(Md ^ pf);

    vectorField vf = Sfb[patchi] & devRhoReffb[patchi];

    fm.first().second() += sum(vf);
    fm.second().second() += sum(Md ^ vf);
}
```

Sfb is the (reference to) the face area vector

It is here multiplied for the pressure boundaryField => pf returns the vector of forces on the chosen patch

$$F = \rho \int p dA = \rho \sum p_i A_i$$

$$M = F \times r = \rho \sum f_i \times r_i$$

Re-compiling the forces library

The basic idea of the openFOAM environement is:
find something similar and modify it as you like, but
DO NOT TOUCH THE ORIGINAL SOURCES!

STEP 1: copy the forces directory from the original location into another directory

STEP 2: copy also the Make folder

STEP 3: substitute strings and modify (all) the file names
(sed ‘s/forces/Myforces/g’ forces.C > Myforces.C)

STEP 4: modify the local functionObject.H file (add the new class to the list of loadable functions)

```
#include "Myforces.H"
   ...
namespace Foam
   {
     typedef OutputFilterFunctionObject<Myforces>forcesFunctionObject;
   }
   ...
```

Re-compiling the forces library

STEP 4: Modify the Make/files:

```
Myforces.C
forcesFunctionObject.C

LIB = $(FOAM_USER_LIBBIN)/LibMyforces
```

STEP 5: modify the Make/options file:

```
EXE_INC = \
    .... all what was already there ...
    -I$(LIB_SRC)/postProcessing/functionObjects/forces/lnInclude
```

include all what was needed by the original library!

STEP 6: compile with wmake libso

Re-compiling the forces library

STEP 7: Add the entries in the controlDict, in order for the library to be loaded and used:

```
libs ( "libMyforces.dylib" ) ;

