PROGRAMMAZIONE AVANZATA

// Abstract

Una classe abstract è una classe che non può essere istanziata. Poichè in C++ non esistono le interfacce (ma esiste l'ereditarietà multipla), possiamo dire che le classi astratte sono equivalenti alle interfacce.
Per definire una classe astratta abbiamo bisogno di almeno un metodo virtuale puro, di cui (in caso di specializzazione della classe) abbiamo bisogno di fare *override*.

Se la virtual non è implementata (virtual void metodo() = 0;) l'intera classe viene vista come abstract quindi non puoi istanziare oggetti di quella classe e il virtual deve essere overridato, se invece è implementata la classe non diventa strana e l'override diventa facoltativo. Per il metodo chiamato funziona come c# tranne per i puntatori sghembi che sono nell'esempio.

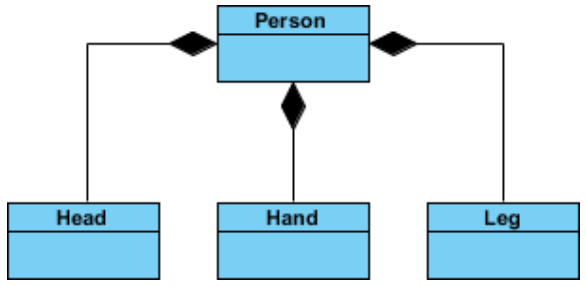
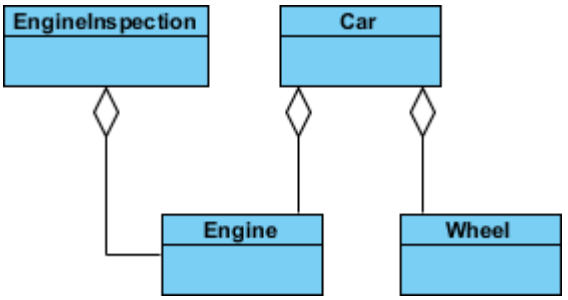
```
#include <iostream>

class A { // classe astratta (o interfaccia)
    public:
        virtual void metodo() = 0;
};

class B: public A {
    public:
    void metodo() {
        std::cout << "Triple A battery" << std::endl;
    }
};

int main(void) {
    //A a; // Non può essere istanziata
    B b; // Può essere istanziata...
    b.metodo(); // ...e il membro richiamato
    return 0;
}
```

// Associazioni: aggregazioni e composizioni

Aggregation and Composition are subsets of association meaning they are specific cases of association. In both aggregation and composition object of one class "owns" object of another class. But there is a subtle difference:

- **Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.
- **Aggregation** implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

Composition Example:
We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also deleted as a result.
Aggregation Example:
It's important to note that the aggregation link doesn't state in any way that Class A owns Class B nor that there's a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link is usually used to stress the point that Class A instance is not the exclusive container of Class B instance, as in fact the same Class B instance has another container/s.

	
COMPOSIZIONE	AGGREGAZIONE

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

typedef enum {DIESEL, BENZINA, GPL} Tcombustibile;
class Combustibile{
    private:
    Tcombustibile comb;
    public:
    Combustibile(Tcombustibile _comb):comb(_comb) {}
    ~Combustibile() {}
    void stampa() const {
        switch (comb){
            case DIESEL: cout << "Diesel"; break;
            case BENZINA: cout << "Benzina"; break;
            case GPL: cout << "Gpl"; break;
            default: cout << "BOH";
        }
    }
};

class Motore{
    private:
    Combustibile* tipo; //losanga vuota 1,1
    Combustibile* tipo2; //losanga vuota 0,1
    public:
    Motore(Combustibile* _tipo) {
```

```
        tipo=_tipo;
        tipo2 = NULL;
    }
    ~Motore() {}
    void stampa() const {
        cout << "[MOTORE tipo:";
        tipo->stampa();
        if (tipo2!=NULL) {
            cout << " tipo2:";
            tipo2->stampa();
        }
        cout << "]";
    }
    void setTipo2(Combustibile* _tipo){
        tipo2 = _tipo;
    }
};

class Auto{
    private:
    Motore motore;  //losanga piena 1,1, composizione
    Motore* motscorta;//losanga piena 0,1, aggregazione
    string marca;
    public:
    Auto(Combustibile* _tipo,string _ma): motore(_tipo){
        marca=_ma;
        motscorta=NULL;
    }
    ~Auto() {
        if (motscorta!=NULL)
            delete motscorta;
    }
    void stampa()const {
        cout << "[AUTO marca:" << marca << " motore:";
        motore.stampa();
        if (motscorta!=NULL){
            cout << " motscorta:";
            motscorta->stampa();
        }
        cout << "]";
    }
    void setMotscorta(Combustibile* _tipo) {
        motscorta = new Motore(_tipo);
    }
};

int main() {
    Combustibile c1 (BENZINA);
    Combustibile c2 (DIESEL);
    Motore motorescorta (&c1);
    Auto a (&c1, "Fiat");
    a.setMotscorta(&c2);
    a.stampa();
```

