
Programming Module

Session 1

Disclaimer

“This offering is not approved or endorsed by OpenCFD Limited, the producer of the OpenFOAM software and owner of the OPENFOAM® and OpenCFD® trade marks.”

C++ Programming

C++ in OpenFOAM

- Data Types & Declaration
- Control structures
- Functions
- Arrays
- Pointers
- Typedef & Classes
- Standard Template Library (STL)
- Some useful C++ programming in OpenFoam
- Five basic Classes in OpenFoam

C++ Programming

What better start could be in C++

Basic structure of a C++ program

```
#include <iostream>
using namespace std;
// main() is where program execution begins.
int main ()
{
    cout << "Hello OpenFOAM"; //prints Hello OpenFOAM
    return 0;
}
```

C++ Programming

C++ Datatypes

<u>Type</u>		<u>Keyword</u>
Boolean	bool	
Character		char
Integer	int	
Floating point		float
Double floating point	double	
Valueless		void
Wide character		wchar_t

Declaration

int a, b, c; → Declaration

int aa = 5.12; → Declaration and initialization, it will only read the integer part

float pi; → Declaration

pi = 3.14159; → Initialization

C++ Programming

Datatypes Modifiers

***char, int, and double* data types to have modifiers preceding**

The modifiers signed, unsigned, long, & short can be applied to **integer base types**.

signed

Unsigned

Long

Short



In addition, signed & unsigned can be applied to **char**, and long can be applied to **double**.

For example

- unsigned int y; → 4 bytes. Range from 0 to 4294967295
- signed int y; → 4 bytes. Range from -2147483648 to -2147483647

C++ Programming

C++ Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

Assignment Operator: =

Arithmetic Operators: +, -, \, *, %, ,

Increase/decrease: ++, --

Relational Operators: ==, !=, >, <, >=, <=

Logical Operators: &&, ||, !

Bitwise Operators: &, |, ^, ~, <<, >>

Assignment Operators: +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=

Misc Operators: ?, comma (,), arrow (->), sizeof(), and so on.

C++ Programming

C++ Control Structures

- *Control structures are portions of program code that contain statements within them and,*
- *Depending on the circumstances, execute these statements in a certain way.*

*There are typically two kinds: **conditionals** and **loops**.*

Conditional structure: if and else

The if keyword is used to execute a statement or block only if a condition is fulfilled.

Syntax:

if (condition) statement

For example:

```
if (x == 10e-05)
```

```
cout << "solution is converged";
```

will print the "solution is converged" only if the value stored in the x variable is indeed 10e-05.

C++ Programming

C++ Control Structures

Conditional structure: if and else

The if - else structures can be concatenated with the intention of verifying a range of values.

Example:

```
if (x > 0)
    cout << "x is positive";
else if (x < 0)
    cout << "x is negative";
else
    cout << "x is 0";
```

The conditional structures can be nested.

C++ Programming

C++ Control Structures

Conditional structure: While loop

loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

Syntax:

while (expression) statement

```
while (n>0)
{
    cout << "n is a positive number";
    n = n - 1; //We can also use --n
}
```

C++ Programming

C++ Control Structures

Conditional structure: do-while loop

It does the same as the while loop, except that the condition in the do-while loop is evaluated after the execution of the statement instead of before.

Syntax:

do statement while (condition)

```
do
{
    cout << "n is not equal to zero";
    n = n + 1; //We can also use n++
}
while (n != 0);
```

C++ Programming

C++ Control Structures

Conditional structure: for loop

It does the same as the while loop, except that the condition in the do-while loop is evaluated after the execution of the statement instead of before.

Syntax:

for (initialization; condition; increase) statement;

```
for (x = 0; x < 10; x++)  
{  
    cout << x << "endl";  
}  
return 0;
```

This program will print out the values 0 through 9, each on its own line.