functions
 (
   Myforces
   {
    type Myforces;
    functionObjectLibs ("libMyforces.dylib");
    outputControl outputTime;
    patches (BottWall);
    pName p;
    Uname U;
    rhoName rhoInf;
    rhoInf 1.0;
    pRef 0;
    CofR (0 0 0);
   }
);
```

Load the library (.dylib on MAC; .so on Linux)

Search in the library for the entry called

Use the library

References

Several examples can be found on-line:

http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2007/
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2008/
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2009/
http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2010/

See for example the work of A. Gonzales.
But have a deep look at the whole web-site,
there's a lot of enlightening material!!

PART 4

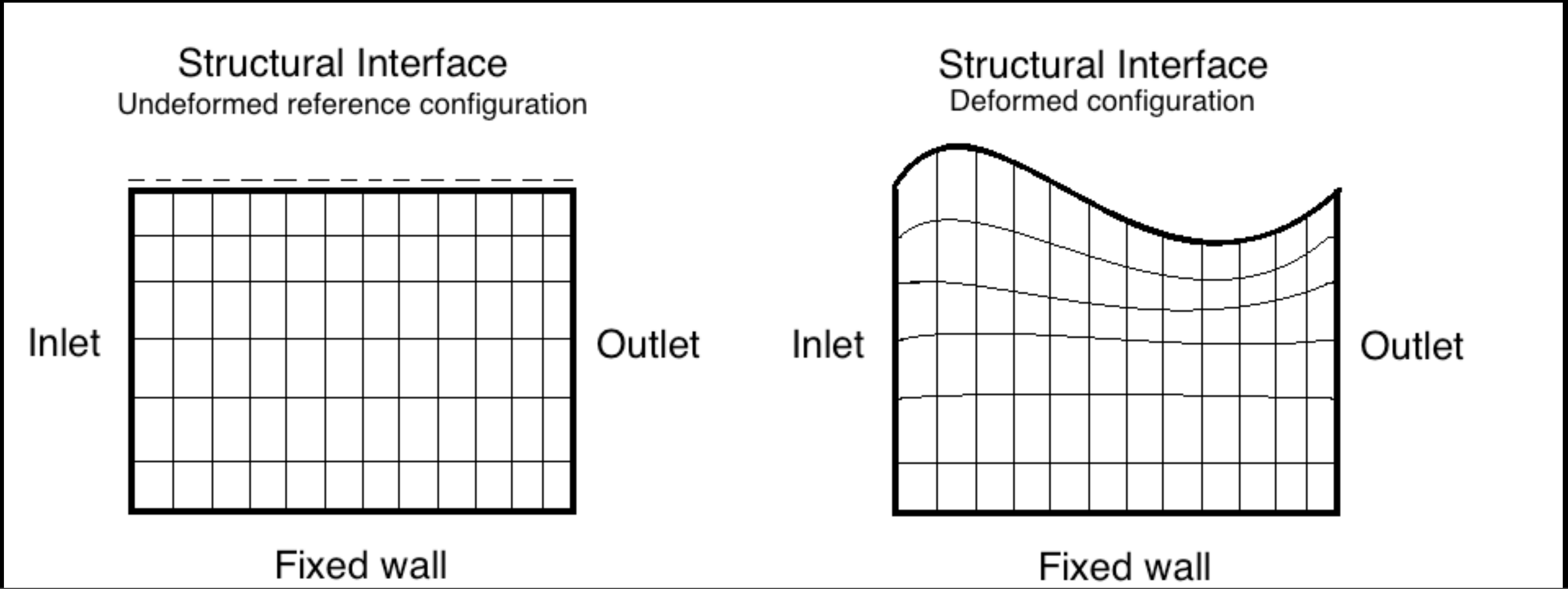
Adding new classes in OpenFOAM
the FSInterface class

Scope of this class

- The class is designed to manage the mesh motions. It is used in the main of the solver pimpleDyMFOAM:
  - establish the communication (if needed) with the external solver trough MPI
  - send pressure data and retrieve mesh displacement data
  - communicate with the AitkenControl class, in charge for calculating the dynamic relaxation factor:

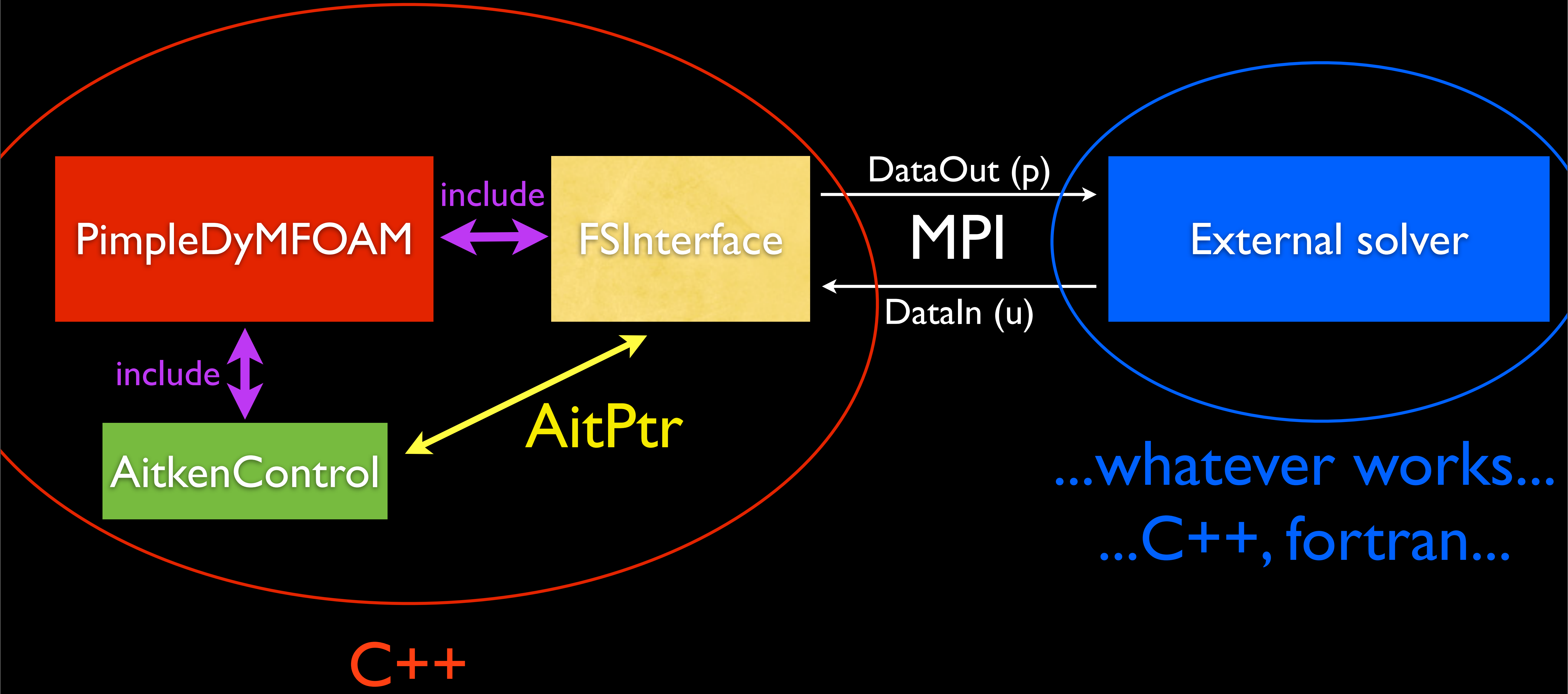
$$u_{k+1} = \omega_k \, \tilde{u}_{k+1} + (1 - \omega_k) \, u_k$$

  - move the fluid mesh (ALE framework)



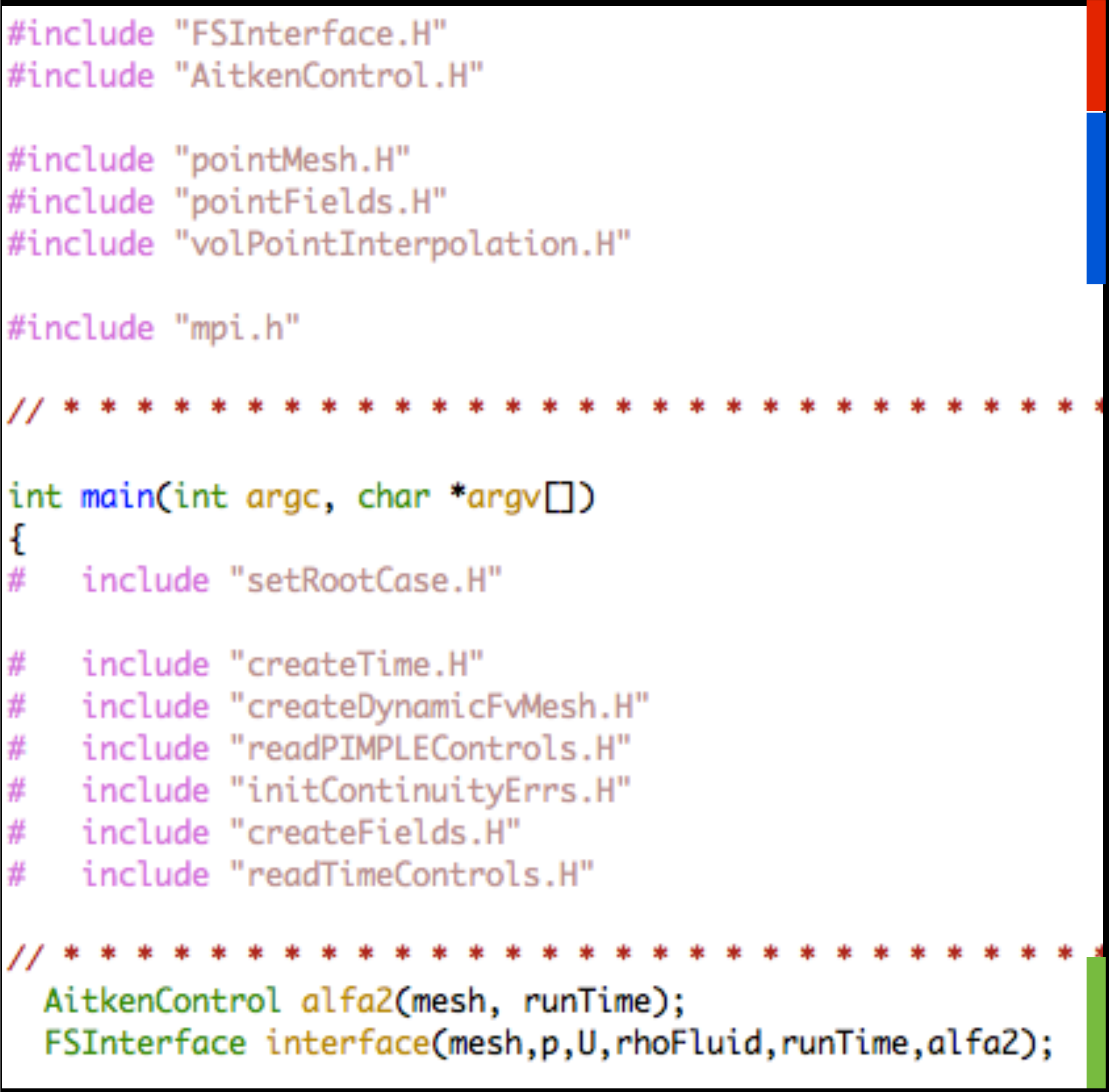
Scope of this class

Multiple Program Multiple Data type environment, the external solver is "spawned" during the execution time. This generates a communicator we can use for exchanging data (white arrows)



Use of the class

In the main solver: Include and declare



```
#include "FSInterface.H"
#include "AitkenControl.H"

#include "pointMesh.H"
#include "pointFields.H"
#include "volPointInterpolation.H"

#include "mpi.h"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

int main(int argc, char *argv[])
{
#   include "setRootCase.H"