```
    return 0;
}
```

// Auto, vector, range

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
using namespace std;

// VECTOR MODIFIERS:
// assign(times, value); push_back(value); pop_back(); ...
// insert(pos, value); erase(pos); clear;
// VECTOR CAPACITY STUFF:
// size(); max_size(), resize(n); empty(); capacity();
// VECTOR ITERATORS:
// begin(), end(): normal iterators pointing to first/last element;
// rbegin(), rend(): reverse iterators pointing to last/first element;
// cbegin(), cend(): const. normal iter.s pointing to f/l element;
// crbegin(), crend(): const. reverse iter.s pointing to l/f element;

int main(int argc, char** argv) {
      vector<int> v;
      //vector<int>::iterator it;
      //vector<int>::reverse_iterator rit;
      for (int i = 0; i < 5; i++) v.push_back(i);
      for (auto it = v.begin(); it!=v.end(); it++) cout << " " << *it;
      cout << endl;
      cout << "[VECTOR] reverse iterator with scope variable" << endl;
      vector<int>::reverse_iterator rit;
      for (rit = v.rbegin(); rit!=v.rend(); rit++)
            cout << " " << *rit;
      reverse(v.begin(), v.end()); // from <algorithm>
      for (auto val : v)
            cout << " " << val;
      auto lambda = [](int val) { cout << " " << val; };
      for_each(v.begin(), v.end(), lambda);
      return 0; }
```

Auto:The auto keyword specifies that the type of the variable that is being declared will be automatically deducted from its initializer. In case of functions, if their return type is auto then that will be evaluated by return type expression at runtime.
Vector: Vector is a template class in STL (Standard Template Library) of C++ programming language. C++ vectors are sequence containers that store elements.
C++ vectors can automatically manage storage. It is efficient if you add and delete data often. Bear in mind however, that a vector might consume more memory than an array.
Vector is a type of dynamic array which has the ability to resize automatically after insertion or deletion of elements. The elements in vector are placed in contiguous storage so that they can be accessed and traversed using iterators. Element is inserted at the end of the vector.

List: List is a double linked sequence that supports both forward and backward traversal. The time taken in the insertion and deletion in the beginning, end and middle is constant. It has the non-contiguous memory and there is no pre-allocated memory.
Range: Uguale al ciclo for ma è solo più intuitivo da leggere

// Bitset

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

void stampa(auto b) {
      cout << b << " b (" << b.size() << ")" << endl; }

int main(int argc, char** argv) {
      bitset<32> b;
      stampa(b);
      string s = "101010000000000000";
      b=bitset<32>(s);
      stampa(b);
      stampa(b<<4); // shifta verso sinistra di 4 bit
      stampa(b>>8); // shifta verso destra di 8 bit
      stampa(b & b); // and con se stesso
      stampa(b | b); // or con se stesso
      stampa(~b); // not ovunque
      cout << b.to_ulong() << endl;
      b.flip(); // not ovunque ma lo salva
      stampa(b);
      cout << b.to_ulong() << endl;
      b.reset(24); // mette a 0 il 24° bit
      stampa(b);
      return 0; }
```

A bitset stores bits (elements with only two possible values: 0 or 1). The class emulates an array of bool elements, but optimized for space allocation: generally, each element occupies only one bit. Each bit position can be accessed individually

// Const

`const int* const FUNCTION (const int* const & PARAM) const {...}`				
the returned left-value will be const	the returned right-value will be const	the passed left-value will be const	the passed right-value will be const	the function will treat other class members as const

// Conversioni

```
double x = 10.3;
int y;
short c = 200;
y=c;               //implicit cast
notation
y = int (x);    // functional
notation
y = (int) x;    // c-like cast
notation
```

// Eccezioni

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.
To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.
An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

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

int main () {
  try
  {
    throw “aa”/2;
  }
  catch (char*/int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
  return 0;
}
```

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

struct MyException : public exception {
   const char * what () const throw () {
      return "C++ Exception";
   }
};
int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
```

```
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

// Friend

The “friend” keyword allow out-of-scope functions and classes to access private or protected members of another class.
E. g.: the Car class enables read-only access to its members (such as speed) from the global scope. By writing “`friend void makeCarFasta(Car&);`” we specify that any function with the same signature (outside of the class scope) has full access to any member. We can then define the function wherever we like.
Obviously this behaviour goes against OOP and it’s not recommended.