C++ Programming

Functions

- *A function is a group of statements that together perform a task*
- *Every C++ program has at least one function which is the main () function*
- *You can divide your code into separate functions*
- *A function declaration tells the compiler about a function's name, return type, and parameters*
- *A function definition provides the actual body of the function*
- *• The C++ standard library provides numerous built-in functions that your program can call*
- *For example,*
 - *function **strcat** () concatenate two strings,*
 - *function **memcpy** () copy one memory location to another location.*

C++ Programming

Functions

Syntax:

type name (parameter1, parameter2, ...) { statements }

where,

- ***type** is the data type specifier of the data returned by the function and **name** is the identifier*
- ***parameters** (as many as needed): they allow to pass arguments to the function when it is called. The different parameters are separated by commas.*
- *Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: **int x**).*
- ***statements** is the function's body. It is a block of statements surrounded by **braces** { }.*

C++ Programming

Functions

Example:

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

int addition (int a, int b)           //function declaration
{
    int c;                           //function definition
    c=a+b;                           //function definition
    return (c);                      //function definition
}

int main ()
{
    int answer;
    answer = addition (5,3);          //call to function addition
    cout << "The result is " << answer;
    return 0;
}
```

C++ Programming

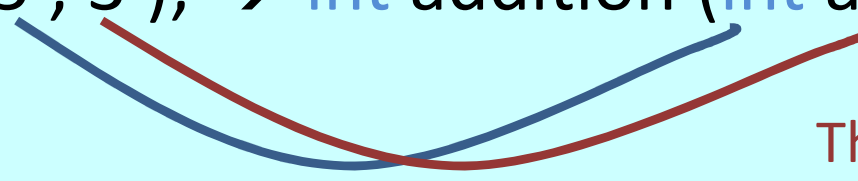
Functions

Example:

int addition (int a, int b)

Calling function

answer = addition (5 , 3); → int addition (int a, int b)



This calls function addition in main function

Return integer value of function “addition”

Argument passing by value

C++ Programming

Functions

Example:

int addition (int& a, int& b)

Calling function

answer = addition (5 , 3); → int addition (int& a, int& b)



This calls function addition in main function

Return integer value of function “addition”

Argument passing by reference

C++ Programming

Functions

Example:

```
#include <iostream>

int addition (int a, int b)           //function declaration
{
    int c;                           //function definition
    c=a+b;                           //function definition
    return (c);                      //function definition
}

int main ()
{
    int answer;
    answer = addition (5,3);          //call to function addition
    cout << "The result is " << answer;
    return 0;
}
```

```
#include <iostream>

int addition (int , int );            //function prototype
//int addition (int a, int b);        //function prototype

int main ()
{
    int answer;
    answer = addition (5,3);          //call to function addition
    cout << "The result is " << answer;
    return 0;
}

int addition (int a, int b)           //function declaration
{
    int c;                           //function definition
    c=a+b;                           //function definition
    return (c);                      //function definition
}
```

C++ Programming

Arrays

An array is a fixed number of elements of the same type stored sequentially in memory. The size of the array is referred to as its dimension.

Syntax:

type arrayName [dimension];

*where type is a valid data type (int, float, etc),
 arrayName is a valid identifier. and
 dimension (which is always enclosed in square brackets []) specifies the number of elements contained in the array.*

Example: **int coor [4];** → coor [0] = 1; coor [1] = 0; coor [2] = 3; coor [3] = 7;

int coor [4] = { 1, 0, 3, 7 };

int coor [2][4];

C++ Programming

Arrays

Example:

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

void printArray (int arg [ ], int length)
{
    for (int n=0; n < length; n++)
    {
        cout << arg [n] << endl;
    }
    cout << endl;
}

int main ()
{
    int firstArray [ ] = {1, 2, 3, 4};
    int secondArray [ ] = {2, 6, 12, 10, 0};
    printArray (firstArray, 4);
    printArray (secondArray, 5);
    return 0;
}
```

C++ Programming

Pointers

A pointer is a variable that holds the address of another variable. Like any variable, we must declare a pointer before we can use it. The general form of a pointer variable declaration is:

Syntax:

type *var-name;

where **type** is a pointers base type (int, float, etc),
 var-name is the name of the pointer variable and
 the **asterisk** is being used to designate a variable as a pointer.