#   include "createTime.H"
#   include "createDynamicFvMesh.H"
#   include "readPIMPLEControls.H"
#   include "initContinuityErrs.H"
#   include "createFields.H"
#   include "readTimeControls.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  AitkenControl alfa2(mesh, runTime);
  FSInterface interface(mesh,p,U,rhoFluid,runTime,alfa2);
```

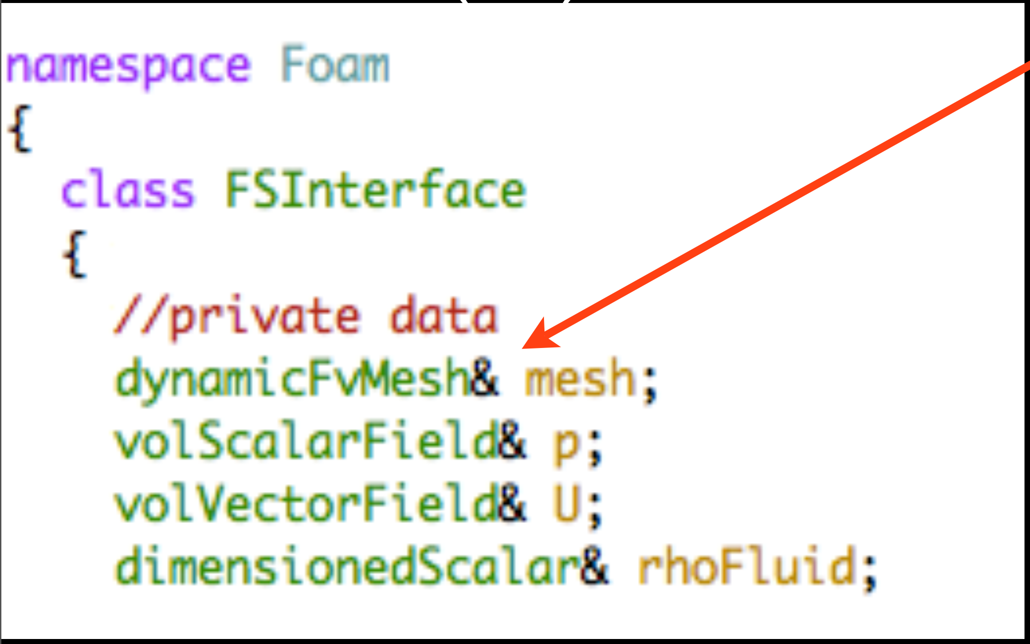
new classes definition

Foam classes needed for the mesh motion

Declare the new objects using the right arguments!

Constructor of FSInterface:

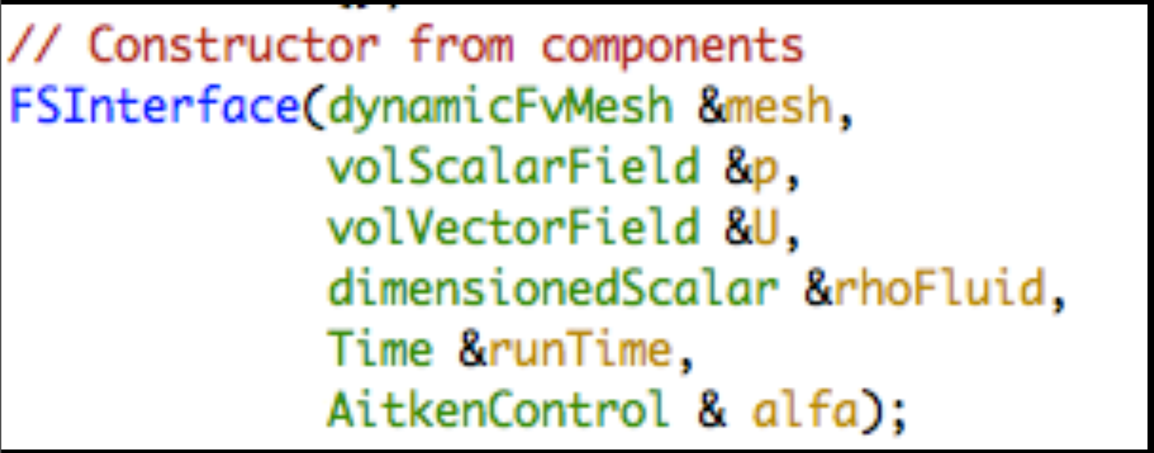
header (.H) file



```
namespace Foam
{
  class FSInterface
  {
    //private data
    dynamicFvMesh& mesh;
    volScalarField& p;
    volVectorField& U;
    dimensionedScalar& rhoFluid;
```

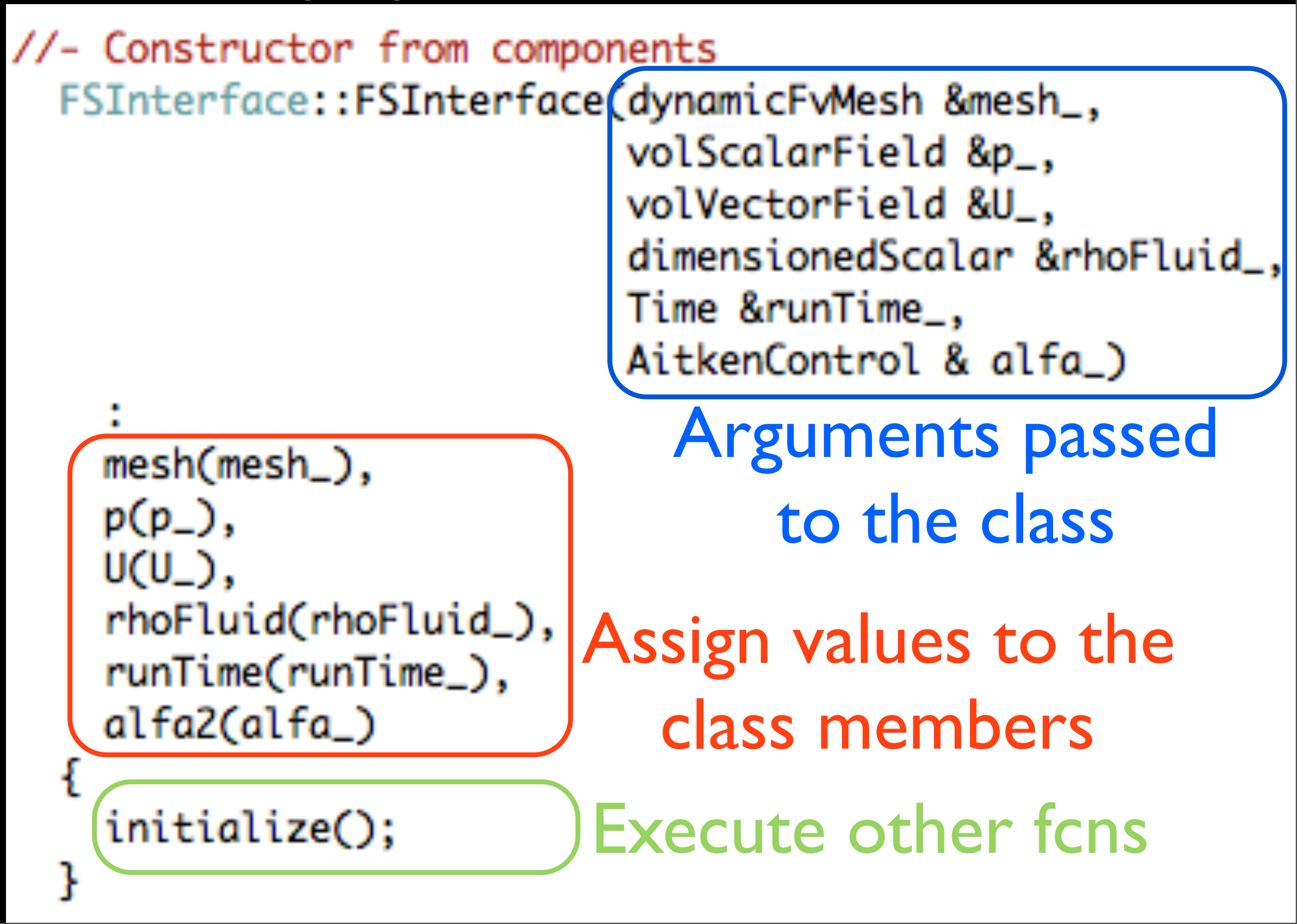
References are constant: they MUST be initialised at the creation. In this case this is done by passing the values to the constructor

... a lot of other stuff...