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

class Car {
    private:
        int speed;
    public:
        friend void makeCarFasta(Car&);
        Car() { speed = 10; }
        int getSpeed() { return speed; }
        void print() {
            cout << "SPEED:" << speed << endl;}
};

void makeCarFasta(Car& c) {
    c.speed*=2; }

int main() {
    Car car;
    car.print(); // OUTPUT: SPEED:10
    makeCarFasta(car);
    car.print(); // OUTPUT: SPEED:20
    return 0;
}
```

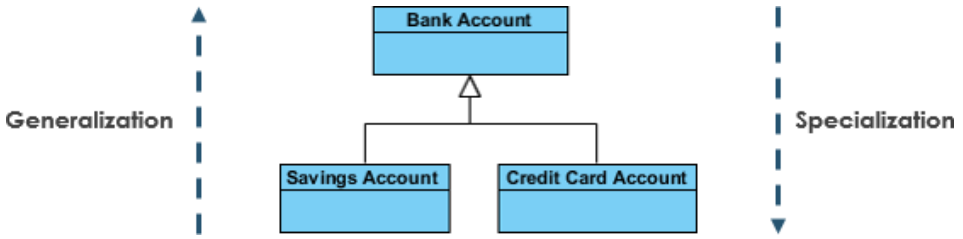
You can also “befriend” two classes

```
#include <iostream>

class Bike {
    int speed = 10;
    public:
        friend class Car;
        void print() {
            std::cout << "bike SPEED:" << speed << std::endl; }
};
```

```
class Car {
    int speed = 10;
    public:
        void destroyBike(Bike & b) {
            b.speed = 0; }
};

int main() {
    Car car;
    Bike bike;
    bike.print(); // OUTPUT: bike SPEED:10
    car.destroyBike(bike);
    bike.print(); // OUTPUT: bike SPEED:0
    return 0;
}
```

// Generalizzazione e specializzazione

Generalization is a mechanism for combining similar classes of objects into a single, more general class. Generalization identifies commonalities among a set of entities. The commonality may be of attributes, behavior, or both. In other words, a superclass has the most general attributes, operations, and relationships that may be shared with subclasses. A subclass may have more specialized attributes and operations.
Specialization is the reverse process of Generalization means creating new sub-classes from an existing class.




```
#ifndef CLASSE_B
#define CLASSE_B
#include<iostream>
using  namespace std;
#include<string>

#include "a.h"

class B:public A{
      string s;
      public:
      B();
      B(int _i,string _s);
      ~B();// non e' necessario
      string get_s();
      };

void test_B();
#endif
```

// Inline

La parola chiave inline si applica alle definizione di funzioni o funzioni membro come forma di ottimizzazione. Essa è una speciale direttiva al compilatore che, se eseguita, consiste nel sostituire la chiamata a funzione con il corpo della funzione stessa.
It makes things go speed. Da usare quando una funzione è breve (in termini di memoria) e richiamata molte volte (per evitare GOTO di assembly)

```
template <typename T>
inline T const& Max (T const& a, T const& b) {
   return a < b ? b:a;
}
```

// Lambda

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

void f5 (const auto &l) { l(); }

int main(int argc, char** argv) {
      int a = 5;
      // LAMBDA: [] lista di cattura, () parametri, {} corpo
      auto l1 = [] () { cout << "lambda #1" << endl; };
      l1();
      auto l2 = [] (int a) -> char {
            cout << "lambda #2 - a:" << a << endl; return 48; };
      cout << "res:" << l2(7) << endl;
      auto l3 = [a] () { cout << "lambda #3 - a:" << a << endl; };
      l3();
      auto l4 = [&a] () mutable {
            a++; cout << "lambda #4 - a:" << a << endl; };
      l4();
      f5([](){ cout << "lambda #5" << endl; });
      return 0; }
```

lambda, è un modo pratico per definire un oggetto funzione anonima (una chiusura) direttamente nella posizione in cui viene richiamato o passato come argomento a una funzione. In genere le lambda vengono usate per incapsulare alcune righe di codice passate agli algoritmi o ai metodi asincroni.

