A walk through some OpenFOAM code: Vector

A walk through some OpenFOAM code: Vector

Prerequisites

- You have a basic knowledge in object oriented C++ programming.
- You have a basic knowledge in the structure of OpenFOAM programming.

Learning outcomes

- You will gain experience in reading OpenFOAM classes and figure out how they work.

The Vector directory

- Let's have a look at some examples in the OpenFOAM `Vector` class:
  `$FOAM_SRC/OpenFOAM/primitives/Vector`
  (go there while looking at the following slides)
  To which library does it belong?
- We find (version dependent):

```
boolVector       doubleVector   labelVector   vector     VectorI.H
complexVector    floatVector    lists    Vector.H
```

  - The `Vector*` files are for the templated `Vector` class (capital first letter `V` means that it is a templated class). Same name as directory means that they are the main files!
  - Inline functions must be implemented in the class *declaration* file, since they must be inlined without looking at the class *definition* file. In OpenFOAM there are usually files named as `VectorI.H` containing `inline` functions, and those files are included in the corresponding `Vector.H` file. There is no `*.C` file in the `Vector` class, since all functions are inlined.
  - Directories `*{V,v}ector` (except `bool*`) are `typedef` for `Vector` of `complex`, `double`, `float`, `label` and `scalar`. The directory `lists` defines lists of vectors.
    What is a scalar? See `$FOAM_SRC/OpenFOAM/primitives/Scalar/scalar/scalarFwd.H`

Vector description

Let's have a close look at the `Vector` class!

It is usually good to read the class description, in `Vector.H`:

```
Description
    Templated 3D Vector derived from VectorSpace adding construction from
    3 components, element access using x(), y() and z() member functions and
    the inner-product (dot-product) and cross product operators.

    A centre() member function which returns the Vector for which it is called
    is defined so that point which is a typedef to Vector\<scalar\> behaves as
    other shapes in the shape hierarchy.
```

Vector.H file header and footer

The `Vector.H` file header is given by:

```
#ifndef Vector_H
#define Vector_H

#include "contiguous.H"
#include "VectorSpace.H"

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

namespace Foam
{

// Forward Declarations
template<class T> class List;
```

You already know what all of this means.

The footer shows that we can think of `VectorI.H` as a part of `Vector.H`:

```
#include "VectorI.H"
```

Vector inheritance

The class declaration shows that the `Vector` class inherits from (read: “is a”) `VectorSpace`:

```
template<class Cmpt>
class Vector
:
    public VectorSpace<Vector<Cmpt>, Cmpt, 3>
{
```

The `VectorSpace` class is found in `$FOAM_SRC/OpenFOAM/primitives/VectorSpace`, where (in `VectorSpace.H`) the tree template parameters are defined as:

```
template<class Form, class Cmpt, direction Ncmpts>
```

I.e., a `Vector` has `3` components of type `Cmpt`, and it has all the attributes of the `VectorSpace` class with the same visibility of those attributes.

We will not look at all the details of the `VectorSpace` class now, but it is important to remember this inheritance and later check up what needs to be checked up in that class!

Some Vector member data

In `Vector.H`, all new attributes are `public`, and some new member data is defined:

```
public:

   // Typedefs

        //- Equivalent type of labels used for valid component indexing
        typedef Vector<label> labelType;


    // Member Constants

        //- Rank of Vector is 1
        static constexpr direction rank = 1;


    //- Component labeling enumeration
    enum components { X, Y, Z };
```

`constexpr`: See `https://en.cppreference.com/w/cpp/language/constexpr`
`direction`: Integer, See: `$FOAM_SRC/OpenFOAM/primitives/direction/direction.H`
`enum`: See `https://en.cppreference.com/w/cpp/language/enum`

