Funtori

L'oggetto che lo definisce è chiamato anche FUNTORE
- "Oggetto che si usa come una funzione"
- Estremamente utile in programmazione generica

Esempio:

```
struct compare_int {
    bool operator()(int a, int b) const {
        return a<b;
    }
};
```

Tipi di dato

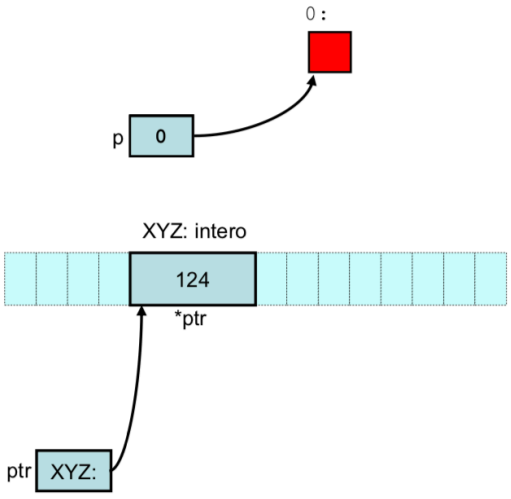
Built-in semplici:
- bool
- char
- int
- float
- double
- Puntatore
- Reference

I tipi char e int possono essere modificati con il termine unsigned che gli elimina il segno.
unsigned int ui;
unsigned char c=-1;
I tipi int e double possono essere estesi (per precisione) con il termine long.
long int li;
long double li;
Il tipo int può essere ridotto di precisione con il termine short.
short int si;
Range e dimensioni dipendono dalla piattaforma HW/SW.
Nello standard C++ sono comunque definiti dei vincoli
- Char è la più piccola entità indirizzabile della piattaforma HW/SW
  - È garantito avere dimensione di almeno 8 bit.
- Int è garantito avere dimensione di almeno 16 bit
- Long è garantito avere dimensione di almeno 32 bit

Il file <limits> contiene le caratteristiche dei dati riferiti alla piattaforma HW/SW utilizzata.
sizeof(), ritorna la dimensione di un tipo (o variabile) come multiplo della dimensione di un char
Esempio
Sizeof(char) = 1
Sizeof(int) = 4
Sizeof(double) = 8
Puntatore
Riferimento ad un indirizzo (accesso indiretto ad un dato)
Contiene l'indirizzo dove trovare un dato di un certo tipo
*Tipo *nome_variabile*

```
tipo *nome_variabile;
```

```
1  int *p;   // senza inizializzazione
2  int *p = 0; // o =NULL definito in <stdlib.h>

3  int intero = 123;
4  int *ptr = &intero; // &var = indirizzo di var

5  int v = *ptr; // dereferenziamento ( v=123 )
6  (*ptr)++; // intero=124;
```



Struct

Raggrupa diversi tipi

struct nome_tipo {
tipo1 nome1;
…
tipoN nomeN;
};

```
struct S
{
 int i;
 double d;
 char carray[10];
};


S s1={10,1.34,"PIPPO"};

S s2[2]={
          { 1,    3.14,    "PLUTO"},  // Completa
          { -986, 1.112345}           // Parziale
        };
```

Enum
Elenco di valori (int) riferibili con un nome
Enum nome {nome1=valore1, …, nomeN=valoreN};
Praticamente delle costanti intere con tipo

```
enum giorno {lun=1, mar=3, mer=2, gio=7, ven=9, sab=90, dom=13};

giorno g;
g = 90;              // Errore: Invalid conversion from int to giorni
g = sab;             // Ok
int d = lun;         // Ok

enum giorno {lun=10, mar, mer, gio, ven, sab, dom};
// mar=11, mer=12, gio=13, etc...

enum giorno {lun, mar, mer, gio, ven, sab, dom};
// lun=X, mar=X+1, mer=X+2, gio=X+3, ven=X+4, sab=X+5, dom=X+6
```

Typedef

La keyword typedef serve per definire nuovi nomi di tipi

Typedef vecchio_tipo nuovo_tipo

```
1  typdef unsigned long int uli;

2  typedef uli *ptr_uli;

3  typedef double darray20[20];

4  uli i;
   ptr_uli p = &i;

5  darray20 d1, d2;
```

Moficatore const

Il dato è costante

const tipo var = valore;

- Immodficabilità.
- Necessaria inizializzazione.
  - È sempre possibile inizializzare una variabile const con una variabile dello stesso tipo non const

```
1  const int ci;            // Errore

2  const int ci=10;         // Ok

3  ci=0;                    // Errore

   int i=0;
4  const int &rci=i;        // Ok

5  rci++;                   // Errore
```

const int *p;

```
int i = 200;
const int *p = &i;      // puntatore a intero costante

*p = 100;               // Errore
 p = 0;                 // Ok
```

int *const p;

