C++

introduction
C++ has different version of standard, 11,14 till 17, on unix to compile a c++ program we ca use
*g++ -std=c++(**11**,**14**,**17**..) -Wall -o out source.cpp*
we can use also Qmake or Cmake...

Coding (Important part)
Some features are similar to C programming.
Code structure and basic operations
in c++ code structure is similar to c program, we have libraries on the top, we can import a library using: #include <libraryName>

we have main function, were our program starts. We can divide programe in **namespace.**
For instance to print something on the screen we use
```
std::cout<<"hello world"<<std::endl;
```
where std:: is the namespace where is contained the function cout or endl, the operators << are overloaded, next pages will show this feature, the :: operator is called **scope resolution operator**.
Input and Output
In order to print, write something on stdout, stderr and stdin we use the iostream library.
Inside it we have the std namespace that contains:
- **cin:** it reads on standard input
- **cout**: it writes on standard output
- **cerr**: it writes on standard error

endl insert a newline in the stream, forcing the flush.
Namespace
```
namespace test{
    void foo(){
            ....
    }
    namespace testInner{
....
}
}
void foo(){
            ....
    }
 int main(){
    test::foo();//namespace function
    ::foo();//global function
    test::testInner::foo(1,2);
 }

```
if we define a global function we use ::function() to call gloval namespace.

We can define sub-namespaces and we call that in cascade using the formula:
A::B:: ... ::func();
We can omit the name space when we call the functions if we use

using namespace *NamespaceName*, after this declaration the functions will be part of the global namespace.
If there are functions that have the same name and parameters will be chose the global method and the namespace method must be called specifying the namespace path.

Functions
in c++ like c functions must be declared before use.
In the declaration parameters is need only the type of the arguments, the name can be put only in the implementation section.
Functions works like C, prototype or declare implementation and declaration in once function used later in the file.

C++ support function overloading, so change arguments type or number, but maintaining the same name.
In C++ we can declare default values.
To do that we assign value in the declaration, **but we cannot declare arguments with default before arguments with no default.**
Rule:
declare before mandatory arguments, then optional with default. When you call function with default you can omit parameter in the brackets

To resolve ambiguity between functions we cast parameters in the type that we need, cast equal C *type(value)*.

String
in c++ there is the support for the strings.
It belongs to standard library and we can get it importing string library.
We declare string and use in the following way:
```cpp
#include <string>
#include <iostream>
using namespace std;
int main(void){
    string str{"hello world"};//create a string
    cout<<str[1]<<endl;//print the char 2 of the stirng
    cout <<str<<" !"<<endl;//concat to string ! In the print
    str.insert(0,"1)");//insert 1) in the string at 0 position
    string tmp=str.substr(1,5);//get substring from char 1 to 5
    tmp.replace(0,5,"hello");/*replace from 0 position a string that contains hello and remove 5 chars*/
    cout<<tmp.find("hello",0)<<endl;//starting from 0 position search hello in the stirng
 }

```
as we can see there different ways to declare a variable, in c++ there are 4 choices:

1. **c style: int a=3**
2. c++ pre-11: int a(3)
3. c++ 11: int a={3}
4. **c++11: int a {3}**

1 and 4 are the most used, the point 4 not cast automatically, if we put 3.14 the point 1 cast to 3 the point 4 get error.

In c++ we can not declare the type of the variable, but we can put auto and the compiler will understand automatically the type of the variable.

Arrays
in c++ arrays works like C, same syntax but we can use foreach loop, we can also use traditional for loop equals to C.

```
.....
int myarray[]{1,2,3,4};
for(auto e: myarray)
{
    cout<<e<<endl;
}
...

```

Memory allocation
In c++ we can allocate a pointer using the keyword **new** and free space using **delete**.
To delete an array we must use **delete[]**

```
int* i0{new int};
char* p0{new char[10]}; // Array
int* p1{new int[5]}; // Array
delete i0;
delete[] p0;
delete[] p1;
```

nullptr assign a void pointer.

```
int * x{nullptr};
```

in C++ there isn't a garbage collector, programmer must clean space explicitly.
Is good choice try to allocate data on the stack or pass by reference.
Object in stack are de-allocated based of the scope where they are declared:

- **local scope**: end of the function
- **class scope**: instance of the class is destroyed
- **namespace**: program end

References
In C++ we can create alias to variables using

```
int& c{a}
```

the difference with pointer is that reference cannot change element to point after declaration.
It's useful to avoid copies when we pass parameters and avoid pointers.

