

Basics of C++ in OpenFOAM

Basics of C++ in OpenFOAM

- To begin with: The aim of this part of the course is not to teach all of C++, but to give a short introduction that is useful when trying to understand the contents of OpenFOAM.
- After this introduction you should be able to *recognize* and make *minor modifications* to most C++ features in OpenFOAM.
- Some books:
 - *C++ direkt* by Jan Skansholm (ISBN 91-44-01463-5)
 - *C++ from the Beginning* by Jan Skansholm (probably similar)
 - *C++ how to Program* by Paul and Harvey Deitel
 - *Object Oriented Programming in C++* by Robert Lafore

C++ basics – types

- Variables can contain data of different *types*, for instance:

```
int myInteger;
```

for a declaration of an integer variable named `myInteger`, or

```
const int myConstantInteger = 10;
```

for a declaration of an *constant* integer variable named `myConstantInteger` with value 10.

- Variables can be added, subtracted, multiplied and divided as long as they have the same type, or if the types have definitions on how to convert between the types.
- In C++ it is possible to define special *types* (classes), and there are many types defined for you in OpenFOAM.
- User-defined types must have the required conversions defined. Some of the types in OpenFOAM can be used together in arithmetic expressions, but not all of them.

C++ basics – Namespace

- When using pieces of C++ code developed by different programmers there is a risk that the same name has been used for different things.
- By associating a declaration with a namespace, the declaration will only be visible if that namespace is used. The standard declarations are used by starting with:

```
using namespace std;
```

- OpenFOAM declarations belong to namespace Foam, so in OpenFOAM we use:

```
using namespace Foam;
```

to make all declarations in namespace Foam visible.

- Explicit naming in OpenFOAM:

```
Foam::function();
```

where `function()` is a function defined in namespace Foam. This must be used if any other namespace containing a declaration of another `function()` is also visible.

C++ basics – input/output

- Input and output can be done using the standard library `iostream`, using:

```
cout << "Please type an integer!" << endl;  
cin >> myInteger;
```

where `<<` and `>>` are output and input operators, and `endl` is a manipulator that generates a new line (there are many other manipulators).

- In OpenFOAM a new output stream `Info` is however defined, and it is recommended to use that one instead since it takes care of write-outs for parallel simulations.

C++ basics, main function

- All C++ codes must have at least one function:

```
int main()  
{  
    return 0;  
}
```

in this case, `main` takes no arguments, but it may (as in OpenFOAM applications).

- The main function should always return an integer, and default is 0, so for the main function it is allowed to write only:

```
main()  
{  
}
```

- Code appearing after the `return` statement is not executed!!!

C++ basics, Example code

In file `basic1.C`:

```
#include <iostream>
using namespace std;
main()
{
    int myInteger;
    const int constantInteger=5;
    const float constantFloat=5.1;
    cout << "Please type an integer!" << endl;
    cin >> myInteger;
    cout << myInteger << " + " << constantInteger << " = "
         << myInteger+constantInteger << endl;
    cout << myInteger << " + " << constantFloat << " = "
         << myInteger+constantFloat << endl;
}
```

Compile and run with:

```
g++ basic1.C -o basic1;
./basic1
```

C++ basics – operators

- `+`, `-`, `*` and `/` are operators that define how the operands should be used.
- Other standard operators are:

<code>%</code>	(integer division modulus)
<code>++</code>	(add 1)
<code>--</code>	(subtract 1)
<code>+=</code>	(<code>i+=2</code> adds 2 to <code>i</code>)
<code>-=</code>	(<code>i-=2</code> subtracts 2 from <code>i</code>)
<code>*=</code>	(<code>i*=2</code> multiplies <code>i</code> by 2)
<code>/=</code>	(<code>i/=2</code> divides <code>i</code> by 2)

etc. User-defined types should define its operators.

