Programmazione ad oggetti semplice (per davvero)

Rovesti Gabriel

Attenzione

Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.
Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

Teoria 3
Introduzione 3
Namespace e Classe Orario: primi costruttori, new, information hiding 4
Costruttori: tipi, explicit, const e const saga, static 14
Overloading, static, operatori di confronto e primi esempi 21
Modularizzazione dei programmi in file, Makefile, Modularizzazione delle classi, Relazione has-a 34
Container, interferenza, shallow/deep copy, distruttori e distruttori profondi, lifetime, array 51
Dichiarazioni di classi, friend, nested classes, iteratori ed esempi 70
Tipi di dati C++, tipi di conversioni, tipi di cast, Template, template di classe, Friend e classi annidate 88
C++ Libreria Standard, vector e metodi, contenitori, Ereditarietà, Static Binding, Basi, derivate, costruttori, operatori e loro definizione 113
Polimorfismo, Dynamic binding/lookup, Overriding, Distruttori virtuali, RTTI 153
Gang of Four, MVC (Model View Controller), Qt e classi principali (Widget, QObject), Signals and slots, Ereditarietà Multipla, Stream di file e stringhe, Eccezioni 175
Qt: Segnali, slot, esempi vari 180
Ereditarietà multipla, ereditarietà a diamante e gestione virtual 184
Gerarchia delle classi input/output, stream, errori ed eccezioni 195

Teoria

Introduzione

Un programma è costituito da:
- Algoritmi (programmazione procedurale)
- Dati su cui operano gli algoritmi (programmazione ad oggetti)

La programmazione orientata agli oggetti (OOP) è un paradigma di programmazione che si basa sul concetto di classi e oggetti. Si usa per strutturare un programma software in pezzi di codice semplici e riutilizzabili (di solito chiamati classi), che vengono usati per creare istanze individuali di oggetti.

La programmazione orientata agli oggetti prevede il raggruppamento in un'area circoscritta del codice sorgente (chiamata classe) della dichiarazione delle strutture di dati e delle procedure che operano su di esse. Le classi, quindi, costituiscono dei modelli astratti, che a tempo di esecuzione vengono invocati per istanziare o creare oggetti software relativi alla classe invocata. Questi ultimi sono dotati di attributi (dati) e metodi (procedure) definiti/dichiarati dalle rispettive classi.
La parte del programma che fa uso di un oggetto è chiamata *client*.

Un linguaggio di programmazione è definito orientato agli oggetti quando permette di implementare tre meccanismi utilizzando la sintassi nativa del linguaggio:
- incapsulamento;
- ereditarietà;
- polimorfismo.

L'incapsulamento consiste nel separare la cosiddetta interfaccia di una classe dalla sua corrispondente implementazione, in modo che i clienti di un oggetto di quella classe possano usare la prima ma non la seconda.
L'ereditarietà consente essenzialmente di definire classi a partire da altre già definite.
Il *polimorfismo* permette di scrivere un client che può utilizzare oggetti di classi diverse, ma con la stessa interfaccia comune; a tempo di esecuzione, tale client attiverà comportamenti diversi senza conoscere a priori il tipo specifico dell'oggetto che gli viene passato.

Utilizziamo come linguaggio il C++, linguaggio naturalmente orientato verso la programmazione ad oggetti ed in grado di incapsulare tutti i nostri costrutti.
Esso fornisce alcuni strumenti utili per estendere i paradigmi di programmazione procedurale e ad oggetti:
- I template di funzione
- I template di classe
- Gestione delle eccezioni

Alcune caratteristiche del C++:

- Compilato
- Tipizzazione forte statica (strongly typed)
- No garbage collector
- Standardizzato (ultimo standard ANSI C++17, C++20 in progress)
- General-purpose e molto diffuso (Adobe SW, Mozilla SW, MySQL, Microsoft SW, Google Chromium, Games...)
- Efficienza
- Operatori e loro overloading
- Librerie