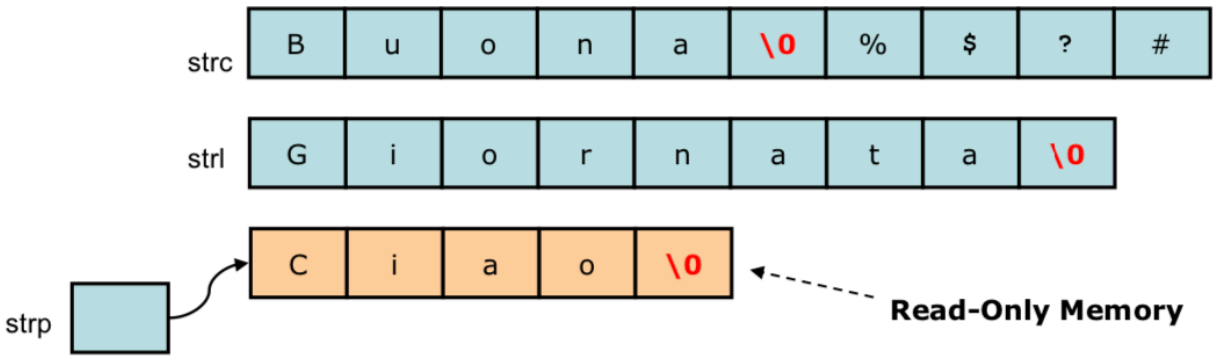
```
int i = 200;
int * const p = &i;     // costante e puntatore a intero

*p = 100;               // Ok
 p = 0;                 // Errore
```

const int *const p;

```
int i = 200;
const int *const p = &i;  // costante e puntatore a intero costante

*p = 100;                 // Errore
 p = 0;                   // Errore
```

Stringhe C
- Un array di caratteri terminato da '\0'
- Literal strings ("Hello World")
- Ha un solo operatore ([])
- Metodi specifici definiti in <cstring>
- Di solito gestite tramite puntatori a char

```
char strc[10]="Buona"; // Inizializzazione dell'array con copia

char strl[]={'G','i','o','r','n','a','t','a','\0'};

char *strp="Ciao";  // Attenzione! La stringa è costante!
                    // Meglio: const char *strp="Ciao";
```



Stringhe C++
- Oggetti(classe string)
- Rappresentazione della stringa più complessa
- Hanno diversi operatori
- Conversione c-string <-> c++-string
- Definite in <string>

```
#include <string>

std::string s1;

std::string s2="pippo";

s1="pluto";

s1[0]='B';

std::cout << s1 << std::endl;  // stampa : Bluto

std::cout << s1.c_str() << std::endl;  // stampa : Bluto
```

Funzioni con parametri opzionali
(Valori di default nel passaggio parametri alle funzioni)

Possiamo assegnare un valore di default ad un parametro di una funzione.
Questo valore viene assegnato nel caso quel parametro non venga passato.
Dopo un parametro opzione non possono esserci parametri “normali” solo altri parametri opzionali.

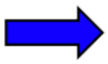
```
// Valori dei parametri di default

void h(char c, int v=20);     // Ok

void f(char c=10, int v=90);  // Ok

void g(int v=90, char c);     // Errore


f(1, 29);

f(1);    // esegue f(1,90)

f();     // esegue f(10,90)
```

Funzioni inline

Suggeriscono al compilatore di sostituire la chiamata con direttamente il codice.

file.cpp

```
inline int massimo(int a, int b)
{
 return (a>b)?a:b;
}

int main()
{
 int a,b,c,d,r;

 r = massimo(a,b) + massimo(c,d);
}
```



```
int main()
{
 int a,b,c,d,r;

 r = (a>b)?a:b + (c>d)?c:d;
}
```

Allocazione Dinamica

I dati dinamici vengo allocati e deallocati a richiesta nello Heap.
Per allocare si utilizza l'istruzione new, per deallocare si utilizza l'istruzione delete.
Questo dati vivono finché non vengono esplicitamente deallocati (**no garbage collection**).
Essi vengo gestiti tramiti i puntatori

```
1  struct Obj {
     double d;
     int arr[1024];
   };

2  Obj *o = new Obj; // crea un obj sullo heap

3  delete o;         // Ok, libera la memoria

4  delete o;         // Il delete di un oggetto già
                     // deallocato è pericoloso

5  o = NULL;         // Da fare sempre per
                     // sicurezza dopo un delete!

6  delete o;         // Ok fare il delete di un
                     // puntatore a NULL
```

Initialization List

- La semantica dell'initialization list coincide con la creazione dei dati membro attraverso chiamate a certi costruttori
  - In generale più efficiente che l'inizializzazione attraverso assegnamento

```
CDate::CDate(void)
{
 // I dati membro sono già stati creati con eventualmente dei valori di default
 mDay=0;
 mMonth=0;
 mYear=0;
 // Si può avere doppia inizializzazione dei dati (default + assegnamento)
}
```

```
CDate::CDate(void) : mDay(0), mMonth(0), mYear(0)
{
 // I dati membro sono creati con i valori indicati
 // Si ha un sola sola inizializzazione dei dati
  ...
}
```

Constness

Non tutte le funzioni membro di una classe possono o devono essere usate su oggetti costanti.

Necessario un meccanismo di controllo d'uso delle funzioni.

Consness significa Const Correttezza.

Definire un oggetto const significa che lo stato dell'oggetto non deve cambiare.

Const impone dei vincoli su quali metodi possono essere chiamati

- I metodi "modificanti" devono essere bloccati
- Solo i metodi "non modificanti devono essere permessi

```
class Punto2D{
 int x,y;
 ...
public:
 ...
 int getX(void) const {return x;}
 void setX(int pX){x=pX;}
 ...
};
```

Metodo Non Modificante

Metodo Modificante

Se si vuole che una funzione membro sia chiamabile su un un oggetto const la funzione DEVE essere dichiarata const promessa di non modificabilità dello stato dell'oggetto)

In una funzione const tutte le variabili membro della classe sono viste read-only e non possono essere modificate.