Vector constructor declarations

Default and specialized constructor *declarations* are found in `Vector.H`:

```
// Generated Methods

    //- Default construct
    Vector() = default;

    //- Copy construct
    Vector(const Vector&) = default;

    //- Copy assignment
    Vector& operator=(const Vector&) = default;


// Constructors

    //- Construct initialized to zero
    inline Vector(const Foam::zero);

    //- Copy construct from VectorSpace of the same rank
    template<class Cmpt2>
    inline Vector(const VectorSpace<Vector<Cmpt2>, Cmpt2, 3>& vs);

    //- Construct from three components
    inline Vector(const Cmpt& vx, const Cmpt& vy, const Cmpt& vz);

    //- Construct from Istream
    inline explicit Vector(Istream& is);
```

Vector constructor definitions

- The constructor *definitions* are usually found in the corresponding `.C` file, but since the constructors for the `Vector` are inlined they are found in the `VectorI.H` file, e.g.:

```
template<class Cmpt>
inline Foam::Vector<Cmpt>::Vector
(
    const Cmpt& vx,
    const Cmpt& vy,
    const Cmpt& vz
)
{
    this->v_[X] = vx;
    this->v_[Y] = vy;
    this->v_[Z] = vz;
}
```

  Here, `this` is a pointer to the object that is being constructed, i.e. we set the member data `v_` (inherited from class `VectorSpace`) to the values supplied as arguments to the constructor, using enumerators `X`, `Y` and `Z` for the three components.

- It is here obvious that the member function `Vector` belongs to the class `Vector`, which is templated with `Cmpt`, and that it is a constructor since it has the same name as the class.

Vector access functions

- Some access functions are *declared* in `Vector.H`, e.g.:

```
inline const Cmpt& x() const;
inline Cmpt& x();
```

and *defined* in `VectorI.H`:

```
template<class Cmpt>
inline const Cmpt& Foam::Vector<Cmpt>::x() const
{
    return this->v_[X];
}

template<class Cmpt>
inline Cmpt& Foam::Vector<Cmpt>::x()
{
    return this->v_[X];
}
```

Again, the pointer `this` points at the object that is used to call the function.
The first one is for `const` objects, and the second one can manipulate non-`const` objects.

Vector member functions

Two member functions are *declared* in `Vector.H`:

```
//- Normalise the vector by its magnitude
inline Vector<Cmpt>& normalise();


//- Return *this (used for point which is a typedef to Vector<scalar>.
inline const Vector<Cmpt>& centre
(
    const Foam::List<Vector<Cmpt>>&
) const;
```

and *defined* in `VectorI.H` (not shown, since we don't care about the definitions at the moment).

We will get back to the last ("Traits") part of `Vector.H` later.

Vector operators

- Some `Vector` operators are defined in `VectorI.H`, e.g.:

```
template<class Cmpt>
inline typename innerProduct<Vector<Cmpt>, Vector<Cmpt>>::type
operator&(const Vector<Cmpt>& v1, const Vector<Cmpt>& v2)
{
    return Cmpt(v1.x()*v2.x() + v1.y()*v2.y() + v1.z()*v2.z());
}

template<class Cmpt>
inline Vector<Cmpt> operator^(const Vector<Cmpt>& v1, const Vector<Cmpt>& v2)
{
    return Vector<Cmpt>
    (
        (v1.y()*v2.z() - v1.z()*v2.y()),
        (v1.z()*v2.x() - v1.x()*v2.z()),
        (v1.x()*v2.y() - v1.y()*v2.x())
    );
}
```

They can be changed for a specific type of vector, such as in `complexVectorI.H`.

We will get back to the strangely written return type of `innerProduct` later ("Traits").

Specialization of the Vector class - vector (Vector<scalar>)

The `Vector` class is templated, so it can work for different component types. The most common one is `Vector<scalar>`. This is implemented in the `vector` directory, which has the files:

```
vector.C  vector.H
```

The most important line in `vector.H` is:

```
typedef Vector<scalar> vector;
```

saying that a `vector` is mainly a `Vector<scalar>`.