Example:

int *ip;	// pointer to an integer double
double *dp;	// pointer to a double float
float* fp;	// pointer to a float char
char * ch	// pointer to character

C++ Programming

Pointers

*Since pointers only hold addresses, when we assign a value to a pointer, the value has to be an address. To get the address of a variable, we use the **ampersand** (&) operator which denotes an address in memory.*

Example: **int nValue = 5;**

int *pnPtr = &nValue;	// assign address of nValue to pnPtr
cout << &nValue << endl;	// print the address of variable nValue
cout << pnPtr << endl;	// print the address that pnPtr is holding

When the above code is compiled and executed, its output is of type

0012FF7C

0012FF7C

C++ Programming

Pointers

Example: **int nValue = 5;**
 cout << &nValue; *// prints address of nValue*
 cout << nValue; *// prints contents of nValue*
 int *pnPtr = &nValue; *// pnPtr points to nValue*
 cout << pnPtr; *// prints address held in pnPtr, which is &nValue*
 cout << *pnPtr; *// prints contents pointed to by pnPtr, which is*
 contents of nValue

When the above code is compiled and executed, its output is of type

0012FF7C

5

0012FF7C

5

Finally, we can also:

- **Pass pointers to functions.**
- **Return pointers from functions.**
- **Use pointers to pointers.**
- **Use an array of pointer.**
- **Perform arithmetic operations on a pointer (++ , -- , + , and -) and**
- **pointer comparisons (== , < , and >).**

C++ Programming

typedef

C++ allows the definition of **our own types** based on other existing data types. We can do this using the keyword **typedef**, whose format is:

Format:

```
typedef existing_type new_type_name ;
```

where **existing_type** is a C++ fundamental or compound type and **new_type_name** is the name for the new type.

Typedef `vector <double> doubleVector;`

Here,

`vector<double> a(8);` is equivalent to: `doubleVector a(8);`

- **typedef** does not create different types. It only creates synonyms of existing types.
- typedefs make the code easy to read. **OpenFOAM** uses them a lot!

C++ Programming

Classes

- *A class is an user defined data type*
- *A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package.*
- *The data and functions within a class are called members of the class. Classes are generally declared using the keyword class.*

```
Format:      class class_name {
                                access_specifier_1: member1;
                                access_specifier_2: member2;
                                } object_names;
```

where

class_name is a valid identifier for the class,
object_names is an optional list of names for objects of this class.

The body of the declaration can contain members, that can be either data or function declarations, and optionally **access specifiers**.

C++ Programming

Classes

- *An access specifier is one of the following three keywords: private, public or protected.*
- *These specifiers gives the access rights that the members following them acquire:*
 - *private members of a class are accessible only from within other members of the same class or from their friends.*
 - *protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.*
 - *public members are accessible from anywhere where the object is visible.*

By default, all members of a class declared with the class keyword have private access for all its members.

C++ Programming

Classes

For example:

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

Declares a class (i.e., a data type) called CRectangle and

rect : *a object (i.e., a variable) of this class called.*

members : *four members*

*two data members of type int (member x and member y) with private access, and
two member functions with public access: set_value () and area ()*

Classes

CRectangle class and rect object:

we can access any public member of the object rect as if they were normal functions or normal variables.

*This is done by using the **dot** (.) operator in combination with the name of the object and the name of the member.*

For example:

```
rect.set_values (3,4);  
myarea = rect.area();
```

C++ Programming

Classes

scope operator (::),

It is used to define a member of a class from outside the class definition itself.

- *You may notice that the definition of the member function area () has been included directly within the definition of the class.*
- *The only difference between defining a class member function within its class or to include only the prototype and later its definition, is that in the first case the function is considered an inline member function, while in the second case it is a normal member function (not-inline).*

This supposes no difference in behavior.