Una funzione const può usare SOLO funzioniconst così non si rompe il contratto di modificabilità.

I costruttore sono sempre usabili

Di un oggetto const si possono usare solo i metodi const, di un oggetto non const si possono usare tutti i metodi.

Copy Constructor

Viene chiamato ogni volta che bisogna creare un oggetto da un altro dello stesso tipo.

Se non esiste, ne viene sempre creato uno automaticamente dal compilatore:

- Per le variabili built-in copia bit a bit
- Richiama ricorsivamente i costruttori di copia delle variabili membro

Fondamentale quando l'oggetto alloca dati sullo heap, controllo della copia effettiva dei dati puntati (non il puntatore).

test(const test& pT);

Operatore di assegnamento

L'operatore di assegnamento tra dati dello stesso tipo DEVE essere definito membro della classe.

Se non viene definito ne viene creato uno automaticamente che applica memberwise assignment ai dati membro e può avere problemi con i puntatori.

Dentro l'operatore controllare sempre il self-assignment altrimenti rischio di perdita delle risorse se queste devono essere rilasciate prima di essere riassegnate.

Complex &operator= (cons Complex &Y)

Differenza tra copy constructor e operatore di assegnamento

Il differenza principale tra il costruttore di copie e l'operatore di assegnazione è che quello copy constructor è un tipo di costruttore che aiuta a creare una copia di un oggetto già esistente senza influenzare i valori dell'oggetto originale mentre l'operatore di assegnazione è un operatore che aiuta ad assegnare un nuovo valore a una variabile nel programma.

Explicit

Costruttori con un solo parametro possono essere utilizzati come: dato un oggetto di tipo T1 (quello del parametro) costruire un oggetto di tipo T2 quello della classe.
Se non si vuole questa semantica bisogna premettere la keyword **explicit** a questi costruttori, così facendo il compilatore non li utilizzerà per le conversioni di tipo.
Explicit objB(const objA &o) {//…}

Friend

Si utilizza per dare accessibilità agli elementi interni data selettivamente a funzione e *classi.*

```
class Y {
        friend class Z; // Accesso consentito ad una classe
        friend void X::g(Y &py); // Accesso consentito ad una funzione di una classe
...
};
```

Una funzione così dichiarata non è propriamente un membro di classe, ma ha il privilegio di accedere a tutti i membri di classe (privati, protetti e pubblici) come se lo fosse.

Template

I template permettono di realizzare codice generico, applicabile a diversi tipi di dati.

STL è realizzata completamente con template (<algorithm> algoritmi generici)

Il codice template (generico) è un prototipo di codice del quale non esiste codice compilato finché non viene usato.

Il codice template è messo nel file .h perché il compilatore deve conoscere come è fatto **TUTTO** il template come nel caso delle funzioni inline.

Quando incontra una richiesta d'uso su un certo tipo di dati crea "al volo" il codice relativo ed esiste una copia del codice per ogni tipo di dati su cui viene usato.

Si possono avere funzioni e classi template.

Definizione:

```
template <typename T> class genericClass{
T value;
public:
T getValue(){return value;}
};
```

Utilizzo

```
genericClass<int> I;
int i=I.getValue();
```

Ogni volta che si fa riferimento alla classe templata all'interno delle funzioni membro, questa viene automaticamente vista con gli stessi parametri template anche se non scritti.
Ogni classe templata definita su un parametro template diverso è un tipo di dato diverso
Es: *vector<int>* **E' UN TIPO DIVERSO** da *vector<long>*
Un parametro template di una classe può avere un valore di default, nell'uso è possibile non specificare il parametro template utilizzando quindi quello di default.
Attualmente non tutti i compilatori supportano i parametri di default per le funzioni (previsto dallo standard).
Si possono avere più template, come per i parametri delle funzioni bisogna partire a inserire i parametri template default da destra.

Typename

Ogni volta che ci si riferisce ad un tipo di dato definito all'interno di una classe templata non ancora specificata è necessario usare la keyword typename.

```
template<class T> void f(const pippo<T> &p)
{
pippo<T>::inner_type value; // Errore
typename pippo<T>::inner_type value; // Ok
...
}
```

La classe templata specificata non esiste ancora (T ignoto) quindi il compilatore non sa riconoscere se inner_type è un dato membro della classe o un tipo definito all'interno della classe.
Con typename diciamo al compilatore che ci stiamo riferendoci ad un tipo di dato.
Se la classe viene specificata non è necessario typename.