// Left/Right values

```
// lvalue and rvalue
#include <iostream>
using namespace std;

// Driver Code
int main()
{
      int a = 10;
```

```
	// Declaring lvalue reference
	// (i.e variable a)
	int& lref = a;

	// Declaring rvalue reference
	int&& rref = 20;

	// Print the values
	cout << "lref = " << lref << endl;
	cout << "rref = " << rref << endl;

	// Value of both a
	// and lref is changed
	lref = 30;

	// Value of rref is changed
	rref = 40;
	cout << "lref = " << lref << endl;
	cout << "rref = " << rref << endl;

	// This line will generate an error
	// as l-value cannot be assigned
	// to the r-vaue referances
	// int &&ref = a;
	return 0;
}
```

“l-value” refers to a memory location which identifies an object. **“r-value”** refers to the data value that is stored at some address in memory.
An rvalue is any expression that has a value, but cannot have a value assigned to it. One could also say that an rvalue is any expression that is not an lvalue . An example of an rvalue would be a literal constant
int x = 1; // x is an lvalue int
y = 2; // y is an lvalue
int z = x + y; // the "+" needs rvalues, so x and y are converted to rvalues //(from lvalues) and an rvalue is returned
Si può convertire un leftvalue in rightvalue, ma non un rightvalue in un leftvalue (senza assegnazione ad una variabile già dichiarata).

// Map

```
#include <iostream>
#include <map>
#include <string>
#include <list>
using namespace std;

int main(int argc, char** argv) {
	map<string,int> m;
```

```
	list<string> l = {"a", "b", "c", "d", "e"};
	for (auto it = l.begin(); it != l.end(); it++) {
		//m.insert(pair<string,int> (*it, rand() % 10));
		m[*it] = rand() % 10;
	}
	cout << "[MAP] size: " << m.size() << endl;
	cout << "[MAP] keyword \"b\": " << m["b"] << endl;
	m["z"] = 89; // se keyword non esiste allora analogo all'insert
	map<string,int>::iterator miter;
	for(miter=m.begin(); miter!=m.end(); miter++){
		cout << "m[" << miter->first << "]: " << miter->second;
		cout << endl; }
	return 0; }
```

Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order. *(dizionari python)*
In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key. The types of key and mapped value may differ, and are grouped together in member type value_type, which is a pair type combining both:

// Metaprogrammazione a Template

La metaprogrammazione a template consiste nell'uso delle generazione di codice ad-hoc in compilation time dei template per generare costanti/variabili/codice pronto all'uso in run-time, senza togliere tempo a quest'ultimo. Mantra: calcolato una volta, calcolato per sempre.

```
#include <iostream>

template <int n> struct Fact {
    enum { RET = n * Fact<n-1>::RET };
};

template <> struct Fact<1> {
    enum { RET = 1 };
};

int main() {
    std::cout << Fact<12>::RET;
    return 0;
}
```

// Multimap

```
#include <iostream>
#include <map>
#include <iterator>
#include <algorithm>
#include <list>
using namespace std;

map<char,list<int> > generaMap(multimap<char,int> in) {
```

```
      map<char,list<int> > out;
      for (auto it = in.begin(); it != in.end(); it++) {
            out[it->first].push_back(it->second);
      }
      return out;
}

int main()
{
    //Multi Map of char and int
    // Initializing with initializer list
    multimap<char, int> mmapOfPos ={
            {'t', 1}, {'h', 1}, {'i', 2}, {'s', 3},{'i', 5},{'s',
6},{'i', 8},
    };

    mmapOfPos.insert(pair<char, int>('t', 9));// Inserimento, vanno
inseriti usando un elemento "pair"

    for (multimap<char, int>::iterator it = mmapOfPos.begin(); it !=
mmapOfPos.end(); it++)//ciclo con iteratore
        cout << it->first << ": " << it->second << endl;

    cout << "--------------------------------" << endl;


    map<char,list<int> > map2 = generaMap(mmapOfPos);
    for (auto it = map2.begin(); it != map2.end(); it++) {
            cout << it->first << ": ";
            for (auto it2 = it->second.begin(); it2 != it->second.end();
it2++) {
                  cout << *it2 << " ";
            }
            cout << endl;
      }

      for(auto& itemList: map2) //itero e posso mettere il tipo base qui
volendo
      for(auto& item: itemList.second)//obbligato auto&
        cout << itemList.first << " : " << item << endl; //gli oggetti
sono itemList.first e itemList.second

    return 0;
}
```

