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."

Introductory OpenFOAM® Course

From 8th to 12th July, 2013

University of Genoa, DICCA

Dipartimento di **I**ngegneria **C**ivile, **C**himica e **A**mbientale

UNIVERSITÀ DEGLI STUDI
DI GENOVA

Joel GUERRERO

joel.guerrero@unige.it

guerrero@wolfdynamics.com

Acknowledgements

These slides and the tutorials presented are based upon personal experience, OpenFOAM® source code, OpenFOAM® user guide, OpenFOAM® programmer's guide, and presentations from previous OpenFOAM® training sessions and OpenFOAM® workshops.

We gratefully acknowledge the following OpenFOAM® users for their consent to use their material:

- Hrvoje Jasak. Wikki Ltd.

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

- Eric Paterson. Applied Research Laboratory Professor of Mechanical Engineering, Pennsylvania State University.

Today's lecture

1. C++: A Crash Introduction

Before continuing,
I want to remind you that this is not
a C++ course

C++: A Crash introduction

My favorite C++ bibliographical references:

- **C++ for Engineers and Scientists.**
  G. J. Bronson. 2012, Cengage Learning.

- **Sams Teach Yourself C++ in One Hour a Day.**
  S. Rao. 2012, Sams Publishing.

- **Sams Teach Yourself C++ in One Hour a Day.**
  J. Liberty, B. Jones. 2004, Sams Publishing.

- **The C++ Standard Library**.
  N. Josuttis. 2012, Addison-Wesley.

- http://www.cplusplus.com/

C++: A Crash introduction

C++ Program Structure

Let us study the basic structure of a C++ program by looking at a simple code that prints the words **Hello world.**

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


// main() is where program execution begins.  It is the main function.
// Every program in c++ must have this main function declared

int main ()
{
    cout << "Hello world";      //prints Hello world
    return 0;                   //returns nothing
}
```

C++: A Crash introduction

C++ Data Types

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

C++ Data Types Modifiers

C++ allows the char, int, and double data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it better fits the actual needs. The data type modifiers are listed here:

- signed

- unsigned

- long

- short

The modifiers signed, unsigned, long, and short can be applied to integer base types. In addition, signed and 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++: A Crash introduction

Variable Declaration & Initialization

- You declare the variables by using the C++ data types.

- You initialize a variable by assigning it a value.

- To initialize a variable you need to first declare it.

- All variables that we plan to use must have been declared (and initialized) before we use them.

- Take a look a the following examples:

```
int a, b, c;        //Declaration
int aa = 5.12;      //Declaration and initialization, it will only read the integer part
int b (2);          //Declaration and initialization, this is equivalent to b =2
float pi;           //Declaration
pi = 3.14159;       //Initialization
```

C++: A Crash introduction

Scope of Variables

- All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code.

- A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.

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

int a = 4;                              //global scope variable

// main() is where program execution begins.
int main ()
{
        int b = 4;                      //local scope variable
        int result;

        result = a*b;
        cout << result << endl;         //prints result
        return 0;
}
```

Constants Variables

- Constants are expressions with a fixed value.

- Constants variable can be declared by using the #define preprocessor directive or by using the const prefix with a specific data type. For example,

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

#define PI 3.14159                      //global scope constant
#define NEWLINE '\n'                    //global scope constant

int main ()
{
        const float pi = 3.14159;        //local scope constant
        float r = 1.0;
        double area_circle, area_circle2;
        area_circle        = 2*PI*r;
        area_circle2       = 2*pi*r;
        cout << area_circle  << endl;    //prints result
        cout << NEWLINE;                 //insert new line
        cout << area_circle2 << endl;    //prints result
        return 0;

}
```

C++ Operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides the following type of operators:

  - Assignment Operator: **=**
  - Arithmetic Operators: **+, -, \, *, %**
  - Increase/decrease: **++, --**
  - Relational Operators: **==, !=, >, <, >=, <=**
  - Logical Operators: **&&, ||, !**
  - Bitwise Operators: **&, |, ^, ~, <<, >>**
  - Assignment Operators: **+=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=**
  - Miscellaneous Operators: **?**, comma (**,**), arrow (**->**), **sizeof()**, and so on.

- **And by the way, the list is not complete.**

C++: A Crash introduction

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. Its form is:

if (condition) statement

For example:

if (x == 7)
cout << "x is 7";

will print the "x is 7" only if the value stored in the x variable is indeed 7.

C++: A Crash introduction

C++ Control Structures

- **Conditional structure: if and else**

If we want more than a single statement to be executed in case that the condition is true we can specify a block using **braces { }**:

```
if (condition) statement
{

}
```

For example:

```
if (x == 7)
{
    cout << "x is ";
    cout << x;
}
```

C++ Control Structures

- **Conditional structure: if and else**

We can additionally specify what we want to do if the condition is not fulfilled by using the keyword else. Its must be used in conjunction with if, as follows:

```
if (condition)
    statement1
else
    statement2
```

For example:

```
if (x == 7)
    cout << "x is 7";
else
    cout << "x is not 7";
```

prints on the screen "x is 7" if indeed x has a value of 7, but if it does not, it prints out "x is not 7".

C++: A Crash introduction

C++ Control Structures

* **Conditional structure: if and else**

The if - else structures can be concatenated with the intention of verifying a range of values. The following example shows us its use:

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

* Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in **braces { }**.

* The conditional structures can be nested.

* It is always a good programming practice to use a single and clear indentation style.

C++: A Crash introduction

C++ Control Structures

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

The while loop

The format is:

> **while (expression) statement**

and it simply repeats the statement while the condition set in expression is true. For example:

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

C++ Control Structures

The do-while loop

The format is:

> **do** statement **while** (condition)

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. For example:

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

C++: A Crash introduction

C++ Control Structures

The for loop

The format is:

> **for (initialization; condition; increase) statement;**

It repeat the statement while the condition remains true, like the while loop. The for loop is designed to allow a counter variable that is initialized at the beginning of the loop and incremented (or decremented) on each iteration of the loop. For example:

```cpp
for (int = 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++: A Crash introduction

C++ Control Structures

In the for loop, the initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write:

```
for (;n<10;)
{
    // whatever here...
}
```

if we want to specify no initialization and no increase. We also could write

```
for (;n<10;n++)
{
    // whatever here...
}
```

if we want to include an increase field but no initialization (maybe because the variable was already initialized somewhere else).

C++: A Crash introduction

C++ Control Structures

Optionally, using the comma operator (**,**) we can specify more than one expression in any of the fields included in a for loop.

The comma operator (**,**) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we want to initialize more than one variable in our loop:

```cpp
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
    // whatever here...
}
```

This loop will execute for 50 times if neither n or i are modified within the loop. n starts with a value of 0, and i with a value of 100, the condition is n! = i (that is to say, n is not equal to i). Because n is increased by one and i decreased by one, the loop's condition will become false after the 50th loop, when both n and i will be equal to 50.

C++ Control Structures

- Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in **braces { }**.

- You will also need to declare the initialization and increase (or decrease) variables.

- The loops structures can be nested.

- It is always a good programming practice to use a single and clear indentation style.

C++: A Crash introduction

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. How you divide your code among different functions is up to you.

- 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.

- A function is also knows as a method, sub-routine, procedure, etc.

Functions

To use a function, we use the following format:

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

where:

- **type** is the data type specifier of the data returned by the function.

- **name** is the identifier by which it will be possible to call the function.

- **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++: A Crash introduction