If we pass values for value will be made a copy the values in the stack instead the reference passage works with the reference of the passed values.

If we want access to data with pointer we must de-reference the pointer using *ptr, but if use reference of c++ we can access like a normal variable.

In the arrays we can use reference in the foreach, like the example:

```
#include <iostream>
 using namespace std;
int main(void) {
    int myarray[] { 1, 5, 3, 6, 2};
    for (auto& i : myarray) {
            i++;
    }
    for (auto& i : myarray) { cout << i << endl; }
}
```

in functions we can return a reference, but we must be careful to not reference a local variable, because it will be deleted on the function return.
We must return the reference of once of our parameters declared as reference.
If we declare a reference parameter as const it will be read-only, we couldn’t edit the value of the pointed variable.
To pass a value, not a variable, to a function that has reference parameter there are 2 ways: the first is declare the parameter as const, the second one is declare a function with a reference that use && , it allowed that type of data, r-value reference(from c++11).

Enum
in c++ to define an enum we can do that like java

```
enum Animal{
    CAT,DOG,BIRD
}
enum class Animal{ //it is not automatically converted to int stringly typed
    CAT,DOG,BIRD
  }

```

Struct and Class
struct in c++ not needs typeof.

```
struct Person {
    string first;
    string last;
    Sex s;
    unsigned int age;
};

```

in c++ there is the support for OOP, hence we have classes!!!
we can create a class using **class** keyword. In c++ the structure of classes is the following

- definition: in MyClass.h file
- implementation in MyClass.cpp file

Example

```
class Fraction {
    public:
            int num() const;
            void num(int numerator);
            int den() const;
```

```
        void den(int denominator);
    private:
        int m_numerator {0}, m_denominator {1};
};

```

we can see we have **public** and **private** blocks, they define the access layer there also **protected** that allow access to the derived classes.
Two objects of the same class can access their private fileds and methods.

The difference between a class and a struct is:
struct has default fields as public instead class private.

In other words: struct T{...} = class T{public: ....}

we can also implement methods inline in the definition section.
Constructor
in c++ we can have multiple constructors, and is we not define nothing will be used default once.
The constructor has the following structure:
ClassName::ClassName (parameters) : init-list(optional){body }

The member initialization list is used to initialize the fields of a class. It is a list the initialize fields with constructor parameters.
We can put fields in the following way:

```
 1 class Fraction {
 2     public:
 3         Fraction() : m_numerator{0}, m_denominator{1} {};
 4         Fraction(int numerator, int denominator=1) :
       m_numerator{numerator},m_denominator{denominator} {};
 5         Fraction() : Fraction{0,1} {};
 6         int num() const;
 7         void num(int numerator);
 8         int den() const;
 9         void den(int denominator);
10     private:
11         int m_numerator {0}, m_denominator {1};
12  };
13
```

we can also chain constructors, in the init we pass constructor, line 5 of the upper code. Important, if we delegate constructor we can’t pass other parameters in the init-list, because object is already created.

Instantiate an Object
To instantiate an object we can use create it both on the stack and on the heap. We can allocate using new and free memory with delete keyword.
Examples:

```
Fraction f1; // 0/1
Fraction f2 {1, 2}; // 1/2
Fraction f3 {7, 5}; // 7/5
Fraction* f5{new Fraction{2,3}}; // 2/3
Fraction* f6{new Fraction(8)}; // 8/1
delete f5;
delete f6;

```
sometimes we need to access to private fields, c++ allowed this to the object that has the keyword **friend.**

```
class Node {
private:
    int key;
    Node* next;
    /* Other members of Node Class */
    // Now class  LinkedList can
    // access private members of Node
    friend class LinkedList;
};
```

We can define also the declaration of outside method that can access to our private fields, example at slide 38 of 2.C++Classes.pdf.
In general we can define like friend the following elements:

- friend class TrustedClass;
- friend int read(MyClass& c);
- friend void Test::look(MyClass& c);

Overload operators
when we print with cout we must use <<, this operator is normally used for bit shift, but in c++ we can overload operators.
This means that when we use that operator on a specific object with specific parameters it will perform something that we specify in the method.
How to do that?

```
Fraction& operator += (const Fraction& f) {
    int temp_numerator { f.m_numerator * m_denominator };
    m_denominator *= f.m_denominator;
    m_numerator *= f.m_denominator;
    m_numerator += temp_numerator; return *this;
}
Fraction& operator -= (const Fraction& f) {
    int temp_numerator { f.m_numerator * m_denominator };
    m_denominator *= f.m_denominator;
    m_numerator *= f.m_denominator;
    m_numerator -= temp_numerator; return *this;
 }