Namespace e Classe Orario: primi costruttori, new, information hiding

Nella programmazione modulare, si ha il problema dell'inquinamento dello spazio dei nomi. In particolare, non si capisce chi stia chiamando cosa.

```
// file Complex.h

struct Complex {
  ... // implementazione1
};

double module(Complex);
```

```
// qualche altro file
#include "Complex.h"

// dichiarazione ILLEGALE
struct Complex {
 ... // implementazione2
}

void f(...) {
// vorrebbe usare
// entrambi i Complex
}
```

Spesso, per capire a chi ci si sta riferendo, si adotta l'operazione di scoping (due volte i due punti), per capire a quale *namespace* ci stiamo riferendo.

```
// file Lib_UNO.h

namespace SPAZIO_UNO {
  struct Complex {...};
  void f(Complex c) {...};
};
```

```
// file Lib_DUE.h

namespace SPAZIO_DUE {
  struct Complex {...};
  void g(Complex c) {...};
};
```

Operatore di scoping

namespace::nome_dichiarazione;

```
#include "Lib_UNO.h"
#include "Lib_DUE.h"

void funzione() {
  SPAZIO_UNO::Complex var1; SPAZIO_UNO::f(var1);
  SPAZIO_DUE::Complex var2; SPAZIO_DUE::g(var2);
  SPAZIO_DUE::g(var1); // ERRORE IN COMPILAZIONE !!
}
```

È anche possibile fornire degli alias come si vede qui:

```
#include "Lib_UNO.h"
#include "Lib_DUE.h"

namespace UNO = SPAZIO_UNO;
namespace DUE = SPAZIO_DUE;

void funzione() {
  UNO::Complex var1; UNO::f(var1);
  DUE::Complex var2; DUE::g(var2);
}
```

Il meccanismo dei namespace permette di incapsulare dei nomi che altrimenti inquinerebbero il namespace globale.
In C++ si utilizzano diversi spazi dei nomi. Possiamo anche creare i nostri spazi dei nomi. Ad esempio, generalmente si utilizza lo spazio dei nomi standard chiamato std. Scriviamo la sintassi come *using namespace std;*

La libreria standard contiene le funzionalità più comuni che si utilizzano nella costruzione delle applicazioni, come i contenitori, gli algoritmi e così via. Se i nomi utilizzati da queste funzionalità fossero stati resi pubblici, ad esempio se avessero definito una classe di coda a livello globale, non sarebbe stato possibile utilizzare di nuovo lo stesso nome senza conflitti. Così è stato creato uno spazio dei nomi, std, per contenere questo cambiamento.

La dichiarazione *using namespace* significa solo che nello scope in cui è presente, rende disponibili tutte le cose sotto lo spazio dei nomi std senza dover anteporre il prefisso std:: a ciascuna di esse.

Sebbene questa pratica vada bene per il codice di esempio, non va bene se si inserisce l'intero spazio dei nomi std nello spazio dei nomi globale, in quanto vanifica lo scopo degli spazi dei nomi e può portare a collisioni di nomi. Questa situazione è chiamata inquinamento dello spazio dei nomi.

Una dichiarazione d'uso rende visibile in un namespace una singola dichiarazione:

```
#include "Lib_UNO.h"
#include "Lib_DUE.h"

using SPAZIO_UNO::Complex;
using SPAZIO_DUE::g;

void funzione() {
  Complex var1; SPAZIO_UNO::f(var1);
  SPAZIO_DUE::Complex var2; g(var2);
}
```

Similmente, la direttiva d'uso, indicata come using namespace:

```
#include "Lib_UNO.h"
//rende visibili tutti i nomi del namespace SPAZIO_UNO
using namespace SPAZIO_UNO;

void funzione() {
  Complex var1; f(var1);
  SPAZIO_DUE::Complex var2; SPAZIO_DUE::g(var2);
}
```

Le componenti di tutte le librerie del C++ standard sono dichiarate in un namespace chiamato **std**.