Functions

For example:

```cpp
#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;
}
```
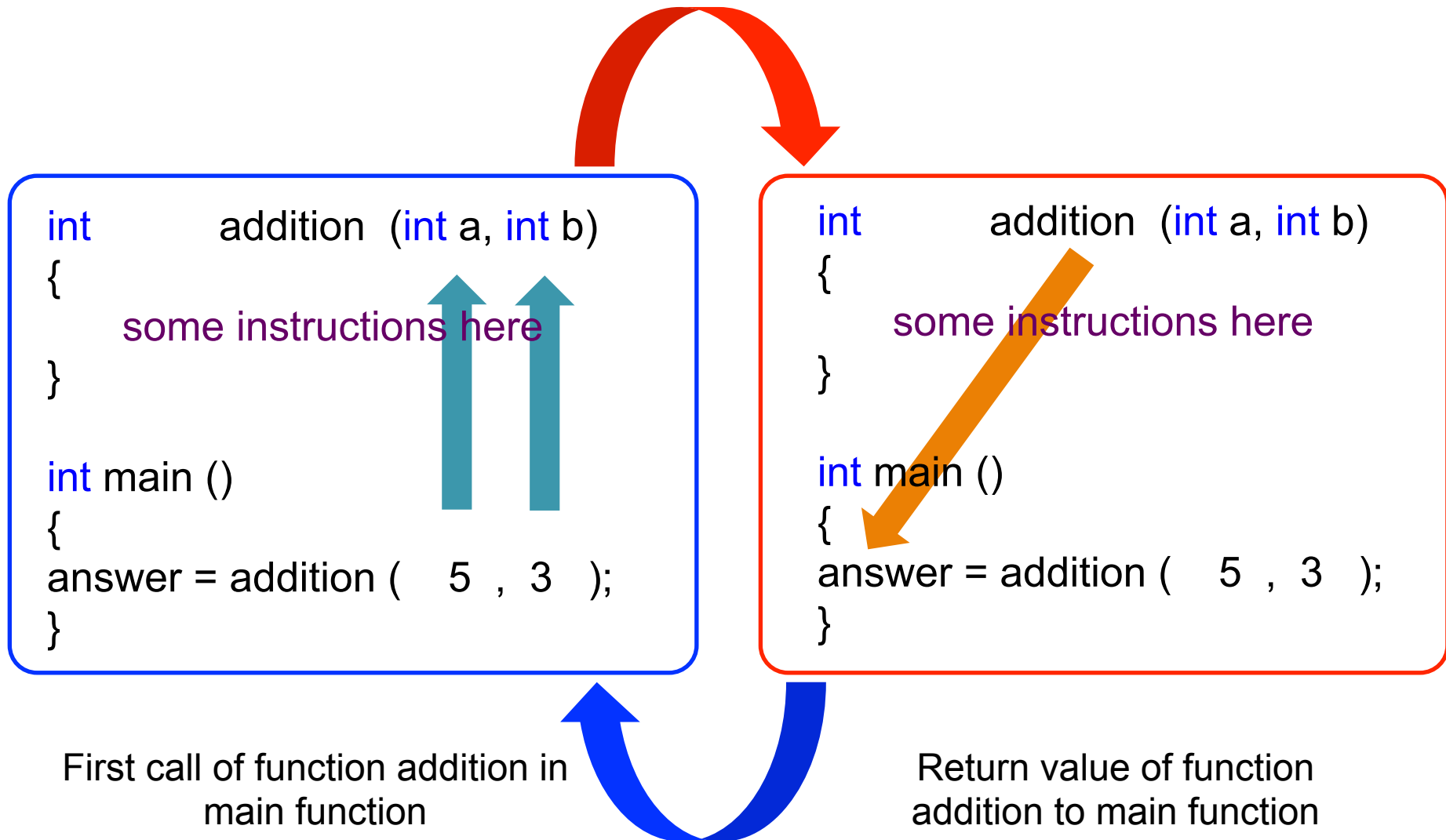
C++: A Crash introduction

Functions

- We can pass the variables or arguments to a function by value or by reference.

- When calling a function and passing the variables by value, what we pass to the function are copies of their values but never the variables themselves.

- When passing the variables by value, any modification to the passed variables within the function will not have any effect in the values of the variables outside it.