```
the structure is similar to method definition, but the name is composed by keyword operator and the operator that we want overload. The parameter is the right member: fractionA+=fractionB , with *return *this* we return the reference of object to itself.

```
Fraction operator+(Fraction a, const Fraction& b)

return a += b;
}

```

the number of parameter depends operator by operator, for example + needs 2 can have 2 or 1 because the meaning of the sign changes.
In the example we sum 2 fractions using the previous operator += and it will return to a the sum between a and b.

some operators can't be overloaded, because they are part of the language like $ or ".
we can also overload ++, -- *= … but not ?, :: and .

We can also overload new and delete if we need to do particular operations when we create/destroy and object.

Conversion constructor
They are constructors that will convert assign value to object using default values.

```
Fraction(int numerator, int denominator=1) : m_numerator{numerator},
m_denominator{denominator} {
     ...
};
```

if we use **explict keyword** on constructors we avoid automatic cast

```
explicit Fraction(int precision)
     : m_precision{precision}
            { // ...
     };
Fraction f {2,3};
f = 5;//error
```

Conversion operators
we can define operator to convert a type to an other

```
operator double () { return (double) m_numerator / m_denominator; }
void myfun(double d) { // ... }
Fraction x { 1, 2}; myfun(x); 0.5
```

Sub-Objects/Composition
We have to manage the memory when we create an object inside an other.
We have to declare **destructor.**
To declare an object composition we have to put a field typed as pointer.
Destructor are declared as constructor but with ~ before class name

```
~ClassName()
ClassName::~ClassName() { }

```

to clean memory we add in the destructor method the delete calls

```
class Box {
    public: Box(string e) : m_label{new Label{e}} {
            cout << "Creating Box" << endl;
    }
    ~Box() {
            cout << "Destroying Box" << endl; delete m_label;
     }
     private:
            Label* m_label;
};
```

RAII
it is a pattern that:

- Acquire resources in the constructor
- Release acquired resources in the destructor

if we assign an object to another and we have pointers old value will have garbage in the memory.
Solution : use copy constructors.

```
ClassName(const ClassName& other);
```

copy constructor copy the reference of the object passed as parameter and copy the right parameters on the new object, we have to overload the operator = to apply copy constructor.

```
Box(const Box& o) : m_label{
new Label{*o.m_label}
} { }
Box& operator=(const Box& o) {
     *m_label = *o.m_label; return *this;
}

```

A class **requires** one of the following methods, the others might be required:

- Destructor, there is default once if not defined.
- Copy constructors
- Copy assignment operators

Static members
the static elements are shared between all instances of a class.
To do in c++ use **static** keyword.

```
static int getinstances() { return instances; }
```

Standard library
contains some base functionality and several utility classes, such strings, streams, containers…

String allow manipulation of character arrays easily.

Stringstream concatenates different types into a string, it's like string builder in Java.

```
using namespace std;
int main() {
 stringstream sstr;
 for (int i=0; i<100000; i++) {
 sstr << i;
 }

 cout << sstr.str() << endl;
}
```

use << to add elements.

Containers are generic and are similar to Java’s collections, we have:

- vector
- list
- map
- set

Vector:

```
int main()
{
 vector<int> v1 {1,2,3,4,5,6};
 // Erase the 4th element
 v1.erase(v1.begin() + 3);
 v1.push_back(5);
 for(int x; cin >> x;) {
 v1.push_back(x);
 }
  for (const auto& x : v1) {
  cout << x << endl;
  }
  cout << "Size: " << v1.size() << endl;

}
```

like C arrays we can initialize data of a vector, but in c++ we can add this behavior to all classes that we develop.
We use the **initializer_list**

```
class MyContainer
{
public:
     MyContainer(const std::initializer_list<int>& values) {
              for (const auto& v : values) {
                       m_MyContainer.push_back(v);
             }
}
    private:
     vector<int> m_MyContainer;
};
int main()
{
     MyContainer c {1,2,3,4,5,6};
}
```

BEWARE!! the constructor that accept the initializer_list will always chose when we create the object with {}

List:

```
list<int> l1 {1,2,3,4,5,6};
l1.push_back(5);
l1.push_front(5);
l1.pop_back();
l1.pop_front();
```