- Comparing operators: `<` `>` `<=` `>=` `==` `!=` Generates `bool` (boolean)
- Logical operators: `&&` `||` `!` (or, for some compilers: `and` `or` `not`). Generates `bool` (boolean)

C++ basics – functions

- Mathematic standard functions are available in standard libraries. They are thus not part of C++ itself.
- Standard library `cmath` contains trigonometric functions, logarithmic functions and square root. (use `#include <cmath>`; if you need them)
- Standard library `cstdlib` contains general functions, and some of them can be used for arithmetics. (use `#include <cstdlib>`; if you need them)

C++ basics – if, for and while-statements

- if-statements:

```
if (variable1 > variable2) {...CODE...} else {...CODE...}
```

- for-statements:

```
for ( init; condition; change ) {...CODE...}
```

- while-statements:

```
while (...expression...) {...CODE...}
```

`break;` **breaks the execution of while**

C++ basics, Example code

In file `basic2.C`:

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

main()
{
    float myFloat;
    cout << "Please type a float!" << endl;
    cin >> myFloat;
    cout << "sin(" << myFloat << ") = " << sin(myFloat) << endl;
    if (myFloat < 5.5){cout << myFloat << " is less than 5.5" << endl;} else
        {cout << myFloat << " is not less than 5.5" << endl;};
    for ( int i=0; i<3; i++ ) {cout << "For-looping: " << i << endl;}
    int j=0;
    while (j<3) {cout << "While-looping: " << j << endl; j++;}
}
```

Compile and run with:

```
g++ basic2.C -o basic2; ./basic2
```

C++ basics – arrays

- **Arrays:**

`double f[5];` (Note: components numbered from 0!)

`f[3] = 2.75;` (Note: no index control!)

`int a[6] = {2, 2, 2, 5, 5, 0};` (declaration and initialization)

The arrays have strong limitations, but serve as a base for array **templates**

- **Array templates** (example `vector`. other: `list`, `deque`):

```
#include <vector>
```

```
using namespace std
```

The type of the vector must be specified upon declaration:

```
vector<double> v2(3); gives {0, 0, 0}
```

```
vector<double> v3(4, 1.5); gives {1.5, 1.5, 1.5, 1.5}
```

```
vector<double> v4(v3); Constructs v4 as a copy of v3 (copy-constructor)
```

- **Array template operations:** The template classes define member functions that can be used for those types, for instance: `size()`, `empty()`, `assign()`, `push_back()`, `pop_back()`, `front()`, `clear()`, `capacity()` etc.
`v.assign(4, 1.0); gives {1.0, 1.0, 1.0, 1.0}`

C++ basics, Example code

In file basic3.C:

```
#include <iostream>
#include <vector>
using namespace std;
main()
{
vector<double> v2(3);
vector<double> v3(4, 1.5);
vector<double> v4(v3);
cout << "v2: (" << v2[0] << ", " << v2[1] << ", " << v2[2] << ")" << endl;
cout << "v3: (" << v3[0] << ", " << v3[1] << ", " << v3[2] << ", " << v3[3] << ")" << endl;
cout << "v4: (" << v4[0] << ", " << v4[1] << ", " << v4[2] << ", " << v4[3] << ")" << endl;
cout << "v2.size(): " << v2.size() << endl;
}
```

Compile and run with:

```
g++ basic3.C -o basic3; ./basic3
```

Note that the standard `vector` class is **not** implemented to be able to execute:

```
cout << "v2: " << v2 << endl;
```

Such functionality is available in OpenFOAM.

C++ basics – function implementation

- Example function named `average`

```
double average (double x1, double x2)
{
    int nvalues = 2;
    return (x1+x2)/nvalues;
}
```

takes two arguments of type `double`, and returns type `double`. The variable `nvalues` is a local variable, and is only visible inside the function. Note that any code after the `return` statement will not be executed.

- A function doesn't have to take arguments, and it doesn't have to return anything (the output type is then specified as `void`).
- There may be several functions with the same names, as long as there is a difference in the arguments to the functions - the number of arguments or the types of the arguments.
- Functions must be *declared* before they are used.

C++ basics, Example code

In file `basic4.C`:

```
#include <iostream>
using namespace std;
double average (double x1, double x2)
{
    int nvalues = 2;
    return (x1+x2)/nvalues;
}
main()
{
    double d1=2.1;
    double d2=3.7;
    cout << "Average: " << average(d1,d2) << endl;
}
```

Compile and run with:

```
g++ basic4.C -o basic4; ./basic4
```

C++ basics – declaration and definition of functions

- The function *declaration* must be done before it is used, but the function *definition* can be done after it is used. Example:

```
double average (double x1, double x2); //Declaration
main ()
{
    mv = average(value1, value2)
}
double average (double x1, double x2) //Definition
{
    return (x1+x2)/2;
}
```

The argument *names* may be omitted in the declaration.

- Declarations are often included from include-files:

```
#include "file.h"
#include <standardfile>
```

- A good way to program C++ is to make files in pairs, one with the declaration, and one with the definition. This is done throughout OpenFOAM.

C++ basics, Example code

In file `basic5.C`:

```
#include <iostream>
#include "basic5.H"
using namespace std;
main()
{
    double d1=2.1;
    double d2=3.7;
    cout << "Average: " << average(d1,d2) << endl;
}
double average (double x1, double x2)
{
    int nvalues = 2;
    return (x1+x2)/nvalues;
}
```

In file `basic5.H`:

```
double average (double, double);
```

Compile and run with: `g++ basic5.C -o basic5; ./basic5`

C++ basics – function parameters / arguments reference and default value

- If an argument variable should be changed inside a function, the type of the argument must be a reference, i.e.

```
void change(double& x1)
```

The reference parameter `x1` will now be a reference to the argument to the function instead of a local variable in the function. (standard arrays are always treated as reference parameters).

- Reference parameters can also be used to avoid copying of large fields when calling a function. To avoid changing the parameter in the function it can be declared as `const`, i.e.

```
void checkWord(const string& s)
```

This often applies for parameters of class-type, which can be large.

- Default values can be specified, and then the function may be called without that parameter, i.e.

```
void checkWord(const string& s, int nmbr=1)
```

C++ basics, Example code

In file basic6.C:

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

double average (double& x1, double& x2, int nvalues=2)
{
    x1 = 7.5;
    return (x1+x2)/nvalues;
}

main()
{
    double d1=2.1;
    double d2=3.7;
    cout << "Modified average: " << average(d1,d2) << endl;
    cout << "Half modified average: " << average(d1,d2,4) << endl;
    cout << "d1: " << d1 << ", d2: " << d2 << endl;
}
```

Compile and run with: `g++ basic6.C -o basic6; ./basic6`

C++ basics – Pointers

- Pointers point at a memory location (while a reference is referring to another variable, as shown before, i.e. they are different).

- A pointer is recognized by its definition (*):

```
int *pint;  
double *pdouble;  
char *pchar;
```

- Turbulence models are treated with the turbulence pointer in OpenFOAM.

In file: \$FOAM_SOLVERS/incompressible/simpleFoam/createFields.H:

```
autoPtr<incompressible::RASModel> turbulence  
(  
    incompressible::RASModel::New(U, phi, laminarTransport)  
);
```

In file \$FOAM_SOLVERS/incompressible/simpleFoam/simpleFoam.C:

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

- We will not discuss pointers any further at the moment.

C++ basics – Types

- *Types* define what values a variable may obtain, and what operations may be made on the variable.
- Pre-defined C++ types are:

signed char	unsigned int
short int	unsigned long int
int	float
unsigned char	double
unsigned short int	long double

- User defined types can be defined in *classes*. OpenFOAM provides many types/classes that are useful for solving partial differential equations.
- OpenFOAM classes are used by including the class declarations in the header of the code, and linking to the corresponding compiled OpenFOAM library at compilation.
- The path to included files that are in another path than the current directory must be specified by `-I`
- The path to libraries that are linked to is specified with `-L`

C++ basics, Example code

In file `basic7.C`:

```
#include <iostream>           //Just for cout
using namespace std;         //Just for cout
#include "tensor.H"           //From OpenFOAM
#include "symmTensor.H"       //From OpenFOAM
using namespace Foam;        //From OpenFOAM
int main()
{
    tensor t1(1, 2, 3, 4, 5, 6, 7, 8, 9); //From OpenFOAM
    cout << "t1[0]: " << t1[0] << endl;
    symmTensor st1(1, 2, 3, 4, 5, 6);      //From OpenFOAM
    cout << "st1[5]: " << st1[5] << endl;
    return 0;}
```

Compile and run with (some trial-and-error, looking at output from `wmake` for `test/tensor`):

```
g++ basic7.C -DWM_DP -I$FOAM_SRC/OpenFOAM/lnInclude \
    -L$WM_PROJECT_DIR/lib/$WM_OPTIONS/libOpenFOAM.so -o basic7; ./basic7
```

Here, `-DWM_DP` is a declaration for the compiler to use double precision.

We include header files (declarations) from `$FOAM_SRC/OpenFOAM/lnInclude`

We link to library (definitions) `$WM_PROJECT_DIR/lib/$WM_OPTIONS/libOpenFOAM.so`