- When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

- Passing by reference is also an effective way to allow a function to return more than one value.
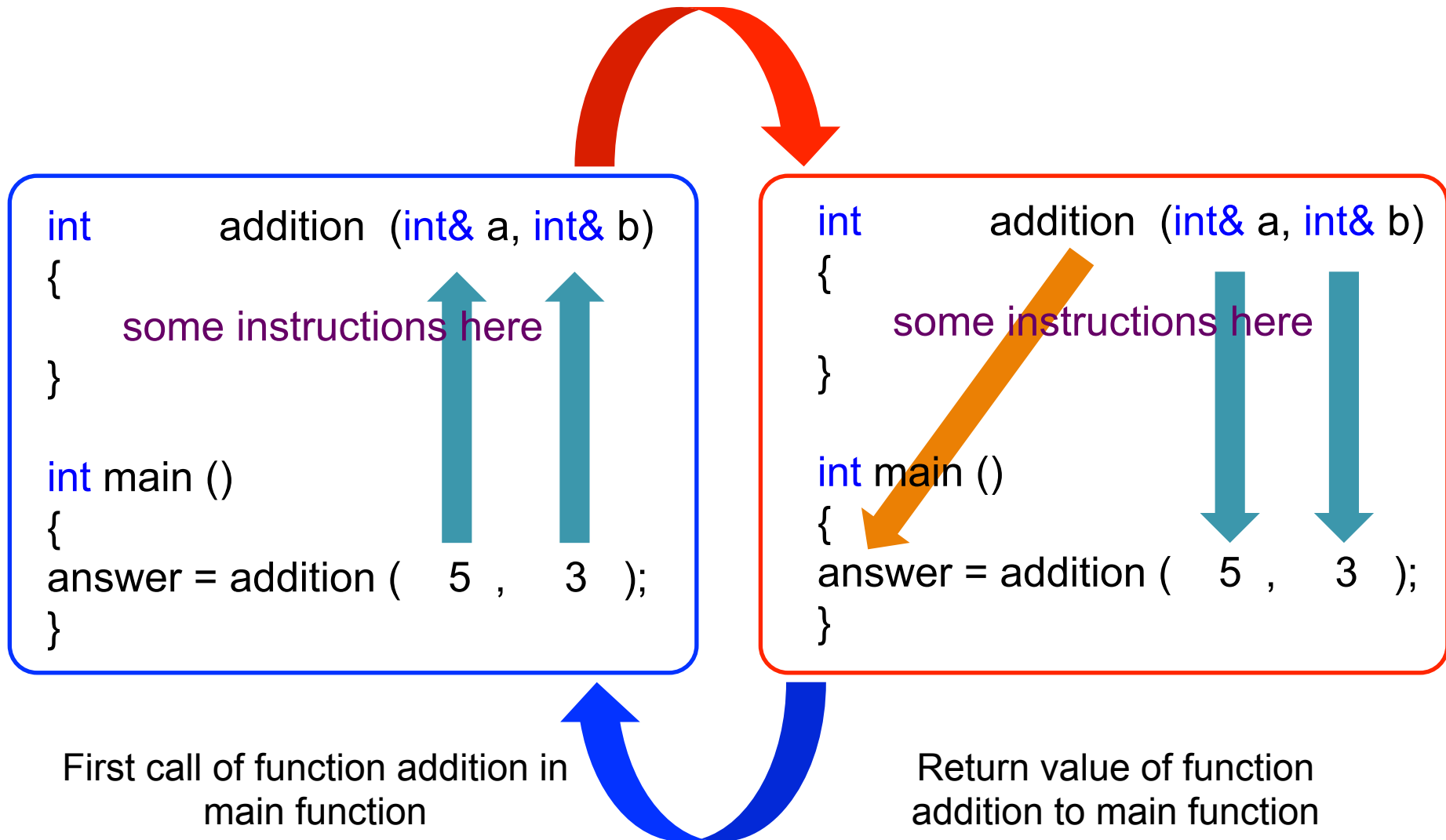
C++: A Crash introduction

Functions. Arguments passed by value.


```
int        addition  (int a, int b)
{
    some instructions here
}


int main ()
{
answer = addition (   5 , 3   );
}
```

```
int        addition  (int a, int b)
{
    some instructions here
}


int main ()
{
answer = addition (   5 , 3   );
}
```

First call of function addition in main function

Return value of function addition to main function

C++: A Crash introduction

Functions. Arguments passed by reference.

```cpp
int        addition  (int& a, int& b)
{
    some instructions here
}

int main ()
{
answer = addition (   5  ,    3   );
}
```

```cpp
int        addition  (int& a, int& b)
{
    some instructions here
}

int main ()
{
answer = addition (   5  ,    3   );
}
```

First call of function addition in main function

Return value of function addition to main function

C++: A Crash introduction

Functions

- The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. Therefore, the scope of local variables is limited to the same block level in which they are declared.

- Nevertheless, we also have the possibility to declare global variables; these are visible from any point of the code, inside and outside all functions.

- If a function does not returns a value, it should be declared as a void function.

- Two different functions can have the same name only if their parameter types or number of parameters are different, this practice is called **overloading functions**.