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

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

//Definition of the member function
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 ( );
    return 0;
}
```

C++ Programming

Five basic Classes in openFOAM

- Space and time: **polyMesh**, **fvMesh**, **Time**
- Field algebra: **Field**, **DimensionedField** and **GeometricField**
- Boundary conditions: **fvPatchField** and **derived** classes
- Sparse matrices: **lduMatrix**, **fvMatrix** and **linear** solvers
- Finite Volume discretization: **fvc** and **fvm** namespace

C++ is a complex programming language, rich in features.

OpenFOAM use all C++ features.

OpenFOAM is an excellent piece of C++ and software engineering.

Decent piece of CFD code.

~ H. Jasak

C++ Programming in OpenFOAM®

Basic tensor classes in OpenFOAM

- *OpenFOAM contains a C++ class library named primitive.*
- *All the classes for the tensor mathematics are available here.*

Rank	Common name	Basic class	Access function
0	Scalar	Scalar	
1	Vector	Vector	x(), y(),z()
2	Tensor	tensor	xx(), xy(). xz() ...

$$\mathbf{T} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$



can be declared in the following way

```
tensor T(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

We can then access the component or using the xz () access function. For instance, the code

```
Info << "Txz = " << T.xz ( ) << endl;
```

Will give the following output → **Txz = 3**

Dimensional units in OpenFOAM

- Units are defined using the *dimensionSet* **class** *tensor*

No.	Property	Unit	Symbol
1	Mass	Kilogram	k
2	Length	meters	m
3	Time	second	s
4	Temperature	Kelvin	K
5	Quantity	moles	mol
6	Current	ampere	A
7	Luminuous intensity	candela	cd

dimensionSet (k, m, s, K, mol, A, cd)

```
dimensionedTensor sigma
(
    "sigma",
    dimensionSet(1, -1, -2, 0, 0, 0, 0),
    tensor(1e6,0,0,0,1e6,0,0,0,1e6)
);
```



$$\sigma = \begin{pmatrix} 10^6 & 0 & 0 \\ 0 & 10^6 & 0 \\ 0 & 0 & 10^6 \end{pmatrix}$$

C++ Programming in OpenFOAM®

Equation discretization in OpenFOAM

Converts the PDEs into a set of linear algebraic equations, $\mathbf{Ax}=\mathbf{b}$, where

*\mathbf{x} and \mathbf{b} are **volFields** (geometricFields),*

*\mathbf{A} is a **fvMatrix**, which is created by a discretization of a geometricField
and inherits the algebra of its corresponding field*

The fvm (Finite Volume Method) and fvc (Finite Volume Calculus) classes contain static functions for the differential operators, and discretize any geometricField.

fvm returns a fvMatrix, and fvc returns a geometricField

(see \$FOAM_SRC/finiteVolume/finiteVolume/fvc and \$FOAM_SRC/finiteVolume/finiteVolume/fvm)

C++ Programming in OpenFOAM®

Discretization of the basic PDE terms in OpenFOAM

Term Description	Mathematical Expression	fvm::fvc
Laplacian	$\nabla^2 \phi$, $\nabla \cdot \Gamma \nabla \phi$	laplacian (phi), laplacian (Gamma, phi)
Time Derivative	$\frac{\partial \phi}{\partial t}$, $\frac{\partial \rho \phi}{\partial t}$	ddt (phi), ddt (rho,phi)
Convection	$\nabla \cdot (\psi)$, $\nabla \cdot (\psi \phi)$	div (psi,scheme), div (psi,phi)
Source	$\rho \phi$	Sp (rho,phi), SuSp (rho,phi)
	ϕ vol<type>Field ρ scalar, volScalarField ψ surfaceScalarField	

C++ Programming in OpenFOAM®

Solution of the PDE in OpenFOAM: convection-diffusion equation

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

In openFOAM we write this as

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

Structure of OpenFOAM

The OpenFOAM code is structured as follows:

- **applications:** source files of all the executables:
 - solvers
 - Utilities
 - Test
- **bin:** basic executable scripts.
- **doc:** pdf and Doxygen documentation.
 - Doxygen
 - Guides-a4
- **lib:** compiled libraries.
- **src:** source library files.
- **tutorials:** tutorial cases.
- **wmake:** compiler settings.