Multi-map in C++ is an associative container like map. It internally store elements in key value pair. But unlike map which store only unique keys, multimap can have duplicate keys (dictionary in python)

// Multiset

```
cout << "[MULTISET] kinda chilling" << endl;
```

```
multiset<int> ms;
for (int i = 0; i < 20; i++)
	ms.insert(rand() % 10);
cout << "size " << ms.size() << endl;
for (auto it = ms.begin(); it!=ms.end(); it++)
	cout << " " << *it;
cout << endl;
```

// OOP

Link perchè è fatto abbastanza bene e sono intrinsecamente pigro: [C++ Classes and Objects]

// Operatori

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

class Complesso {
	private:
		int r; int i;
	public:
		int getR() const { return r; }
		int getI() const { return i; }
		Complesso() { // DEFAULT
			r = 0; i = 0; }

		Complesso(int _r, int _i) {
			i = _i; r = _r; }

		Complesso(const Complesso& c) // COPIA
		/*:Complesso(c.getR(), c.getI())*/ {
			r = c.getR(); i = c.getI(); }

		Complesso (Complesso&& other) { // SPOSTAMENTO
			// uguale al copia in questo caso. se ci fossero
			// proprietà puntatori copieremmo il solo puntatore,
			// non l'oggetto per intero come il copia dovrebbe già
			// saper fare
			r = c.getR(); i = c.getI(); }

		~Complesso() { // DISTRUTTORE ALT+126
			cout << "[DIST] rip" << endl; };

		Complesso& operator = (const Complesso& c) {
			r = c.getR(); i = c.getI(); }

		Complesso operator + (const Complesso& c) {
			Complesso temp(r + c.getR(), i + c.getI());
```

```
                return temp; }

            Complesso& operator += (const Complesso& c) {
                r += c.getR();
                i += c.getI(); }

            Complesso operator ++ (int) { //postincremento: var++
                Complesso temp = *this; // copia
                r++; i++; return temp; }

            Complesso operator ++ () { //preincremento: ++var
                r++; i++; return *this; } // copia

            bool operator == (const Complesso& c) {
                return r == c.getR() && i == c.getI(); }
            bool operator != (const Complesso& c) {
                return !(*this == c); }
            friend ostream& operator << (ostream& s, const Complesso&
c);
            void stampa() {
                cout << "C:(" << r << "," << i << ")" << endl; }
};

ostream& operator << (ostream& s, const Complesso& c) {
      s << "C:(" << c.getR() << "," << c.getI() << ")"; return s; }

int main(){
      Complesso a(6, 8);
      Complesso b(1, -5);
      Complesso c;
      c = a + b;
      c.stampa();
      bool res = a == b;
      cout << res << endl;
      res = a != b;
      cout << res << endl;
      cout << a << endl;
      return 0; }
```

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

// Puntatori

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

// &var nel metodo che riceve, niente in quello che invia-> passa il
riferimento in memoria
// func(int* var){return *var+1} e func(var)
// & -> indirizzo in memoria
// * -> valore in memoria
```

```
void stampaNormale(int x){
    cout <<"[STAMPA NORMALE INDIRIZZO IN MEMORIA] "<< &x<<endl;
    cout <<"[STAMPA NORMALE VALORE IN MEMORIA] "<< x<<endl; }
void stampaPuntatore(int * x){//dereferenzia quindi guarda il valore
all'interno del valore in memoria e se lo salva nel proprio indirizzo in
memoria
  cout << endl <<"[STAMPA PUNTATORE INDIRIZZO MEMORIA] "<< &x<<endl;
  cout <<"[STAMPA PUNTATORE VALORE IN MEMORIA] "<< x<<endl; }

void stampaRiferimento(int &x){
  cout << endl <<"[STAMPA RIFERIMENTO INDIRIZZO MEMORIA] "<< &x<<endl;
  cout <<"[STAMPA RIFERIMENTO VALORE IN MEMORIA] "<< x<<endl;}

int main(int argc, char** argv) {
  int var = 2;//classica
  stampaNormale(var);
  int* var2;//crea un puntatore
  var2 = &var;//il valore di ptr è un indirizzo e va assegnato con &
  stampaPuntatore(var2);//senza & perchè passa in automatico il valore e
quindi un indirizzo in memoria
  //e ricevendolo con * quindi dereferenziandolo guarda il valore in
quell'indirizzo

  int var3 = *var2;//var3 prende il valore che sta effettivamente
nell'indirizzo in memoria di var2
  stampaRiferimento(var3);

  return 0;
}
```