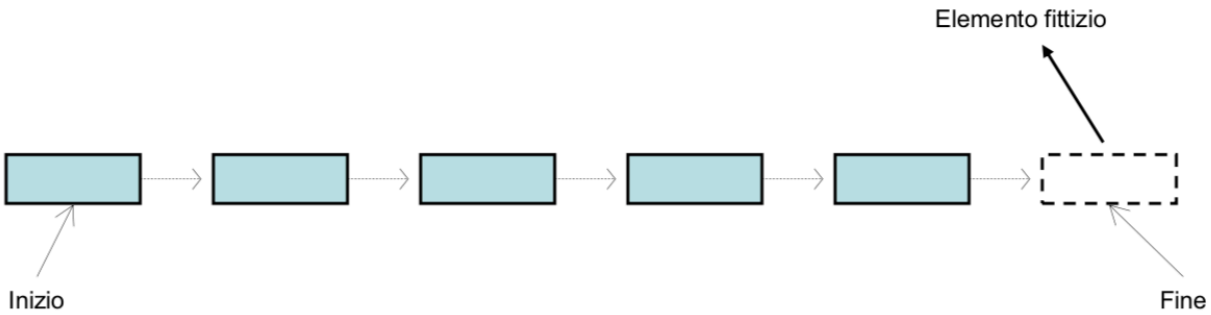
```
pippo<int>::inner_type; // Ok
```

Preprocessore

In questa lezione dobbiamo parlare di un elemento importante nella **programmazione in C**, il **pre-processore**. Questo strano elemento ha un ruolo apparentemente trasparente, ma allo stesso tempo importantissimo. Il pre-processore è incaricato, fondamentalmente, di trovare delle **direttive** all'interno di un file sorgente e di eseguirle; dove sta l'arcano? che il pre-processore opera su un file sorgente e "restituisce un file sorgente" (che poi, a questo punto, viene passato al compilatore), perché le direttive che lui interpreta sono principalmente di **inclusione**, di **definizione** e **condizionali**.

Iteratori

Una qualunque successione di valori ha un inizio ed una fine.
Queste proprietà devono valere indipendentemente dalla sua implementazione effettiva, il container che la rappresenta (es. vector) deve poter indicare l'inizio della sequenza.
Per accedere alla sequenza la libreria standard del C++ utilizza il concetto di iteratore.

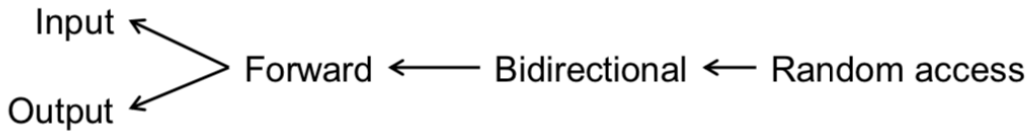


Un iteratore è una astrazione ed ha una semantica e sintassi simile ad un puntatore, esso permette di accedere ai dati utilizzando gli operatori tipici di un puntatore:

- *operator**
- *operator->*
- *operator[]*
- *operator++*
- *…*

Usando una sintassi (del puntatore) unica per accedere ai dati di una sequenza, si ha disaccoppiamento tra effettiva implementazione e uso, ogni container dovrebbe supportarli affinché sia usabile con gli algoritmi generici della libreria standard.
Nella STL sono previste 5 categorie specifiche di iterativi.
Gli iteratori **NON SONO** dei puntatori, non esiste elemento nullo e possono consentire solo certe operazioni.

Iterator Operations and Categories					
Op\Categoria	Output	Input	Forward	Bidirectional	Random access
Read		*i, i->	*i, i->	*i, i->	*i, i-> [ ]
Write	*i, i->		*i, i->	*i, i->	*i, i-> [ ]
Iteration	++	++	++	++ --	++ -- + - += -=
Comparison		== !=	== !=	== !=	== != < > <= >=



I puntatori sono in sostanza dei random access iterator.
Gli iteratori dovrebbero essere utilizzati da qualunque oggetto che rappresenta un qualche tipo di container.
Una classe che vuole adottare il concetto di iteratori deve esporre dei costrutti per recuperare l'iteratore da usarsi per accedere ai dati.
Deve anche esporre dei metodi per identificare l'inizio e la fine della sequenza.
Bisogna stare attenti che se la classe è usata const, l'iteratore non deve permettere la modifica dei dati puntati.
L'STL ha definito un design preciso per gli iteratori e le classi che li usano.
Ogni classe che include gli iteratori deve contenere:

- **Un tipo membro iterator**
  - L'iteratore da usarsi in lettura/scrittura
- **Un tipo membro const_iterator**
  - L'iteratore da usarsi sull'oggetto const(non modifica i dati puntati)
- **Una funzione membro begin()**
  - Ritorna un iteratore che punta all'inizio della sequenza di dati
- **Una funzione membro end()**
  - Ritorna un iteratore che punta alla fine della sequenza di dati

Le funzionalità di iterator e const_iterator dipendono dalla categoria scelta.
La categoria dipende dalla classe che li deve implementare

- Struttura dati interna
- Funzionalità

L’accesso ai dati tramite iteratori deve essere coerente con quello attraverso i metodi della classe container.
Gli algoritmi generici hanno spesso la necessità di avere informazioni sul tipo di dato che stanno trattando.
La STL usa il concetto di traits, associare ad un tipo di dato informazioni relative alle sue proprietà o funzionalità che si possono sfruttare negli algoritmi.
Un algoritmo può usare/interrogare i traits per eseguire (o meno) delle operazioni.
Necessario includere i traits negli iteratori, le informazioni che deve contenere e definire sono:

- **iterator_category**
  - Un tag corrispondente ad una delle 5 categorie (definiti in <iterators>)
- **value_type**
  - Definisce il tipo “puntato” dall’iteratore
- **reference**
  - Definisce il reference al tipo “puntato” dall’iteratore
- **pointer**
  - Definisce il puntatore al tipo “puntato” dall’iteratore
- **difference_type**
  - Definisce il tipo usato per rappresentare la differenza tra due puntatori (di solito si usa il tipo predefinito ptrdiff_t)

I traits possono anche essere automaticamente creati derivano i nostri iteratori da una classe base templata iterator.
NOTA: nel caso l’iteratore sia un alias di un puntatore, i traits sono derivati automaticamente dal compilatore attraverso le classi template.
Gli iteratori danno accesso ai dati secondo una certa policy, prestare attenzione all’efficienza.
Sono il collante tra codice utente e libreria standard.
Regola d’uso:

- Un iteratore rimane valido (punta a qualcosa di sensato) finché la sequenza di dati non viene modificata.
- Se la sequenza viene modificata gli iteratori precedentemente usati non sono (in generale) più validi!
  - ES: rimozione di elementi della sequenza
  - Necessario recuperarli di nuovo tramite begin() e end()

Gli iteratori collegati ad una classe container servono per l’accesso ai dati, essi possono anche avere vita propria, tipicamente sono generatori di sequenze di dati.