```
// Constructor from components
FSInterface(dynamicFvMesh &mesh,
            volScalarField &p,
            volVectorField &U,
            dimensionedScalar &rhoFluid,
            Time &runTime,
            AitkenControl & alfa);
```

source (.C) file



```
//- Constructor from components
  FSInterface::FSInterface(dynamicFvMesh &mesh_,
                           volScalarField &p_,
                           volVectorField &U_,
                           dimensionedScalar &rhoFluid_,
                           Time &runTime_,
                           AitkenControl & alfa_)

    :
    mesh(mesh_),
    p(p_),
    U(U_),
    rhoFluid(rhoFluid_),
    runTime(runTime_),
    alfa2(alfa_)
  {
    initialize();
  }
```

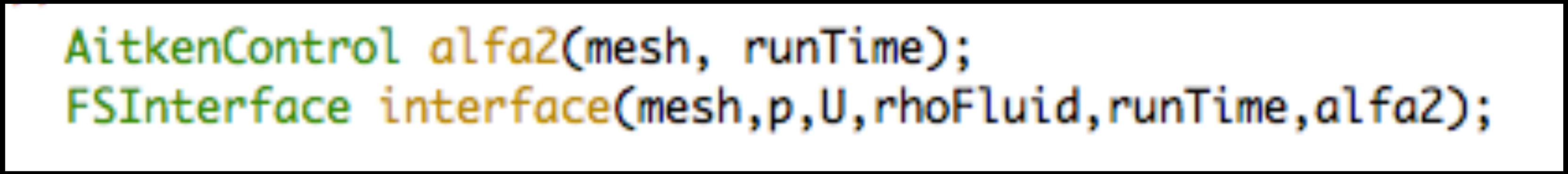
Arguments passed to the class

Assign values to the class members

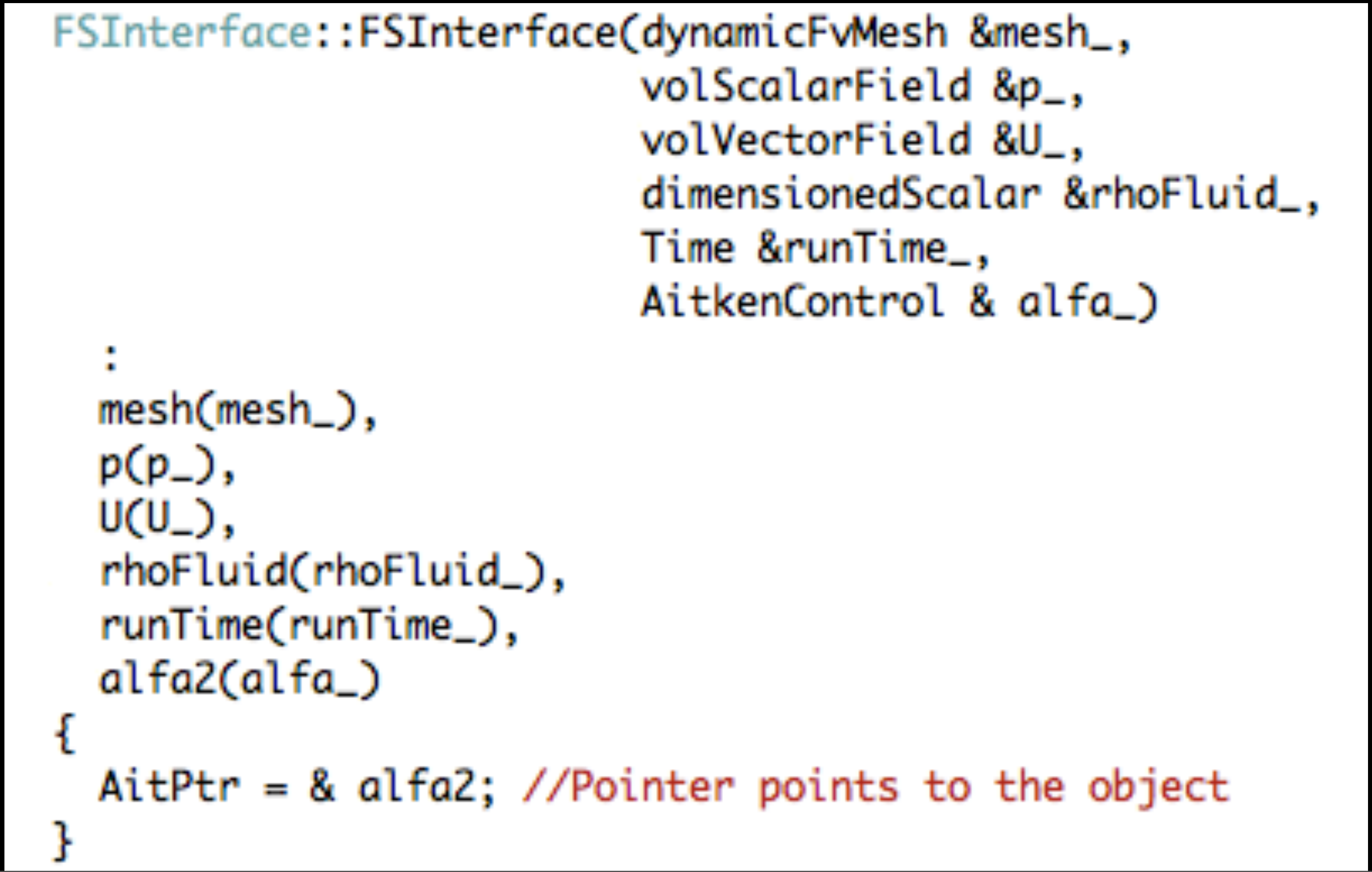
Execute other fcns

linking to the AitkenControl class:

An object of the type AitkenControl is created right before the object FSInterface. A reference to this object is passed to the constructor. This reference is stored in a pointer main code:


```
AitkenControl alfa2(mesh, runTime);
FSInterface interface(mesh,p,U,rhoFluid,runTime,alfa2);
```

alfa2 is instantiated in the constructor, the pointer is referenced also in the constructor:

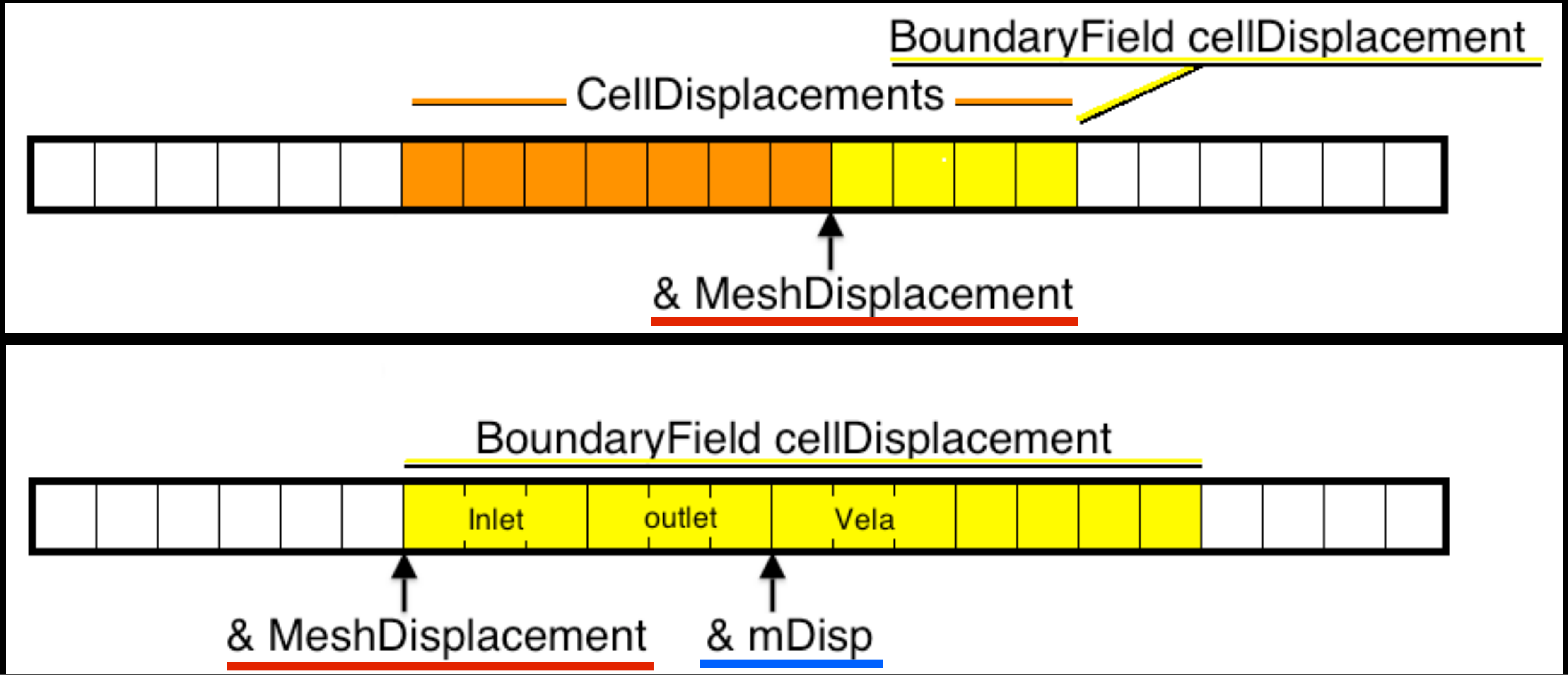

```
FSInterface::FSInterface(dynamicFvMesh &mesh_,
                         volScalarField &p_,
                         volVectorField &U_,
                         dimensionedScalar &rhoFluid_,
                         Time &runTime_,
                         AitkenControl & alfa_)
  :
  mesh(mesh_),
  p(p_),
  U(U_),
  rhoFluid(rhoFluid_),
  runTime(runTime_),
  alfa2(alfa_)
{
  AitPtr = & alfa2; //Pointer points to the object
}
```

Searching the mesh motion entries

Depending upon the motion solver, the mesh motion is stored in a field "pointDisplacements", "cellDisplacement" or "motionU".

A field in OpenFOAM is defined as: internalField + boundaryField

Imposing the motion of a boundary means writing the motion in the correspondent entry of the boundaryField. For example, "&MeshDisplacement" is the address of the BoundaryField, while "&mDisp" is the address of the mesh interface we want to move



Searching the mesh motion entries

```
IOdictionary couplingDict
(
    IOobject
        (
            "CouplingDict",
            runTime.constant(),
            mesh,
            IOobject::MUST_READ,
            IOobject::NO_WRITE
        )
);
```

Opens the FOAM dictionary named "couplingDict", to be read only. The dict must be placed in the folder constant

```
word temp(couplingDict.lookup("fluidPatch"))
word interface = temp;
```

Searches and reads the entry "fluidPatch" in CouplingDict

```
label fluidPatchID = mesh.boundaryMesh().findPatchID(interface);
```

Searches the the entry "fluidPatchName" in the mesh boundaryField. Returns a (integer) label: the id of the patch in the mesh order

Searching the mesh motion entries

We need now to find the references to the mesh motion boundaryField. Using displacementLaplacian...

```
pointVectorField & PointDisplacement =
    const_cast<pointVectorField&>(mesh.objectRegistry::
        lookupObject<pointVectorField>("pointDisplacement"));

pDispPtr = & ( refCast<vectorField>(PointDisplacement.boundaryField()[fluidPatchID]));
```

Search in the objectRegistry of the mesh an object of the type: pointVectorField called pointDisplacement. Whatever its type, impose (const_cast) to be a reference of the type PointVectorField

The entry number "fluidPatchID" is the reference to the interface mesh motion. Store this reference into a pointer

The mesh motion is imposed using the surcharged operator == :