Map:
```
map<string,int> contacts { {"Foo", 123456}, {"Bar", 2342342}};
 contacts ["Eve"] = 32333;
 if (contacts.count("Foo") == 1) {
      cout << "Foo exists" << endl;
 }
 contacts.erase ("Bar");
```

Pointer and ownership

To avoid problems of ownership of memory elements we exploit **smart pointers.**

- **unique_ptr**: it is a smart pointer which cannot be shared and implements unique ownership
- **shared_ptr**: it is a smart pointer which allows for shared ownership

unique_ptr overload * to be dereferenced; if we pass an unique_ptr ti a function as parameter compiler stop us, because unique_ptr **CAN'T** be copied.
To move a unique_ptr we must use **move(ptr)** it will change the owner of the object referenced.

shared_ptr can share access to a pointer and when the last copy is destroyed the associated memory is released.
make_shared allocates and return a pointer wrapper inside a shared_ptr.
Both unique_ptr and shared_ptr has get and release method, get will return the wrapped pointer and release, release the ownership.

Smart pointer should be used when we don't intend to take a ownership use a normal pointers, if we want to take shared ownership use shared if single use unique.
Exceptions
```
try {
 throw 42;
 } catch (int e) {
 cout << "Exception!Value " << e << endl;
 }
```
when try-catch detects an exception stack element are destroyed when catch is running.
Heap objects must be cleaned by programmers.
They don't support finally block.
Like java with **throw** we can lunch exceptions. With the keyword no-except means that the code declared can't throw exceptions.

Lippincott functions
they are functions to wrap the handling of many different exceptions into a single reusable function.
It is a function which manage the exceptions, in other words

```
void handle_exception() {
 try {
 throw;
 } catch (const out_of_range& e) {
 // handle out_of_range
 cout << "Handling out_of_range\n";
 } catch (const runtime_error& e) {
 // handle runtime_error
 cout << "Handling runtime_error\n";
  }
 }
```

Inheritance and polymorphism
To extend a class in C++ we use the following syntax:

```
class Lecturer : public Employee {...
```

we can inherit only public and protected fields!!!
the methods' access of the base class will transform based on modification access. If we use private inheritance lose sense.
Classname() method return the name of the class.

If we want access to pointer's data we use → for stack variable or reference use dot notation.

If I pass an object in a variable that has base class' type will be copy only the base class section the remain will be lost, no hiding like java.
There are 2 type of binding:

- **Static binding**: it bind classes at compilation, not allow polymorphism if we have a derived class and we call classname it will return always base class name, default binding
- **Dynamic binding:** it is done run-time and it support polymorphism.

We enable it using **virtual** keyword.

To make a polymorphic method we add the keyword virtual in the signature.

```
class Employee
{
public:
Employee(string name, string, institute, int nr);
virtual string classname() const;
...
```

pointers to virtual methods are stored inside a table referenced from the class structure.
The vtable is shared between all instances of the class.
To use the base class methods we use BaseClassName::method()

Abstract classes

Abstract classes can’t be instantiated and to get it in c++ we have to put all methods equal 0 and virtual, they will become pure virtual method in this way.
This class are comparable to interfaces in Java.
virtual string name() const = 0;

A concrete class implements the methods contained in an abstract class.

Hiding: if we have a method in the base class and we redefine the same method with same name but different parameters, all methods with the same name in the base class will be hidden.
There are many solutions, delegate calling base class method in the new, or using base class importing with:

```
using BaseClass::method;
```

Some methods can’t be inherit like

- copy constructor
- copy assignment operator
- destructors

we can declare a destructor virtual to override the operations called when the object will be destroyed.
Destructor should always be declared as virtual.
Final keywords needs to avoid inheritance and overriding, we can put it after class name or after () of a method.
If we use delete instead of default we make the method inaccessible. If we put constructor equal default we are asking to the compiler to reinsert the default implementation of method.