Functions

Declaring functions (both formats are equivalent):

```
#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;
}
```

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

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
}
```

Arrays

An array is a fixed number of elements of the same type stored sequentially in memory. Therefore, an integer array holds some number of integers, a character array holds some number of characters, and so on. The size of the array is referred to as its dimension. To declare an array in C++, we write the following:

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.

To declare an integer array named coor, which is made up of four elements, we write

int coor [4];

Arrays

Like normal variables, the elements of an array must be initialized before they can be used; otherwise we will almost certainly get unexpected results in our program. There are several ways to initialize the array. One way is to declare the array and then initialize some or all of the elements:

```
int   coor [4];
coor [0] = 1;
coor [1] = 0;
coor [2] = 3;
coor [3] = 7;
```

Another way is to initialize some or all of the values at the time of declaration:

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

Sometimes it is more convenient to leave out the size of the array and let the compiler determine the array's size for us, based on how many elements we give it:

```
int coor [ ] = { 1, 0, 3, 7, 2, 5, 2, 11 };
```

Here, the compiler will create an integer array of dimension 8.

C++: A Crash introduction

Arrays

C++ also supports the creation of multidimensional arrays, through the addition of more than one set of brackets. Thus, a two-dimensional array may be created by the following:

type arrayName [dimension1][dimension2];

The array will have **dimension1 x dimension2** elements of the same type. The first index indicates which of **dimension1** sub-arrays to access, and then the second index accesses one of **dimension2** elements within that sub-array. Initialization and access thus work similarly to the one-dimensional case:

```
int  coor [2][4];
coor [0][0] = 1;
coor [0][1] = 0;
coor [0][2] = 3;
coor [0][3] = 7;
coor [1][0] = 2;
coor [1][1] = 5;
coor [1][2] = 2;
coor [1][3] = 11;
```

Arrays

The array can also be initialized at declaration in the following ways:

       int coor [2][4] = { 1, 0, 3, 7, 2, 5, 2, 11 };

       int coor [2][4] = { { 1, 0, 3, 7 } , { 2, 5, 2, 11 } };

Note that dimensions must always be provided when initializing multidimensional arrays, as it is otherwise impossible for the compiler to determine what the intended element partitioning is.

Multidimensional arrays are merely an abstraction for programmers, as all of the elements in the array are sequential in memory. Declaring int coor[2][4]; is the same thing as declaring int coor[8];.

C++: A Crash introduction

Arrays

To initialize all elements of a multidimensional array to 0 (or any other value), we do as follows:

```cpp
int coor [2][20] = { 0 };
```

Similar for a normal array:

```cpp
int coor [4] = { 0 };
```

Arrays can be also initialized (and accessed) by using for loops, as follows:

```cpp
int   coor [2][20];
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 20; j++)
    {
        coor[i][j] = 0;
    }
}
```

Arrays

Accessing the values of an array. In any point of a program in which an array is visible, we can access the value of any of its elements individually as if it was a normal variable, thus being able to both read and modify its value. The format is as simple as:

array_name[index]

Following the previous examples in which coor has 4 elements and each of those elements was of type int,

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

For example, to store the value 30 in the third element of coor, we write the following statement:

coor[2] = 30;

To pass the value of the third element of coor to a variable named a, we write:

a = coor[2];

Therefore, the expression coor[2] is for all purposes like a variable of type int.

Arrays

Arrays as parameters. If we want to pass a single-dimension array as an argument in a function, the only thing we need to do when declaring the function is to specify in its parameters the element type of the array, an identifier and a pair of void **square brackets [ ]**. For example, the function:

```
void myFunction(int param[ ])
{
      //some instructions here
}
```

accepts a parameter of type array of int, named param. In order to pass to this function an array declared as

```
int coor [100];
```

It would be enough to write a call like this:

```
myFunction (coor);
```

C++: A Crash introduction

Arrays

For example:

```cpp
#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;
}
```

In this program, the first parameter (int arg [ ]) in the function **printArray**, accepts any array whose elements are of type int, whatever its length. For this reason we have included a second parameter that tells the function the length of each array that we pass. This allows the for loop to know the range to iterate.

Arrays

- In a function declaration it is also possible to include multidimensional arrays. For a two dimensions array, the format is: **type name [ ] [dimension2]**.

- For example, a function with a multidimensional array as argument could be:

void myFunction (int param [ ][4])

Notice that the first brackets are left blank while the following ones are not. This is because the compiler must be able to determine within the function which is the size of each additional dimension.

- Alternatively, you could also define the dimension of the array when passing it to the function as a parameter, as follows:

void myFunction (int param [2][4])

void myFunction (int param [4])

- When passing an array to a function, we pass the starting address of the array. As a consequence of this we have direct access to the array. This means that any change we do to the array passed to the function is a change to the original array.

C++: A Crash introduction

Pointers

- Pointers are one of the most powerful and confusing aspects of the C++ language. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, can not be performed without them.

- 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:

    type *var-name;

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk is being used to designate a variable as a pointer. Some examples of pointers are:

    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++: A Crash introduction

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.