Nel C++ standard (ad esempio il compilatore **g++**), il seguente codice non compila.

```
#include <iostream>

int main() {
  cout << "Ciao!" << endl;
}
// COMPILATORE g++:
// `cout' undeclared
// `endl' undeclared
```

Posso riferirmi alternativamente alle componenti *cout* ed *endl* tramite l'operatore di scoping:

```
#include <iostream>

using namespace std;

int main() {
  cout << "Ciao!" << endl;
}
// Compila correttamente con g++
```

Alternativamente usando l'operatore di scoping:

```
#include <iostream>

int main() {
  std::cout << "Ciao!" << std::endl;
}
// Compila correttamente con g++
```

In C++ si ha il cosiddetto tipo *string*, esempio di classe presente nella libreria standard STL.

```
// stringhe nello stile C
char * st = "ecco una stringa/0";
char * st = "ecco una stringa"; // equivalente!
```

Le stringhe sono oggetti che rappresentano sequenze di caratteri.
La classe standard string fornisce il supporto per tali oggetti con un'interfaccia simile a quella di un contenitore standard di byte, ma con l'aggiunta di funzionalità specificamente progettate per operare con stringhe di caratteri a singolo byte.

```
#include<iostream>

using namespace std;

int main() {
  char * s1 = "pippo\0";
  for(int i=0; *(s1+i); i++) cout << *(s1+i); cout << endl;
  // stampa: pippo
  char * s2 = "pippo";
  for(int i=0; *(s2+i); i++) cout << *(s2+i); cout << endl;
  // stampa: pippo
}
```

La classe *string* è un esempio di template; quindi, si possono istanziare sia caratteri che stringhe come si vede qui:

```
#include <string>
using namespace std;
int main() {
 char * s = "ecco una stringa\0";  // stile C
 string st("ecco una stringa");    // stile C++
 string st = "ecco una stringa";   // stile C++
}
```

Vediamo anche un primo esempio di costruttori; l'esempio della costruzione di una stringa vuota oppure *di copia*:

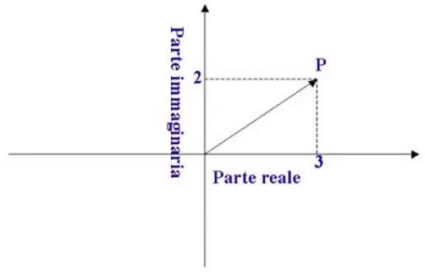
```
string st1;       // costruttore: dichiara una stringa e la
                  // inizializza come vuota
string st2(st);   // costruttore di copia:
                  // st2 è una copia di st
```

Definiamo gli ADT/Abstract Data Type come un tipo (o classe) per oggetti il cui comportamento è definito da un insieme di valori e da un insieme di operazioni. La definizione di ADT menziona solo le operazioni da eseguire, ma non il modo in cui queste operazioni saranno implementate. Non specifica come verranno organizzati i dati in memoria e quali algoritmi verranno utilizzati per implementare le operazioni. Si chiama "astratto" perché fornisce una visione indipendente dall'implementazione.
Normalmente questi hanno:

- Dei metodi pubblici con proprie operazioni
- Una rappresentazione interna inaccessibile

Un esempio tranquillo di questo è la rappresentazione cartesiana:

```
//complessi.h
struct comp { double re, im; };

comp inizializzaComp(double, double);
double reale(comp);
double immag(comp);
comp somma(comp, comp);
```



Rappresentazione Cartesiana

```
//complessi.cpp
#include"complessi.h"

comp inizializzaComp(double re, double im){
  comp x;
  x.re=re; x.im=im;
  return x;
}

double reale(comp x){
  return x.re;
}

double immag(comp x){
  return x.im;
}

comp somma(comp x, comp y){
  comp z;
  z.re=x.re+y.re; z.im=x.im+y.im;
  return z;
}
```

```
#include "complessi.h"
#include<iostream>

int main() {
 comp z1;
 comp x1 = inizializzaComp(0.3,3.1);
 comp y1 = inizializzaComp(3,6.3);
 z1=somma(x1,y1);

 // possiamo però usare la rappresentazione interna dell'ADT !
 comp x2 = {0.3,3.1}, y2 = {3,6.3};
 comp z2;
 z2.re=x2.re+y2.re; z2.im=x2.im+y2.im;

 std::cout << "z1 => (" << reale(z1) <<"," << immag(z1) << ")\n";
 std::cout << "z2 => (" << z2.re <<"," << z2.im << ")\n";
}
```