C++ types, classes and objects

- The variables we assign to a type are objects of that class.
- Object orientation focuses on the objects instead of the functions.
- An object belongs to a class of objects with the same attributes. The class defines:
 - The construction of the object
 - The destruction of the object
 - Attributes of the object (member data)
 - Functions that can manipulate the object (member functions)
- The objects may be related in different ways, and the classes may inherit attributes from other classes.
- A benefit of object orientation is that the classes can be re-used, and that each class can be designed and bug-fixed for a specific task.
- In OpenFOAM, the classes are designed to define, discretize and solve PDE's.

class Definition

- The following structure defines the class with name myClass and its public and private member functions and member data.

```
class myClass
{
    public:
        declarations of public member functions and member data
    private:
        declaration of hidden member functions and member data
};
```

- An object of a class myClass is defined in the main code as:

```
myClass myObject; (c.f. int i)
```

- The object myObject will then have all the attributes defined in the class myClass.
- Any number of objects may belong to a class, and the attributes of each object will be separated.
- The member functions operate on the object according to its implementation. If there is a member function write that writes out the contents of an object of the class myClass, it is called in the main code as:

```
myObject.write();
```


member Functions

- The member functions may be defined either in the declaration of the class, or in the definition of the class. The syntax is basically:

```
inline void myClass::write()
{
    Contents of the member function.
}
```

- where

- **myClass::** tells us that the member function write belongs to the class myClass.
- **void** tells us that the function does not return anything
- **inline** tells us that the function will be inlined into the code where it is called instead of jumping to the memory location of the function at each call (good for small functions). Member functions defined directly in the class declaration will automatically be inlined if possible.

- The member functions have direct access to all the member data and member functions of the class.

association of classes

- A good programming standard is to make the class files in pairs, one with the class declarations (**.H**), and one with the class definitions (**.C**).
- The class declaration file must be included in the files where the class is used
- The compiled definition file is statically or dynamically linked to the executable by the compiler.
- Inline functions must be implemented in the class declaration file, since they must be inlined without looking at the class definition file.
 - Example: In OpenFOAM there are usually files named as **VectorI.H** containing inline functions, and those files are included in the corresponding **Vector.H** file.

Constructors of Class

- A constructor is a special initialization function that is called each time a new object of that class is constructed. Without a specific constructor all attributes will be undefined.
- A null constructor must always be defined.
- A constructor can be used to initialize the attributes of the object.
- A constructor is recognized by it having the same name as the class - here Vector. (Cmpt is a template generic parameter (component type), i.e. the Vector class works for all component types). **See Vector.**
- The actual initialization usually takes place in the corresponding **.C** file, but since the constructors for the Vector are inlined, it takes place in the **VectorI.H** file
- A copy constructor has a parameter that is a reference to another object of the same class: **myClass(const myClass&);**
- The copy constructor copies all attributes. A copy constructor can only be used when initializing an object.
- A type conversion constructor is a constructor that takes a single parameter of a different class than the current class, and it describes explicitly how to convert between the two classes.

Destructors of Class

- When using dynamically allocated memory it is important to be able to destruct an object.
- A destructor is a member function without parameters, with the same name as the class, but with a ~ in front of it.
- An object should be destructed when leaving the block it was constructed in, or if it was allocated with new it should be deleted with delete
- To make sure that all the memory is returned it is preferable to define the destructor explicitly.

static members of Class

- Static members of a class only exist in a single instance in a class, for all objects
- They are defined as static, which can be applied to member data or member functions.
- Static members do not belong to any particular object, but to a particular class, so they are used as:

`myClass::staticFunction(parameters);`

- See in **Vector.H**

Inheritance of Class

- A class can inherit attributes from already existing classes, and extend with new attributes.