- Generatori di numeri random
- Generatori di valori staticamente descrivibili
- Tokenizers
- …

Composizione

Esempio di composizione

```
class Address { //... };
class PhoneNumber { //... };
class Person {
public:
        ...
private:
        std::string name; Address address;
        PhoneNumber voiceNumber;
        PhoneNumber faxNumber;
};
```

È una relazione “has-a” “is-implemented-in-term-of”

Ereditarietà

Esempio ereditarietà

```
class people{
        //...
};
class student : public people {
        //…
};
```

È una relazione “is-a”, “is-like-a”.

La classe derivata può essere usata (idealmente) ovunque è usata la classe base.

```
class Base{
  void f1() {};
protected:
  void f2() {};
public:
  void f3() {};
};

class Derivata : public Base {
public:
  void f(){
    f1(); // Errore!
    f2(); // Ok
    f3(); // Ok
  }
};

Derivata d;
Base b;
d.f2(); // Errore!
d.f3(); // Ok
```

```
class Base{
  void f1() {};
protected:
  void f2() {};
public:
  void f3() {};
};

class Derivata : private Base {
public:
  void f(){
    f1(); // Errore!
    f2(); // Ok
    f3(); // Ok
  }
};

Derivata d;
1 d.f2(); // Errore!
2 d.f3(); // Errore!
```

A che serve ereditare una classe private?

Solo la classe derivata può usare l’interfaccia di quella Base

Meglio usare Composition!

NOTA: ereditare con protected ha un comportamento simile.

Le funzioni della classe base overloadate o ridefinite, non sono richiamabili direttamente sulla classe derivata, devono essere chiamate con il nome della classe base.

```
class Base {
public:
 void g() {cout<<"Base::g()"<<endl;}
 void g(int, int) {cout<<"Base::g(int,int)"<<endl;}
};

class Derivata : public Base {
public:
 void g(int) {cout<<"Derivata::g(int)"<<endl;}
};

Derivata d;

d.g(9);    // stampa "Derivata::g(int)"

d.g();     // Errore "no matching function"
d.g(1,2); // Errore "no matching function"

d.Base::g(1,2); // stampa "Base::g(int,int)
```

Regole generali:
Quando viene chiamato il costruttore di una classe derivata

- La parte base viene costruita per prima
- Dopo vengono costruiti i dati membro della classe derivata
- Per ultimo viene eseguito il costruttore della classe derivata

Se non specificato altrimenti, il compilatore chiama automaticamente i costruttori di default della classe base e degli oggetti membro.

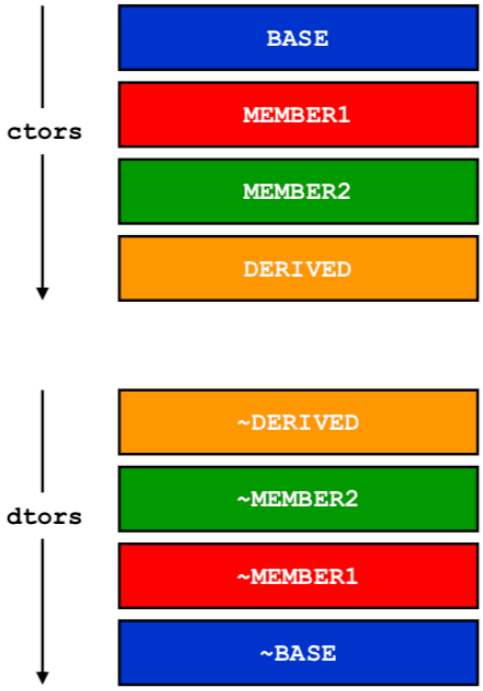
- Quelli delle classi base vengono chiamati in cascata a partire da quello più lontano

I costruttori secondari della classe base o degli oggetti membro devono essere chiamati esplicitamente usando l'initialization list.

```
class Member {
public:
 Member(void) {cout<<"Member::Member ()"<<endl;}
 Member(int) {cout<<"Member::Member (int)"<<endl;}
};

class Base {
public:
 Base(void) {cout<<"Base::Base()"<<endl;}
 Base(int) {cout<<"Base::Base(int)"<<endl;}
};

class Derived : public Base {
 Member m;
public:
 Derived(int) {cout<<"Derived::Derived(int)<<endl;}
 Derived(int,int) : Base(123), m(0) {cout<<"Derived::Derived(int,int)"<<endl;}
};

Derived d(1);       // Stampa Base::Base()
                    // Member::Member()
                    // Derivata::Derivata(int)

Derived d(1,1);     // Stampa Base::Base(int)
                    // Member::Member(int)
                    // Derivata::Derivata(int,int)
```