Essi come si vede implementano vari concetti:

- Dichiarazione di campi dati e metodi
- Definizione o implementazione della classe

Un esempio standard: la classe Orario.

```
class orario {
private:          // unico campo dati della classe
 int sec;         // scegliamo di rappresentare l'ora
                  // del giorno con il numero di
                  // secondi trascorsi dalla mezzanotte




};
```

```
class orario {
private:          // unico campo dati della classe
 int sec;         // scegliamo di rappresentare l'ora
                  // del giorno con il numero di
                  // secondi trascorsi dalla mezzanotte

public:           // metodi pubblici della classe
 int Ore();       // selettore delle ore
 int Minuti();    // selettore dei minuti
 int Secondi();   // selettore dei secondi
};
```

Definizione della classe orario

```
int orario::Ore() {
  return sec / 3600;
}

int orario::Minuti() {
  return (sec / 60) % 60;
}

int orario::Secondi() {
  return sec % 60;
}
```

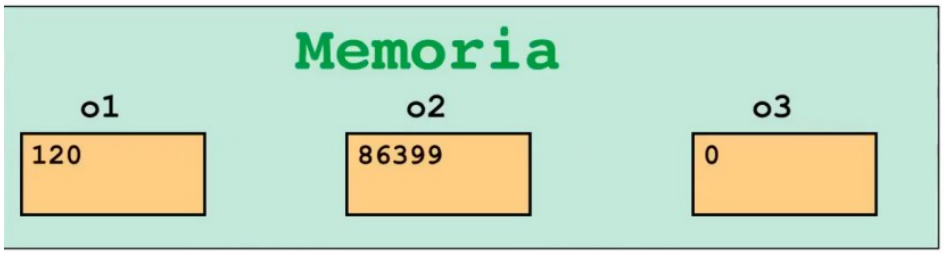
Avremmo anche potuto scrivere:

```
class orario {
public:
 int Ore() { return sec / 3600; }
 int Minuti() { return (sec / 60) % 60; }
 int Secondi() { return sec % 60; }
private:
 int sec;
};
```

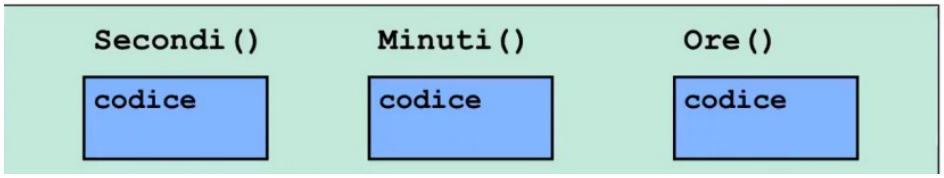
Questa seconda definizione è detta inline, cioè una funzione per la quale il compilatore copia il codice dalla definizione della funzione direttamente nel codice della funzione chiamante, invece di creare un insieme separato di istruzioni in memoria. In questo modo si elimina l'overhead del call-linkage e si possono ottenere notevoli opportunità di ottimizzazione.

Ogni oggetto memorizza i propri valori dei campi dati

Le variabili di tipo *orario* sono oggetti della classe *orario*. Ciascun oggetto memorizza i propri valori dei campi dati, mettendo un'unica copia in memoria dei metodi per tutti gli oggetti:



Unica copia in memoria dei metodi per tutti gli oggetti



Un legame implicito tra oggetti di invocazione (di tipo C) e il metodo invocato è il this.