```
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 (on my computer):

0012FF7C

0012FF7C

C++: A Crash introduction

Pointers

- **Using Pointers (Dereferencing pointers):** The other operator that is commonly used with pointers is the dereference operator (*). A dereferenced pointer evaluates the contents of the address it is pointing to.

    ```cpp
    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
    ```

The above program prints (on my computer)

    0012FF7C

    5

    0012FF7C

    5

In other words, when pnPtr is assigned to &nValue: pnPtr is the same as &nValue and *pnPtr is the same as nValue

Pointers

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++: A Crash introduction

Namespaces

- Namespaces address the problem of naming conflicts between different pieces of code.

- For example, we might be writing some code that has a function named **myFunction ( )**. If we decide to use a third-party library, which also has a **myFunction ( )** function. The compiler has no way of knowing which version of **myFunction ( )** we are referring to within our code. We can not change the library's function name, and it would be a big pain to change your own.

- A namespace is designed to overcome this difficulty and is used to differentiate similar functions, classes, variables etc., that have the same name available in different libraries. Using namespace, we can define the context in which names are defined. In essence, a namespace defines a scope.

- **OpenFOAM® uses namespaces.**

C++: A Crash introduction

Namespaces

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

namespace first_space          // first name space
{
        void func()
        {
                cout << "Inside first_space" << endl;
        }
}
namespace second_space         // second name space
{
        void func()
        {
                cout << "Inside second_space" << endl;
        }
}

int main ()
{
  // Calls function from first name space.
  first_space::func();

  // Calls function from second name space.
  second_space::func();

  return 0;
}
```

- A namespace definition begins with the keyword namespace followed by the namespace name as follows:

  namespace namespace_name
  {
         // code declarations
  }

- To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

  namespace_name::code;
  // code could be variable or function.

Defined Data Types (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:

    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. For example:

    typedef vector<double> doubleVector;

    In this way:

    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++: A Crash introduction

Classes

- A class is nothing else but an user defined data type (I like to see them as user defined data types on steroids).

- 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. That is, in one class I can declare the variables type and what I want to do with those variables (functions).

- The data and functions within a class are called members of the class.

- The members of the class can be private, protected, or public access. These access specifier determine how the members are accessed.

- By default, all members of a class have private access for all its members.

Classes

- Classes are generally declared using the keyword class, with the following 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++: A Crash introduction

Classes

- An access specifier is one of the following three keywords: private, public or protected. These specifiers modify 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.

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 a object (i.e., a variable) of this class called **rect**. This class contains 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 ( )**, of which for now we have only included their declaration, not their definition.

C++: A Crash introduction

Classes

- In the previous declaration of the **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();

- In the next slide we show a complete program for the class **CRectangle**, where we declare the class with all its members, then the functions definition and then we access the class within the main function.

C++: A Crash introduction

Classes

```cpp
#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;
}
```

- The most important new feature in this program is the scope operator (**::**), included in the definition of **set_values ( )**. 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.

C++: A Crash introduction

Class Constructor

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

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

//Definition of the constructor function
CRectangle::CRectangle ( int a , int b )
{
        x = a;
        y = b;
}

int main ( )
{
        CRectangle recta   (3,4);
        CRectangle rectb (5,12);
        cout<<"recta area: "<< recta.area ( ) << endl;
        cout<<"rectb area: "<< rectb.area ( ) << endl;
        return 0;

}
```

- Objects generally need to initialize variables or assign dynamic memory during their creation to become operative and to avoid returning unexpected values during their execution.

- In order to initialize classes, a special function called constructor is used. It is automatically called whenever a new object of the class is created.

- This constructor function must have the same name as the class and can not have any return type, not even void.

- Constructor can not be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created.