I distruttori sono chiamati **SEMPRE AUTOMATICAMENTE** in ordine inverso rispetto ai costruttori:
- Distruttore corrente (classe derivata)
- Distruttore degli oggetti membro
- Distruttori della classe base a partire da quella più “vicina”

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

class Base {
public:
 Base(){cout<<"Base"<<endl;}
~Base(){cout<<"~Base"<<endl;}
};

class Member1 {
public:
 Member1(){cout<<"Member1"<<endl;}
~Member1(){cout<<"~Member1"<<endl;}
};

class Member2 {
public:
 Member2(){cout<<"Member2"<<endl;}
~Member2(){cout<<"~Member2"<<endl;}
};

class Derived : public Base {
  Member1 m1;
  Member2 m2;
public:
  Derived(void){cout<<"Derived"<<endl;}
  ~Derived(void){cout<<"~Derived"<<endl;}
};

int main()
{
  Derived d;
}
```



Il **Copy constructor (cctor)** se non definito nella classe derivata viene automaticamente definito e contiene:
- Una chiamata automatica al cctor della classe base
- Chiamate automatiche ai cctor dei dati membro della derivata

Se invece viene definito nella classe derivata sono necessarie chiamate esplicite, nell’initialization list, ai vari cctor della classe base e oggetti membro, altrimenti vengono usati i ctor di default.

```
class Base {
public:
  Base(const Base& other){...}
};

class Member {
public:
  Member(){}
  Member(const Member& other){...}
};

class Derived : public Base {
  Member m;
public:

  Derived(const Derived& other) : Base(other), m(other.m) {...}
};
```

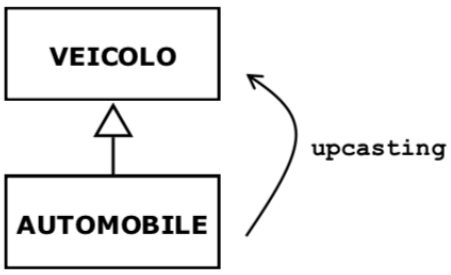
L'Operatore di assegnamento (op=) se non definito nella classe derivato viene automaticamente definito e contiene:
- Chiamata all'op= della classe base
- Chiamate agli op= dei dati membro della classe derivata

Se invece viene definito è necessario gestire esplicitamente gli assegnamenti.

```
class Base {
public:
 Base &operator=(const Base& other) {}
};

class Member {
public:
 Member &operator=(const Member& other) {}
};

class Derived : public Base {
 Member  m;
public:
 ...
 Derived &operator=(const Derived& other)
 {
   if (this != &other) {
     Base::operator=(other);
     m=other.m;
   }
   return *this;
 }
};
```

Upcasting
Derived is-a Base

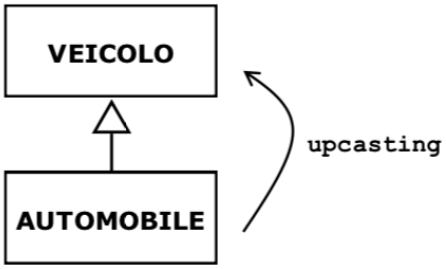
L'upcasting è un passaggio da un tipo specifico a un tipo generale, così da usare un oggetto derivato come fosse quello base.




```
class Veicolo {
protected:
  std::string ID;
  std::string Proprietario;
public:
  std::string& getID();
  std::string& getProprietario();
  //...
};

class Automobile : public Veicolo {
// caratteristiche proprie
public:
//...
};

void Registra(Veicolo &v){//...}

Automobile a;
Registra(a);

Veicolo *b=&a;
```

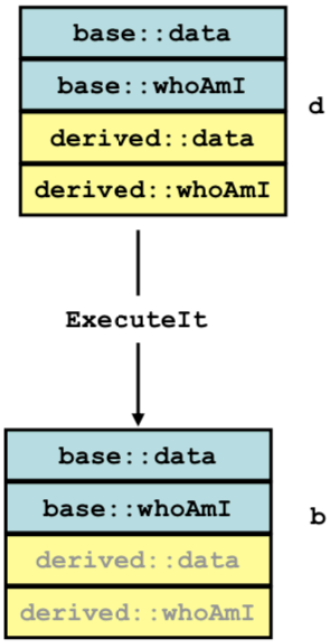
L'upcasting è fondamentale per il polimorfismo ecco alcuni esempio di upcasting per reference/ pointer.

```
Base *b=&derivata;
Base *b=derivata_ptr;

Base &b=derivata;
```

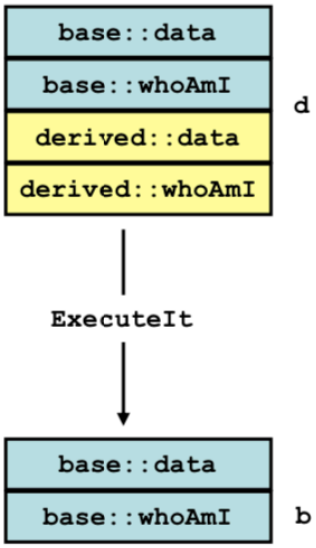


- In b si usano solo le informazioni della parte "base"
- Le informazioni proprie della parte "derivata" esistono ma sono nascoste

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

class base
{
  int data;
public:
  void whoAmI(){cout<<"base"<<endl;}
};

class derived : public base
{
  int data
public:
  void whoAmI(){cout"derived"<<endl;}
};

void ExecuteIt(base &b) {
  // uso b
}

int main()
{
  derived d;
  ExecuteIt(d);
}
```