The rest of that file contains "Traits", which will be discussed later.

Specialization of the Vector class - vector (Vector<scalar>)

The base class `VectorSpace` has declared (in `VectorSpace.H`) that the sub-classes should have some static member data:

```
// Static Data Members

    static const char* const typeName;
    static const char* const componentNames[];
    static const Form zero;
    static const Form one;
    static const Form max;
    static const Form min;
    static const Form rootMax;
    static const Form rootMin;
```

These are not given any value in neither the `VectorSpace` nor the `Vector` classes, since they depend on the type of component.

Specialization of the Vector class - vector (Vector<scalar>)

The so far un-set static members of base class `VectorSpace` are set in `vector.C`:

```
template<>
const char* const Foam::vector::vsType::typeName = "vector";
template<>
const char* const Foam::vector::vsType::componentNames[] = {"x", "y", "z"};
template<>
const Foam::vector Foam::vector::vsType::zero(vector::uniform(0));
template<>
const Foam::vector Foam::vector::vsType::one(vector::uniform(1));
template<>
const Foam::vector Foam::vector::vsType::max(vector::uniform(VGREAT));
template<>
const Foam::vector Foam::vector::vsType::min(vector::uniform(-VGREAT));
template<>
const Foam::vector Foam::vector::vsType::rootMax(vector::uniform(ROOTVGREAT));
template<>
const Foam::vector Foam::vector::vsType::rootMin(vector::uniform(-ROOTVGREAT));
```

We will not go into the details of this now.

Other specialization of the Vector class

The other specializations are defined in a similar way:

```
complexVector
doubleVector
floatVector
labelVector
```

However, `boolVector` is implemented another way, since it "does not share very many vector-like characteristics".

Traits

At the end of `Vector.H` we see a section named "Traits".

Traits is a way to specialize a templated class.

The section starts with (e.g.):

```
//- Data are contiguous if component type is contiguous
template<class Cmpt>
struct is_contiguous<Vector<Cmpt>> : is_contiguous<Cmpt> {};
```

This says that if the `Cmpt` is contiguous (composed only of `Foam::scalar` elements), also `Vector<Cmpt>` is contiguous. Those structures (classes) are defined in:

`$FOAM_SRC/OpenFOAM/primitives/contiguous/contiguous.H`

E.g.:

```
// Base definition for (integral | floating-point) as contiguous
template<class T>
struct is_contiguous
:
    std::is_arithmetic<T>
{};
```

See: `https://en.cppreference.com/w/cpp/types/is_arithmetic`
Search: `grep -r "if (is_contiguous" $FOAM_SRC`

Traits

We also see e.g.:

```
template<class Cmpt>
class typeOfRank<Cmpt, 1>
{
public:

    typedef Vector<Cmpt> type;
};
```

This declares a class named `typeOfRank`, templated with `Cmpt` (any type of component) and rank `1` (for a vector, which would be `2` for a tensor). The class only defines a `typedef` such that `type` means `Vector<Cmpt>`.

The class is just waiting to be used.

If there is one for vector, there should be one for each rank?

Traits

Search for declarations of `class typeOfRank`:

```
$ grep -r "class typeOfRank" $FOAM_SRC
$FOAM_SRC/OpenFOAM/primitives/Tensor/Tensor.H:class typeOfRank<Cmpt, 2>
$FOAM_SRC/OpenFOAM/primitives/Vector/Vector.H:class typeOfRank<Cmpt, 1>
$FOAM_SRC/OpenFOAM/primitives/VectorSpace/products.H:class typeOfRank
$FOAM_SRC/OpenFOAM/primitives/VectorSpace/products.H:class typeOfRank<Cmpt, 0>
```

We see that `VectorSpace/products.H` declares the existence of the class:

```
template<class Cmpt, direction rank>
class typeOfRank
{};
```

It also defines the specialization for rank `0`:

```
template<class Cmpt>
class typeOfRank<Cmpt, 0>
{
public:

    typedef Cmpt type;
};
```

Then the `Vector` and `Tensor` classes add ranks `1` and `2`.