A pointer however, is a variable that stores the memory address as its value.
A pointer variable points to a data type (like int or string) of the same type, and is created with the * operator. The address of the variable you're working with is assigned to the pointer

// Puntatori a funzioni

```
#include <iostream>
using namespace std;
void stampa(int val) {
      cout << "val:" << val << endl;}
int main(int argc, char** argv) {
      // we need (*name) instead of *name, otherwise we would
      // apply operator * on the return value
      void (*funzione)(int) = stampa;
      funzione(5);     // those two are...
      (*funzione)(5);  // ...equivalent
      return 0;}
```

A function pointer is a variable that stores the address of a function that can later be called through that function pointer. This is useful because functions encapsulate behavior. For instance, every time you need a particular behavior such as drawing a line, instead of writing out a bunch of code, all you need to do is call the function.

Benefits of Function Pointers:

- Function pointers provide a way of passing around instructions for how to do something (similar to lambda functions)
- You can write flexible functions and libraries that allow the programmer to choose behavior by passing function pointers as arguments (like a callback function)
- This flexibility can also be achieved by using classes with virtual function

Lambda vs Function Pointers:

- Lambdas are implemented through a class, created by the compiler
- Lambdas are suitable for single-scope applications
- F. p. may have problems with multithreading, by being globally accessible

// Set

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

int main(int argc, char** args) {
      set<int> s;
      //set<int>::iterator it;
      //set<int>::reverse_iterator rit;
      for (int i = 0; i < 20; i++) s.insert(rand() % 10 + 1);
      for (auto it = s.begin(); it!=s.end(); it++)
            cout << " " << *it;
      set<int>::reverse_iterator rit;
      for (rit = s.rbegin(); rit!=s.rend(); rit++)
            cout << " " << *rit;
      auto lambda = [](int val) { cout << " " << val; };
      for_each(s.begin(), s.end(), lambda);
      return 0; }
```

Sets are containers that store unique elements following a specific order.
In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.
Internally, the elements in a set are always sorted following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).
Sets are typically implemented as binary search trees.
I multiset permettono l'inserimento di elementi con valori duplicati. Ammettono quindi valori ripetuti.
Gli unordered set non mantengono l'ordine degli elementi, conservando comunque la loro unicità.
Esistono anche gli unordered multiset, anche se la sola idea di tale abominio sembri capovolgere l'equilibrio del tessuto della realtà.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
```

```
    set<int> s;
    s.insert(50);
      s.insert(60);
    auto pos = s.find(0);
      // ritornerà 2, la dimensione vera del vettore e della pos di
vector.end()
      int pos2 = *s.find(50);
      // ritornerà 50, la dimensione vera del vettore e della pos di
vector.end()
    cout << pos2<< endl;
      return 0;
}
```

// Smart pointers

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

class Patata {
      public:
            Patata() { cout << "new patata instance created" << endl; }
            ~Patata() { cout << "patata instance is gone" << endl; } };

int main(int argc, char** argv) {
      // unique_ptr
      // - one owner of the underlying pointer
      // - it is destroyed when going out of scope
      // shared_ptr
      // - multiple owners of the underlying pointer.
      // - e: you ret. copy of ptr from a container but you want to
      //   keep the original.
      // - shared_ptr destr. when all the owners are out of their scopes
      // weak_ptr
      // - allows to ref. to a shared_ptr without being a proper owner
      Patata *ptr = new Patata();
      cout << "using normal pointer..." << endl;
      delete ptr;
      unique_ptr<Patata> sptr (new Patata());
      cout << "using smart pointer..." << endl;
      return 0; } // we are out of scope here
```

L'obiettivo principale di questo linguaggio è assicurare che l'acquisizione delle risorse avvenga contemporaneamente all'inizializzazione dell'oggetto, in modo che tutte le risorse per l'oggetto vengano create e rese disponibili in una riga di codice
So in C++ 11, it introduces smart pointers that automatically manage memory and they will deallocate the object when they are not in use when the pointer is going out of scope automatically it’ll deallocate the memory

// Template

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

template <typename T>T massimo(const T x, const T y){
    cout << "Tipo: " << typeid(x).name() << endl; // i, f, Ss
    if (x>y) {return x;}
    return y; }
int main(int argc, char** argv) {
      int a = 10, b = 20;
      //x: 10, y: 20, max:20 Tipo: i
      float c = 10.7, d = 11.3;
      //x: 10.7, y: 11.3, max:11.3 Tipo: f
      string e = "AAA", f = "aaa";
      //x: AAA, y: aaa, max:AAA Tipo: Ss
      return 0; }
```