C++: A Crash introduction

Class Destructors

```cpp
class CRectangle
{
        int x, y;
public:
        CRectangle ( int , int );       //Constructor
        ~CRectangle ( );                //Destructor
        int area ()                     //Member function
        {
                return (x*y);
        }
};
//Definition of the constructor function
CRectangle::CRectangle ( int a , int b )
{
        x = a;
        y = b;
}
//Definition of the constructor function
CRectangle::~CRectangle ( )
{
        //delete x;     //for dynamic memory
        //delete y;     //for dynamic memory
}
```

- We can also use a special function called destructor. A destructor does the opposite of a constructor. It is automatically called when an object is destroyed, either because its scope of existence has finished or because it is an object dynamically assigned and it is released using the operator delete.

- As for constructors, the destructors must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

- The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

C++: A Crash introduction

Classes

- If we do not declare any constructor and destructor in a class definition, the compiler assumes the class to have a default constructor and destructor with no arguments.

- It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid data type, so we can use the class name as the type for the pointer. For example:

  CRectangle * prect;

  is a pointer to an object of class **CRectangle**.

- In order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (**->**) of indirection.

  prect -> set_values (2,4)

- **In OpenFOAM® there is a wide number of classes defined. They are used to define, discretize and solve PDE systems.**

Templates

Templates are the foundation of generic programming which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function.

- **Function templates:** functions templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

- **Class templates:** we can also write class templates, so that a class can have members that use template parameters as types.

Function Templates

- The general form of a template function definition is as follows:

```
template <class identifier> identifier function_name (parameter list)
{
    // body of function
}
```

- For example, to create a template function that adds two numbers:

```
template <class T>  T addval(T a, T b)
{
    return (a + b);
}
```

Here, we have created a template function with T as its identifier. The template identifier represents a type that has not yet been specified. The function **addval**, returns the addition of the two parameter of this still undefined type.

Function Templates

- To use this function template, we use the following format for the function call:

   function_name <type> (parameter list);

- For example, to call the function **addval** to add two variables of type int:

   int x, y;

   addval <int> (x,y);

   When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of T by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

- In the next slide we show the whole program

C++: A Crash introduction

Function Templates

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

template <class T>
T addval(T a, T b)
{
    T result;
    result = a + b;
    return (result);
}

int main ( )
{
    int k = 6, m = 4, p;
    double n = 3.14, r = 13.01, u;
    p = addval <int> (k,m);
    u = addval <double> (n,r);
    cout << "The result is " << p << endl;
    cout << "The result is " << u << endl;
    return 0;
}
```

In this case, we have used T as the template identifier, but you can use any name identifier you like.

In this example, we used the function template **addval** twice. The first time with arguments of type int and the second one with arguments of type double. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the **addval** template function even to declare new objects of that type:

    T result;

Function Templates

- We can also define function templates that accept more than one type parameter, simply by specifying more template identifiers between the **angle brackets < >**. For example

        template <class T, class U> T addval(T a, U b)

        {

            T result;
            result = a + b;
            return (result);

        }

In this case, our function template **addval ( )**, accepts two identifiers of different types and returns an object of the same type as the first identifier that is passed. After this declaration, we can call **addval( )**:

        int x;

        double y;

        addval <int, double> (x,y);

Class Templates

- The general form of a template class definition is as follows:

template <class identifier> class class_name

{

    // body of the class

}

- For example, to create a template class:

```
template <class T>
class twovals
{
        T values [2];
public:
        twovals (T first, T second)
        {
                values [0] = first;
                values [1] = second;
        }
} ;
```

Function Templates

- The class template that we have just defined, stored two elements of any valid type. For example, if we want to declare an object of this class to store two integer values with the values 100 and 25:

> **twovals\<int\> my_object (100,25);**

- This same class template can be used to create an object to store any other type

> **twovals\<int\> my_object (3.14159, 343.3);**

- The only member function in the previous class template has been defined inline within the class declaration itself. In case that we want to define a function member outside the declaration of the class template, we must always precede that definition with the template < > prefix:

- In the next slide we show a sample program.

C++: A Crash introduction

Class Templates

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

template <class T>
class twovals
{
            T a, b;
        public:
            twovals (T first, T second)
            {
                a = first;
                b = second;
            }
            T getmax ();
} ;
template <class T>
T twovals <T>::getmax ()
{
        T retval;
        retval = a>b ? a : b;
        return retval;

}

int main ( )
{
        twovals <int> myobject (100,75);
        cout << myobject.getmax ( ) << endl;
        return 0;

}
```

In this case, we have used T as the template identifier, but you can use any name identifier you like.