So, how can it be used?

Traits

Copy `$FOAM_APP/test/vector` and change `Test-vector.C` to:

```
#include "vector.H"
#include "tensor.H"
#include "IOstreams.H"

using namespace Foam;

int main(int argc, char *argv[])
{
    Info<< typeOfRank<scalar, 0>::type(1)<<endl;
    Info<< typeOfRank<scalar, 1>::type(1,2,3)<<endl;
    Info<< typeOfRank<bool, 1>::type(true,false,true)<<endl;
    Info<< typeOfRank<vector, 1>::type(vector(1,2,3),vector(1,2,3),vector(1,2,3))<<endl;
    Info<< typeOfRank<scalar, 2>::type(1,2,3,4,5,6,7,8,9)<<endl;

    return 0;
}
```

Compile and run!

We see that the class `::type` can be used as a type, e.g. `scalar(1)`, `vector(1,2,3)`.

So, where is it used?

Traits

Search for `typeOfRank` (where we get in addition to the declarations):

```
$ grep -r "typeOfRank" $FOAM_SRC
...
$FOAM_SRC/OpenFOAM/primitives/VectorSpace/products.H:    typedef typename typeOfRank
$FOAM_SRC/OpenFOAM/primitives/VectorSpace/products.H:    typedef typename typeOfRank
$FOAM_SRC/OpenFOAM/primitives/VectorSpace/products.H:    typedef typename typeOfRank
...
```

One of those is:

```
template<class arg1, class arg2>
class innerProduct
{
public:

    typedef typename typeOfRank
    <
        typename pTraits<arg1>::cmptType,
        direction(pTraits<arg1>::rank) + direction(pTraits<arg2>::rank) - 2
    >::type type;
};
```

This class also makes a `typedef` of `type`, which it gets from the class `typeOfRank`, using the `cmptType` of `arg1` and the ranks of both arguments (yielding `type` and `rank` of an inner product of arguments of that type).

There are also `outerProduct` and `crossProduct`, differing in the calculated `rank`.

This class is also just waiting to be used.

A quick look at `pTraits`...

Traits

In `$FOAM_SRC/OpenFOAM/primitives/pTraits/pTraits.H`, we see:

```
template<class PrimitiveType>
class pTraits
:
    public PrimitiveType
{
public:

    // Constructors

        //- Copy construct from primitive
        explicit pTraits(const PrimitiveType& p)
        :
            PrimitiveType(p)
        {}

        //- Construct from Istream
        explicit pTraits(Istream& is)
        :
            PrimitiveType(is)
        {}
};
```

The class inherits from the class of the template argument, which may be `Vector<Cmpt>`, which inherits from `VectorSpace`.

In `VectorSpace.H` we see: `typedef Cmpt cmptType;`

In `Vector.H` we see: `static constexpr direction rank = 1;`

Traits

In `VectorI.H` we see the definition of the inner product as:

```
template<class Cmpt>
inline typename innerProduct<Vector<Cmpt>, Vector<Cmpt>>::type
operator&(const Vector<Cmpt>& v1, const Vector<Cmpt>& v2)
{
    return Cmpt(v1.x()*v2.x() + v1.y()*v2.y() + v1.z()*v2.z());
}
```

I.e., the returned type is specified above by:

```
typename innerProduct<Vector<Cmpt>, Vector<Cmpt>>::type
```

This uses the "trait" classes `innerProduct` and `typeOfRank` to determine the returned type depending on the types of the arguments to the inner product.

Please help me design an example of why it needs to be done this way!

We will not investigate the "Traits" of `vector.H` now.

The Vector class in Doxygen

- Use Doxygen to search for `Vector`, click on `Vector`, and click on `Vector< Cmpt >`
- Find all member data and member functions, including inherited ones.