Function 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.
When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType 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.

// Threads

```
#include <iostream>
#include <thread>
#include <unistd.h>
#include <string>
#include <mutex>
#include <atomic>
using namespace std;

mutex mtx;
atomic<int> atomicSum(0); //init a 0
int normalSum = 0;
class Albero {
      private: string tipo;
      public: Albero(string _tipo) { tipo = _tipo; }

      public: void stampa() {
            cout << "[TASK 4] albero: " << tipo << endl; }

      public: static void stampaS(string tipo) {
            cout << "[TASK 3] albero:" << tipo << endl; } };

void task89(int d) {
      for(int i = 0; i < 1000000; i++) {
```

```
            atomicSum += d;
            normalSum += d; } }

void task67(char c, int n) {
      mtx.lock();
      for (int i = 0; i < n; i++) {
            cout << c;
            sleep(1); }
      cout << endl;
      mtx.unlock(); }

void task1 () {
      cout << "[TASK 1] started" << endl;
      sleep(2); cout << "[TASK 1] finished" << endl; }

void task2 (int time) {
      cout << "[TASK 2] started" << endl;
      sleep(time); cout << "[TASK 2] finished" << endl; }

int main(int argc, char** argv) {
      // intro
      thread thread1(task1);
      thread thread2(task2, 4);
      thread1.join();
      thread2.join();
      // metodi e metodi statici
      thread thread3(&Albero::stampaS, "quercia");
      Albero alberello("abetino");
      thread thread4(&Albero::stampa, alberello);
      thread3.join();
      thread4.join();
      // lambda
      auto lamba = [](int val) {
            cout << "[TASK 5] started" << endl;
            cout << val << endl; cout << "[TASK 5] finished" << endl; };
      thread thread5(lamba, 5);
      thread5.join();
      // mutex
      thread thread6(task67, 'A', 5);
      thread thread7(task67, 'B', 5);
      thread6.join();
      thread7.join();
      // atomic
      thread thread8(task89, 3);
      thread thread9(task89, 2);
      thread8.join();
      thread9.join();
      cout << "atomic:" << atomicSum << endl;
      cout << "normal:" << normalSum << endl; // probably lower
      return 0;
}
```

A thread of execution is a sequence of instructions that can be executed concurrently with other such sequences in multithreading environments, while sharing the same address space.
An initialized thread object represents an active thread of execution; Such a thread object is joinable, and has a unique thread id.
Uno dei vantaggi dei thread rispetto ai processi è che condividono lo spazio di memoria tra di loro, tutto quello che è nello spazio di indirizzamento del processo padre può essere passato ai figli.

// Virtual

“Il modificatore virtual altera una funzione membro permettendo il late-binding (o dynamic linkage), che consiste nella risoluzione dinamica di una funzione, dando priorità alla funzione ridefinita dall’oggetto in uso (right value) piuttosto che il tipo dell’invocazione (left value).”
Alla buona: se abbiamo un right value derivato e un left value base, il metodo invocato con virtual sarà il membro definito in override implicito dalla classe derivata.
Alla più buona: se la virtual non è implementata (virtual void metodo() = 0;) l’intera classe viene vista come abstract quindi non puoi istanziare oggetti di quella classe e il virtual deve essere overridato, se invece è implementata la classe non diventa strana e l’override diventa facoltativo. Per il metodo chiamato funziona come c# tranne per i puntatori sghembi che sono nell’esempio.

```
#include <iostream>
using namespace std;
class base {
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }
};

class derived : public base {
public:
    void print()
    {
        cout << "print derived class" << endl;
    }
};

int main()
{
    derived a;
    a.print(); //derived class
    base* bptr;
    bptr = &a;

    // virtual function, binded at runtime
    bptr->print();  //base class
}
```