```
PointDisplacement.boundaryField()[ fluidSideI ] == U_kp1
```

Searching the mesh motion entries

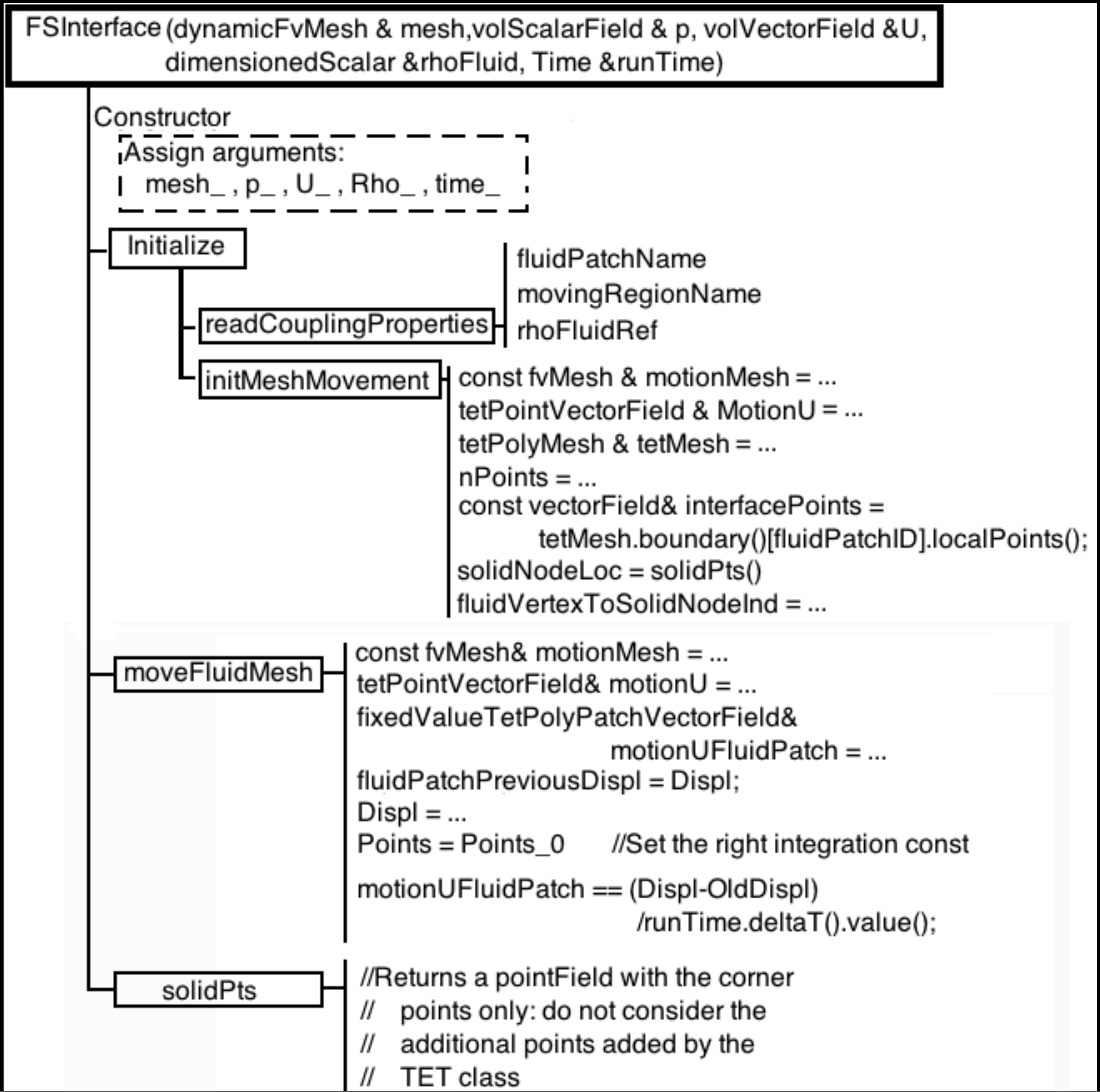
We need now to find the references to the mesh motion boundaryField. Using laplaceFaceDecomposition...

```
const fvMesh& motionMesh =
    runTime.objectRegistry::lookupObject<fvMesh>(movingRegionName);

tetPointVectorField& motionU =
    const_cast<tetPointVectorField&>
    ( motionMesh.objectRegistry::lookupObject<tetPointVectorField>("motionU") );

tetPolyMesh& tetMesh = const_cast<tetPolyMesh&>(motionU.mesh());

motionUFluidPatchPtr = &
    refCast<fixedValueTetPolyPatchVectorField>
        (  motionU.boundaryField()[fluidPatchID]  );
```

The mesh motion is imposed using the surcharged operator == :

```
* motionUFluidPatchPtr == ( U_kp1 - U_old ) / runTime.deltaT().value();
```

Scheme of the class



FSInterface (dynamicFvMesh & mesh,volScalarField & p, volVectorField &U,
dimensionedScalar &rhoFluid, Time &runTime)

Constructor

Assign arguments:
mesh_ , p_ , U_ , Rho_ , time_

Initialize

readCouplingProperties
- fluidPatchName
- movingRegionName
- rhoFluidRef

initMeshMovement

```
const fvMesh & motionMesh = ...
tetPointVectorField & MotionU = ...
tetPolyMesh & tetMesh = ...
nPoints = ...
const vectorField& interfacePoints =
        tetMesh.boundary()[fluidPatchID].localPoints();
solidNodeLoc = solidPts()
fluidVertexToSolidNodeInd = ...
```

moveFluidMesh

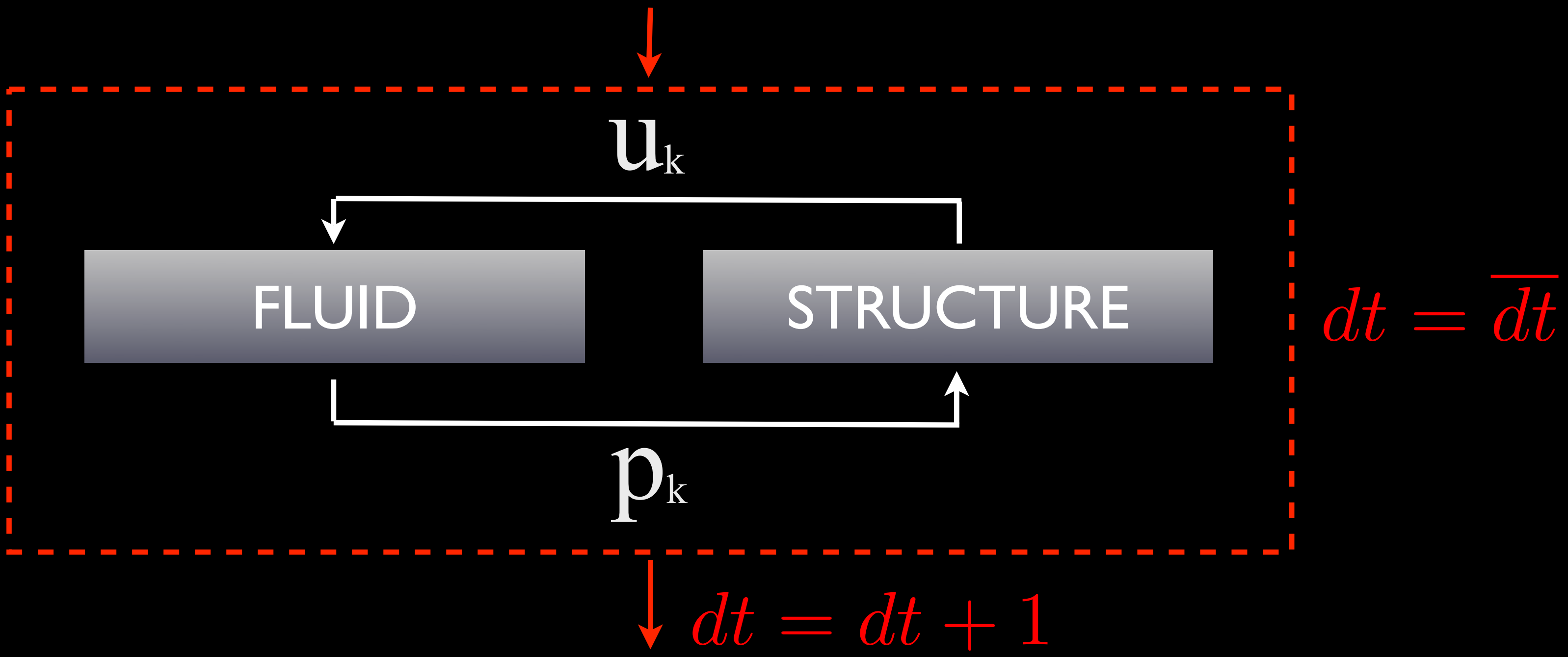
```
const fvMesh& motionMesh = ...
tetPointVectorField& motionU = ...
fixedValueTetPolyPatchVectorField&
                        motionUFluidPatch = ...
fluidPatchPreviousDispl = Displ;
Displ = ...
Points = Points_0       //Set the right integration const
motionUFluidPatch == (Displ-OldDispl)
                        /runTime.deltaT().value();
```

solidPts