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

class base
{
 int data;
public:
 void whoAmI(){cout<<"base"<<endl;}
};

class derived : public base
{
 int data
public:
 void whoAmI(){cout"derived"<<endl;}
};

void ExecuteIt(base b) {
 // uso b
}

int main()
{
 derived d;
 ExecuteIt(d);
}
```



Si ha slicing!!

Il **downcasting** è il passaggio da un tipo generale a un tipo specifico, fare upcasting è sempre safe mentre invece fare downcasting è quasi sempre unsafe, a meno che la classe non sia polimorfa.

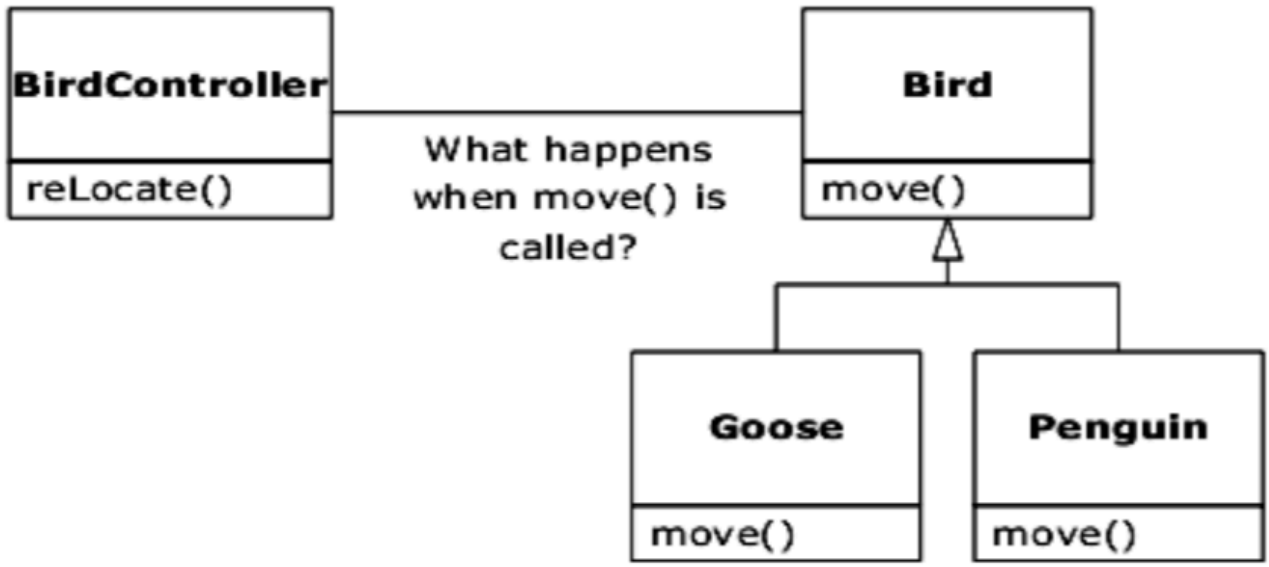
```
#include <iostream>
using namespace std;

class Instrument {
public:
 Instrument(void){}
 void play(){cout<<"Instrument::play()"<<endl;}
};


class Harp : public Instrument
{
public:
 Harp(void){}
 void play(){cout<<"Harp::play()"<<endl;}
};

class Violin : public Instrument
{
public:
  Violin(void){}
  void play(){cout<<"Violin::play()"<<endl;}
};
```

```
void play_instrument(Instrument &b){
 ...
 b.play();
 ...
}

void main(void) {

 Harp i1;
 play_instrument(i1);

 Violin i2;
 play_instrument(i2);

}
```

```
Instrument::play()
Instrument::play()
```

Polimorfismo

Al contrario di JAVA, in c++ le classi **NON** sono polimorfe di default, è necessario abilitare il polimorfismo per ciascuna classe che ne abbia bisogno.



Premettere il sostantivo virtual rende il metodo polimorfo, così da inserire del codice specifico per la gestione delle chiamate a funzione (possibile overhead nelle prestazioni).
Una volta definito virtual, il metodo è virtual in tutte le classi derivate.
Una classe con funzioni virtual è detta polimorfa e la ridefinizione di uno di essi è detta overriding.

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

class Instrument {
public:
 Instrument(void) {
   cout<<"Instrument::Instrument()"<<endl;
 }

 virtual ~Instrument::Instrument(void) {
   cout<<"Instrument::~Instrument()"<<endl;}
};


class Harp : public Instrument
{
public:
 Harp(void) {
  cout<<"Harp::Harp()"<<endl;
 }

 ~Harp(void) {
  cout<<"Harp::~Harp()"<<end;
 }

};
```

```
int main(void){
 Harp *h = new Harp;
 ...
 delete h;
}
```

```
Instrument::Instrument()
Harp::Harp()
Harp::~Harp()
Instrument::~Instrument()
```

```
int main(void) {
 Instrument *i = new Harp;
 ...
 delete i;
}
```

```
Instrument:: Instrument()
Harp::Harp()
Harp::~Harp()
Instrument::~Instrument()
```