Multiple inheritance

in c++ we can have multiple inheritance.
To extend classes we put all classes that we need to extend separated by “,” in the same place where we extend 1 class. All classes need also the access modifier

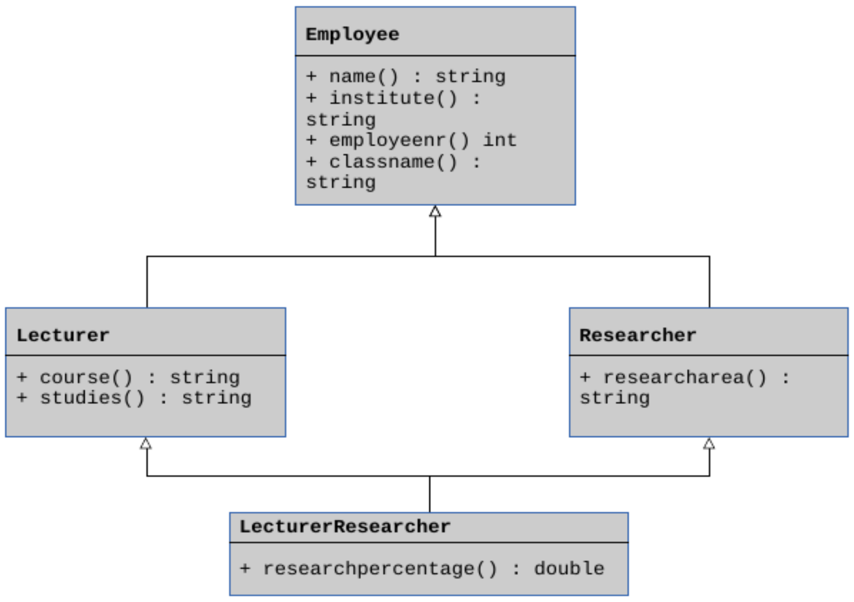
```
class LecturerResearcher : public Lecturer, public Researcher
{
	public:
	LecturerResearcher(string name, string institute, int nr,
	string studies, string course, string
	researcharea, double researchpercentage)
...
```

The order is important, constructors will call in the same order of the calls.

To avoid duplication and ambiguity we must put base class as virtual.

```cpp
class Lecturer : public virtual Employee
{
        // ...
};
class Researcher : public virtual Employee
{
        // ...
};
```

to avoid problems in the highest class of the hierarchy we call the constructor in the lowest class and we init it from our final object.



```cpp
LecturerResearcher(string name, string institute, int
nr,
string studies, string course, string researcharea,
double researchpercentage)
: Employee{name, institute, nr},
Lecturer{name, institute, nr, studies, course},
Researcher{name, institute, nr, researcharea},
m_researchpercentage{researchpercentage}
{}
```

in c++ there are 4 types off hars.

- **Static_cast**:
- **Dynamic_cast**
- **Reinterpret_cast**
- **Const_cast**

Static_cast:

```
static_cast<type>(object);
```

We use static to cast a pointer from/to base class to derived. Cast of objects/values (for example, from int to float).
We can also convert enum.
Dynamic_cast:

```
dynamic_cast<type>(pointer_or_reference);
```

we use dynamic to upcast from derive to a base class, downcasting from base class to a derived if there is polymorphism, pointer must have at least one virtual method.
Dynamic can cast **only references/pointers, no cast for objects and values!!!.**
If dynamic_cast fails return 0, so we can simulate an instanceof, if we need to know if some object belong to some class.
Reinterpret_cast:

```
reinterpret_cast<type>(pointer_or_reference);
```

It is used to convert a pointer of some data type into a pointer of another data type, even if the the data types before and after conversion are different.

it allows to access to memory structure as a different type, it cannot use with multiple or virtual inheritance.
It cannot convert objects/values only references or pointers.
When use it:

- reinterpret_cast is a very special and dangerous type of casting operator. And is suggested to use it using proper data type i.e., (pointer data type should be same as original data type).
- It can typecast any pointer to any other data type.
- It is used when we want to work with bits.
- If we use this type of cast then it becomes a non-portable product. So, it is suggested not to use this concept unless required.
- It is only used to typecast any pointer to its original type.
- Boolean value will be converted into integer value i.e., 0 for false and 1 for true.

Const_cast:

```
const_cast<type>(pointer_or_reference);
```

it allows to remove const or volatile, it cannot cas objects and values.
If something is declared const we can cast his value only with this cast.

```
int main() {
const Base* b{new Base};
b->foo(); // Ok
//b->bar(); // Error! Not const!
//Base* bc = static_cast<Base*>(b); // Not permitted
//Base* bc = reintepret_cast<Base*>(b); // Not permitted
//Base* bc = dynamic_cast<Base*>(b); // Not permitted
Base* bc = const_cast<Base*>(b); // Ok
bc->bar(); // Ok
}
```

typeid function allows to determinate th type of an object at runtime, it is contained in the library typeinfo.