```
//Returns a pointField with the corner
//   points only: do not consider the
//   additional points added by the
//   TET class
```

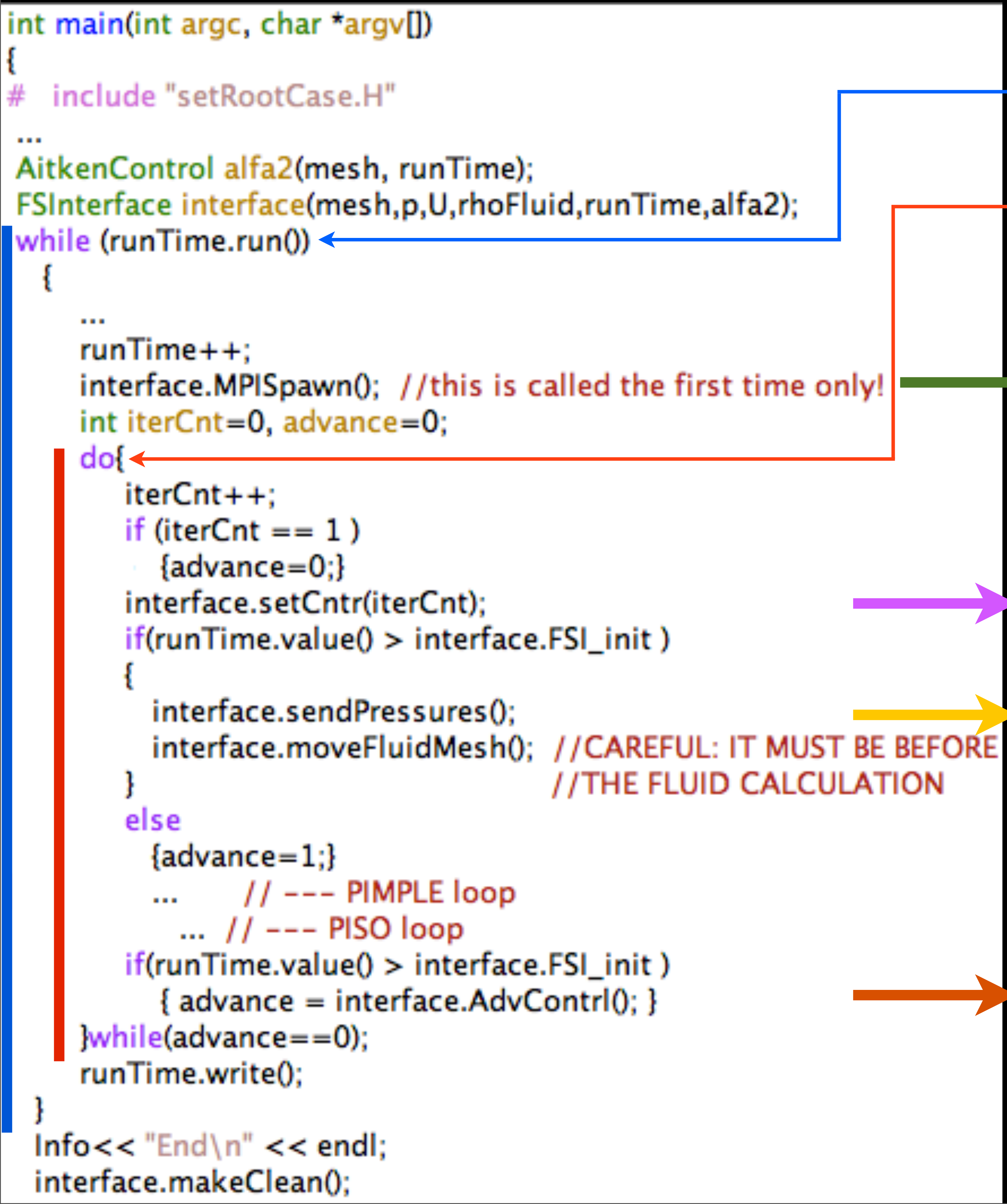
Algorithm

Implicit coupling: the equilibrium within the time-step is verified using a fixed-point between the fluid and the structural solver. Although this algorithm is computationally expensive, it is unconditionally stable



This is realised in OpenFOAM using a while loop within the time-step. The convergence criterion is verified by a function of the class FSInterface

Algorithm (main)



```
int main(int argc, char *argv[])
{
#   include "setRootCase.H"
 ...
 AitkenControl alfa2(mesh, runTime);
 FSInterface interface(mesh,p,U,rhoFluid,runTime,alfa2);
while (runTime.run())
  {
      ...
      runTime++;
      interface.MPISpawn();  //this is called the first time only!
      int iterCnt=0, advance=0;
      do{
          iterCnt++;
          if (iterCnt == 1 )
             {advance=0;}
          interface.setCntr(iterCnt);
          if(runTime.value() > interface.FSI_init )
          {
             interface.sendPressures();
             interface.moveFluidMesh();  //CAREFUL: IT MUST BE BEFORE
          }                              //THE FLUID CALCULATION
          else
             {advance=1;}
             ...       // --- PIMPLE loop
               ...  // --- PISO loop
          if(runTime.value() > interface.FSI_init )
             { advance = interface.AdvContrl(); }
      }while(advance==0);
      runTime.write();
  }
  Info<< "End\n" << endl;
  interface.makeClean();
```

time loop

Aitken iterations loop

Launch the structural solver

Send the number of iteration to the interface (different calculations if iterCntr == 1)

Send the pressures, retreive the displacements

Convergence is checked by interface

Compiling

As for every FOAM application, we need to edit:

Make/files:

```
FSIfluidFOAMtet.C
EXE = $(FOAM_USER_APPBIN)/FSIfluidFOAMtetMPI
```

$\rightarrow$ Source file

$\rightarrow$ Name and path of the compiled application

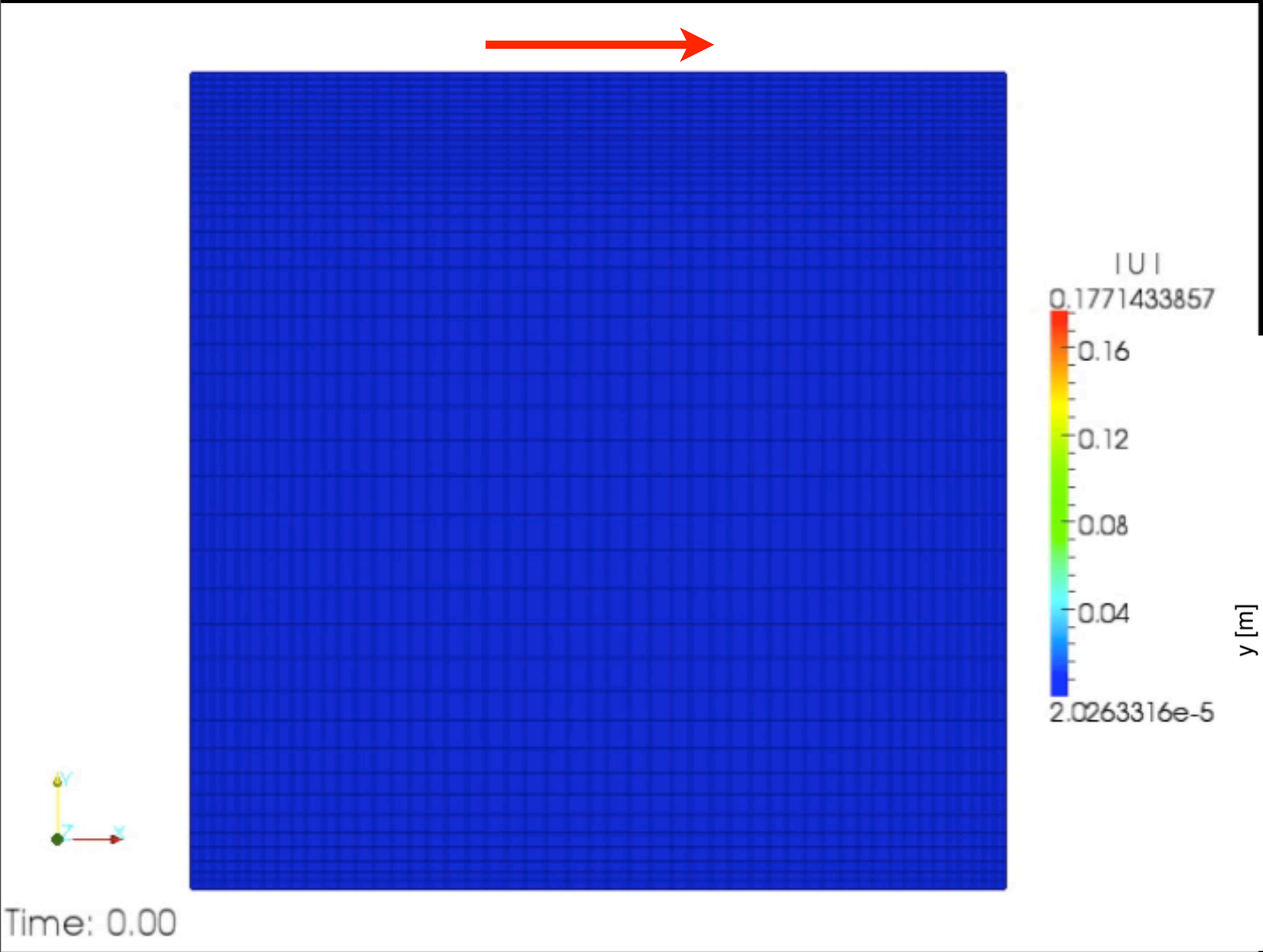
Make/options:

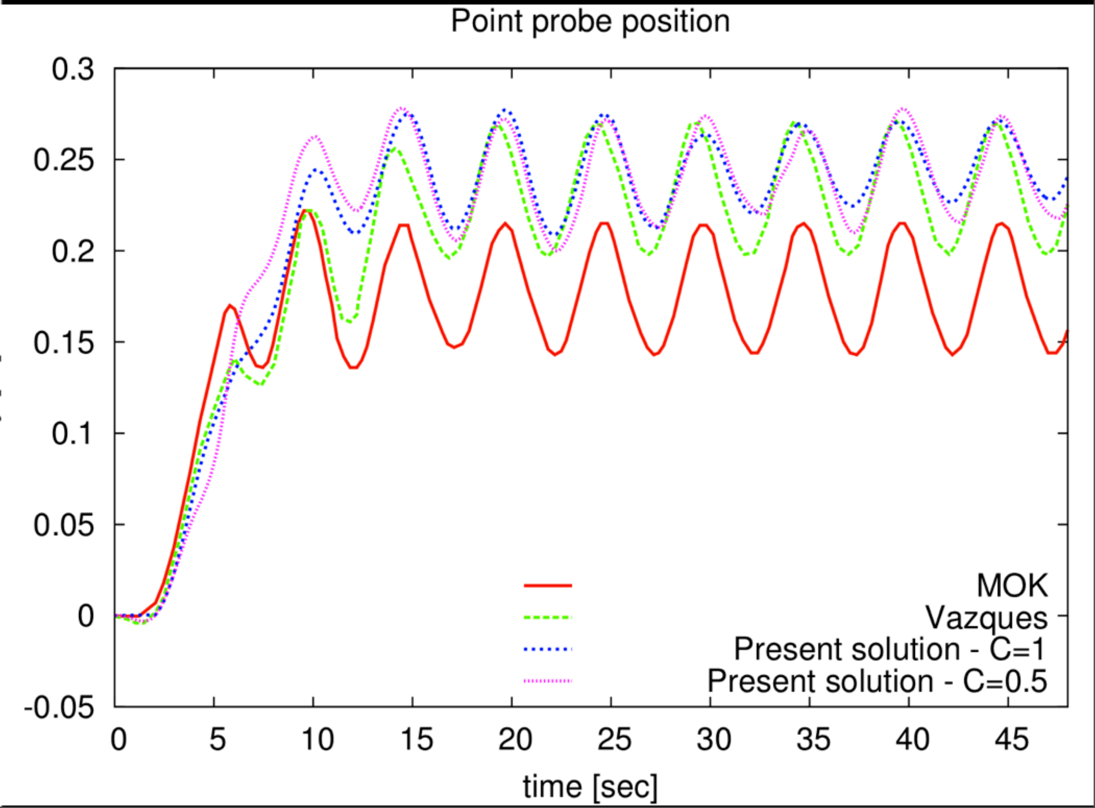
```
include $(RULES)/mplib$(WM_MPLIB)
EXE_INC = \
      ...all remains as in the original file...
EXE_LIBS = \
   ...all remains as in the original file, but add:
    -lmpi \
    $(WM_DECOMP_LIBS)
```

And finally some results!

FLUID DENSITY	FLUID VISCOSITY	STRUCTURE's THICKNESS	STRUCTURE's DENSITY	STRUCTURE's POISSON RATTIO	STRUCTURE's YOUNG'S MODULUS
ρ_f	ν_f	t_s	ρ_s	ν_s	E_s
Kg/m^3	m^2/s	m	Kg/m^3	-	N/m^2
1.0	0.01	0.002	500	0	250

$$u = 1 - cos(2\pi t/5)$$




Point probe position
y [m]
time [sec]
MOK
Vazques
Present solution - C=1
Present solution - C=0.5