- Syntax, when defining the new class:

```
class newClass : public oldClass { ...members... }
```

- where newClass will inherit all the attributes from oldClass.
- newClass is now a sub-class to oldClass.

- OpenFOAM example:

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

- where class Vector is a sub-class to VectorSpace.

- An attribute of newClass may have the same name as one in oldClass. Then the newClass attribute will be used for newClass objects and the oldClass attribute will be hidden. Note that for member functions, all of them with the same name will be hidden, irrespectively of the number of parameters.

Inheritance of Class

- A hidden member of a base-class can be reached by `oldClass::member`
- Members of a class can be public, private or protected.
- private members are never visible in a sub-class, while public and protected are.
- However, protected are only visible in a sub-class (not in other classes).
- The visibility of the inherited members can be modified in the new class using the reserved words public, private or protected when defining the class. (public in the previous example). It is only possible to make the members of a base-class less visible in the sub-class.
- A class may be a sub-class to several base-classes (multiple inheritance), and this is used to combine features from several classes.

virtual member functions

- Virtual member functions are used for dynamic binding, i.e. the function will work differently depending on how it is called, and it is determined at run-time.
- The reserved word `virtual` is used in front of the member function declaration to declare it as virtual.
- A sub-class to a class with a virtual function should have a member function with the same name and parameters, and return the same type as the virtual function.
 - That sub-class member function will automatically be a virtual function.
- OpenFOAM uses this to dynamically choose turbulence model.
 - Virtual functions make it easy to add new turbulence models without changing the original classes.

abstract classes

- A class with at least one virtual member function that is undefined (a pure virtual function) is an abstract class.
- The purpose of an abstract class is to define how the sub-classes should be defined.
- An object can not be created for an abstract class.
- See RANS Model
- The most important function is the correct() function, which is called as a pointer in the application as:

`turbulence->correct();`

templates

- The most obvious way to define a class is to define it for a specific type of object.
 - However, often similar operations are needed regardless of the object type.
 - Instead of writing a number of identical classes where only the object type differs, a generic template can be defined.
 - The compiler then defines all the specific classes that are needed.
- A template class is defined by a line in front of the class definition, as follows:
 - `template<class T>`
 - where T is the generic parameter .
 - The word class defines T as a type parameter.
- The generic parameter(s) are then used in the class definition instead of the specific type name(s).
- A template class is used to construct an object as:

`templateClass<type> templateClassObject;`

typedef

- OpenFOAM is full of templates.
- To make the code easier to read OpenFOAM re-defines the templated class names, for instance:

```
typedef List<vector> vectorList;
```

- A list of vectors can then simply be constructed as

```
integerVector iV;
```

- This is used to a large extent in OpenFOAM, and the reason for this is to make the code easier to read.

namespace

- A namespace with the name myNamespace is defined as

```
namespace myNamespace {  
    declarations  
}
```

- You can see

```
namespace Foam { }  
  
all over OpenFOAM
```

Acknowledgements

These slides are mainly based upon, OpenFOAM® user guide, OpenFOAM® programmer's guide, and presentations from previous OpenFOAM® training sessions , OpenFOAM® workshops and personal experience.

We gratefully acknowledge the following OpenFOAM® users for their training material:

Hrvoje Jasak, Wikki Ltd.

Hakan Nilsson, Department of Applied Mechanics, Chalmers University of Technology.

Eric Paterson, Applied Research Lab., Professor, Mechanical Engineering, Pennsylvania State University.

Tommaso Lucchini, Department of Energy, Politecnico di Milano.

Gianluca Montenegro, Dept. of Energy, Politecnico di Milano, OpenFOAM Workshop Training 2008

Joel Guerrero, University of Genoa, DICAT

Thank you.....