Notice the syntax of the definition of member function **getmax ( )**

> **template <class T>**
> **T twovals <T>::getmax ()**

Confused by so many T?. There are three T in this declaration. The first one is the template identifier. The second T refers to the type returned by the function. And the third T (the one between **angle brackets < >**) is also a requirement, it specifies that this function's template identifier is also the class template identifier.

Class and Functions Templates

- If you look for the definition of template in a dictionary, you will find a definition that is similar to the following:

    "a template is a model that serves as a pattern for creating similar objects".

- One type of template that is very easy to understand is that of a stencil. A stencil is an object (e.g. a piece of cardboard) with a shape cut out of it (e.g. the letter J). By placing the stencil on top of another object, then spraying paint through the hole, you can very quickly produce stenciled patterns in many different colors.

- Note that you only need to create a given stencil once, you can then use it as many times as you like to create stenciled patterns in whatever colors you like.

- Even better, you don't have to decide the color of the stenciled pattern you want to create until you decide to actually use the stencil.

C++: A Crash introduction

Class and Functions Templates

- At a first glance Class and Functions Templates are difficult to understand, you just need to get use with the syntax.

- In C++, templates serve as a pattern for creating other similar functions or classes.

- The basic idea behind templates in C++ is to create functions and classes without having to specify the exact type of some or all of the variables.

- Instead, we define the function or class using placeholder types, called template type parameters (identifiers). Once we have created a function using these placeholder types, we have effectively created a function stencil or class stencil.

- To make the code easier to read OpenFOAM® re-defines the templated class names, for instance:

      typedef List<vector> vectorList;

  so that an object of the class template List of type vector is called vectorList.

- ## OpenFOAM® uses class and function templates a lot!

C++: A Crash introduction

C++ Standard Template Library (STL)

- The C++ STL library is a powerful set of C++ template classes that provide general purpose templated classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

- The core of the C++ STL library is made up of three well-structured components:

    - **Containers:** they are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, etc.

    - **Algorithms:** they act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.

    - **Iterators:** are used to step through the elements of collections of objects.

- **OpenFOAM® uses its own template library, which is STL conforming. In principle it works similar to the STL library.**

C++: A Crash introduction

C++ Standard Template Library (STL)

- A full discussion about all the three C++ STL components is outside of the scope of this lecture. From now on, keep in mind that all the three components have a rich set of pre-defined functions which help us in doing complicated tasks in a very efficient and easy fashion.

- Let us take a look at the following program to demonstrates the vector container (a C++ Standard Template) which is similar to an array with an exception that it automatically handles its own storage requirements in case it grows (next slide).

C++: A Crash introduction

C++ Standard Template Library (STL)

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

int main()
{
    // create a templated vector object to store int
    vector<int> vec;
    int i;

    // display the original size of vec
    cout << "vector size = " << vec.size() << endl;

    // push 5 values into the vector
    for(i = 0; i < 5; i++)
    {
        vec.push_back(i);
    }
```

First part of the code

```cpp
    // display extended size of vec
    cout << "extended vector size = " << vec.size()
    << endl;

    // access 5 values from the vector
    for(i = 0; i < 5; i++)
    {
    cout << "value of vec [" << i << "] = " << vec[i]
    << endl;
    }

    // use iterator to access the values
    vector<int>::iterator v = vec.begin();

    while( v != vec.end())
    {
    cout << "value of v = " << *v << endl;
    v++;
    }

return 0;
}
```

Second part of the code

C++: A Crash introduction

- In the **$path_to_openfoamcourse/c++_tuts** folder, you will find the source code of all the examples we illustrated in the previous slides.

- To compile them:

    - g++ file_name.C -o executable_name

    - ./executable_name

- Take your time, and try to understand the concepts implemented in these examples.

- Also, try to get familiar with the syntax.

C++: A Crash introduction

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

Mesh generation using open source tools

Additional tutorials

In the folder **$path_to_openfoamcourse/c++_tuts,** you will find many tutorials, try to go through each one to understand the basic concepts of C++.