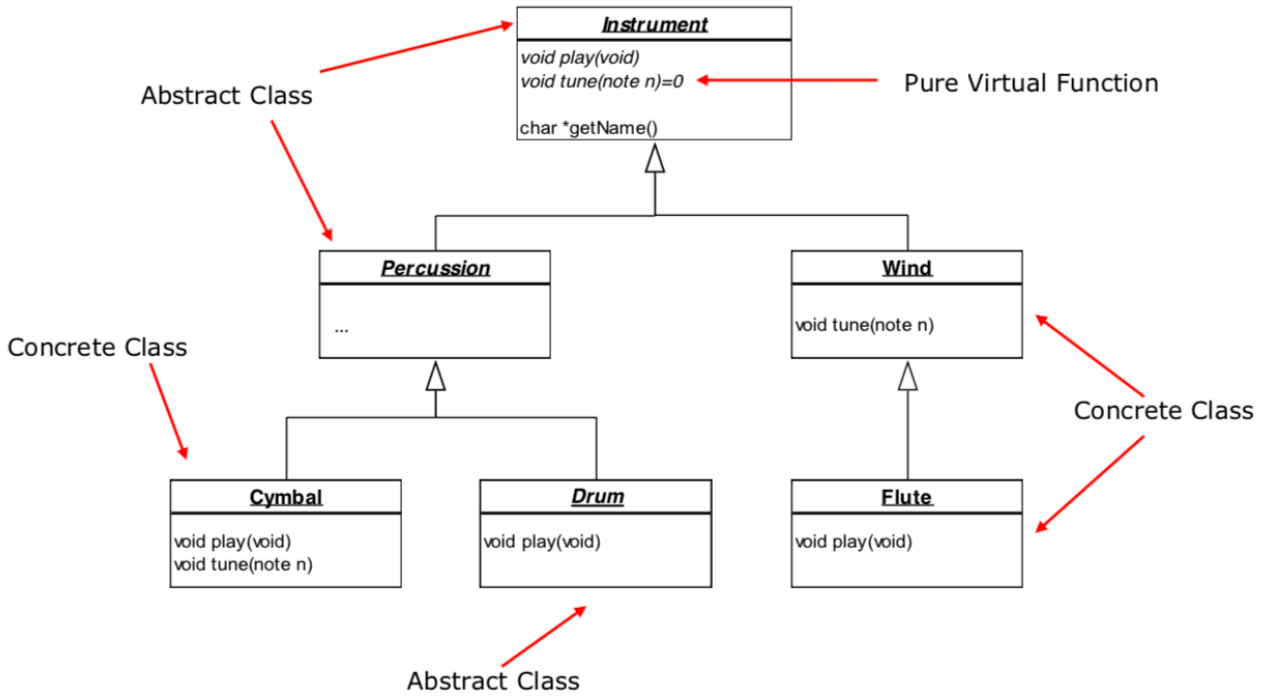
È possibile forzare un metodo ad essere “vituale puro”.
La classe con metodi virtuali puri è una abstract class e non è possibile istanziarla, è usabile come pura interfaccia per le classi derivate.
Solo le classi derivate che fanno override sono istanziabili, la classe che dichiara un metodo virtuale puro può anche definirlo, ma rimane non istanziabile.

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

class Instrument {
public:
 Instrument(void){}
 virtual void play()=0;
};

class Harp : public Instrument
{
public:
 Harp(void){}
 void play(){cout<<"Harp::play()"<<endl;}
};

class Violin : public Instrument
{
public:
  Violin(void){}
  void play(){cout<<"Violin::play()"<<endl;}
};
```

Inutile implementare un codice specifico

```
void play_instrument(Instrument &b){
 ...
 b.play();
 ...
}

void main(void) {

 Instrument i0; // Errore

 Harp i1;
 play_instrument (i1);

 Violin i2;
 play_instrument (i2);

}
```

```
Harp::play()
Violin::play()
```



Se tutte le funzioni membro sono “virtuali pure”, la classe è simile ad una INTERFACE in JAVA, non c’è però l’obbligo di implementare tutte le funzioni o esplicitare la keyword abstract.
Il cast esplicito tra tipi è utilizzabile SOLO con oggetti polimorfi, esso viene effettuato a rum-time, con overhead per i controlli ed è utile per un downcast safe.
Se il cast non è possibile, su un puntatore ritorna null altrimenti su reference lancia una eccezione di tipo std::bad_cast.

```
class BaseA{
public:
 virtual ~Base(){}
};

class DerivataA1 : public BaseA {};
class DerivataA2 : public BaseA {};

class BaseB {};

Class DerivataB1 : public BaseB {};


int main(void){
 DerivataA1 A1;
 BaseA &A=A1;

 DerivataA2 *A2=dynamic_cast<DerivataA2*>(A);     // A2=NULL
 DerivataA2 &A2=dynamic_cast<DerivataA2&>(A);     // Exception std::bad_cast

 DerivataB1 B1;
 BaseB *B=&B1;

 DerivataB1 *BB1=dynamic_cast<DerivataB1*>(B);    // Error: no polymorphic type
 DerivataB1 &BB1=dynamic_cast<DerivataB1&>(B);    // Error: no polymorphic type
}
```

È possibile ereditare funzionalità da più classi contemporaneamente.

Fondamentali

Nelle classi implementare con la massima attenzione

- Costruttore di default
- Distruttore
- Copy Conctructor
- Operatore Assegnamento

Ecco il link del mio progetto in GitHub