Per comprendere il puntatore 'this', è importante sapere come gli oggetti guardano le funzioni e i membri dei dati di una classe.
Ogni oggetto ottiene la propria copia del membro dei dati.
Tutti accedono alla stessa definizione di funzione presente nel segmento di codice.
Ciò significa che ogni oggetto ottiene la propria copia dei membri dei dati e tutti gli oggetti condividono una singola copia delle funzioni membro.

Se esiste una sola copia di ogni funzione membro, utilizzata da più oggetti, come si accede ai membri dei dati e come li si aggiorna?
Il compilatore fornisce un puntatore implicito insieme ai nomi delle funzioni come 'this'.
Il puntatore 'this' viene passato come argomento nascosto a tutte le chiamate a funzioni membro non statiche ed è disponibile come variabile locale nel corpo di tutte le funzioni non statiche. Il puntatore 'this' non è disponibile nelle funzioni membro statiche, poiché le funzioni membro statiche possono essere chiamate senza alcun oggetto (con il nome della classe).

Possiamo "esplicitare" il parametro implicito `this` nella dichiarazione del metodo `Secondi()` e nella sua chiamata:

```
// la definizione
int orario::Secondi() { return sec % 60; }
// esplicitando il parametro this diventerebbe
int orario::Secondi(orario* this) {
                            return (*this).sec % 60;
}

// mentre la chiamata
int s = mezzanotte.Secondi();
// esplicitando il parametro implicito diventerebbe
int s = Secondi(&mezzanotte);
```

this è un puntatore e *this è un puntatore dereferenziato.

Se una funzione restituisse this, sarebbe un puntatore all'oggetto corrente, mentre una funzione che restituisse *this sarebbe un "clone" dell'oggetto corrente, allocato sullo stack, a meno che non si sia specificato il tipo di ritorno del metodo per restituire un riferimento.

`this` è una keyword

Dereferenziazione `*this` nel corpo di un metodo

Ad esempio:

```
int orario::Secondi() {
  return (*this).sec % 60;
}
```

```
int orario::Secondi() {
  return this->sec % 60;
}
```

A volte l'utilizzo esplicito del puntatore **`this`** diviene necessario nella definizione di qualche metodo.

```
class A {
 private:
   int a;
 public:
   A f();
}

A A::f() {
  a = 5;
  return *this;
}
```

In C++ si ha il concetto di information hiding, vale a dire, impostare la modalità di accesso alle variabili, specificando:

- Una parte pubblica (interfaccia pubblica)
- Una parte privata (default)
- Interfaccia pubblica di accesso
- Specificatori di accesso tramite le keywords *public* oppure *private*

Dall'esterno si accede solo alla parte pubblica; i metodi della classe, ovviamente, possono accedere alla parte privata: non solo a quella dell'oggetto di invocazione ma a qualsiasi altro oggetto della classe (es. i parametri del metodo).

Aggiungiamo alla classe complesso una serie di parametri e li definiamo:

```
//complesso.cpp

class complesso {
 private:
  double re, im;
 public:
  void inizializza(double, double);
  double reale();
  double immag();
};

void complesso::inizializza(double r, double i)
{ re = r; im = i; }

double complesso::reale()
{ return re; }

double complesso::immag()
{ return im; }
```

```
//complesso2.cpp

#include<math.h>

// rappresentazione polare: modulo mod e argomento arg
class complesso_pol {
private:
  double mod, arg;
 public:
  void inizializza(double, double);
  double reale();
  double immag();
};

void complesso_pol::inizializza(double r, double i)
{ mod = sqrt(r*r + i*i); arg = atan(i/r); }

double complesso_pol::reale()
{ return mod*cos(arg); }

double complesso_pol::immag()
{ return mod*sin(arg); }
```

Con questa dichiarazione, l'uso di *sec* è un campo privato della classe orario ed è errore:

```
orario mezzanotte;
mezzanotte.sec = 0; // Errore: sec è privato
```

I costruttori sono tipicamente dichiarati nella parte pubblica di una classe.
Il costruttore in C++ è un metodo speciale che viene invocato automaticamente al momento della creazione di un oggetto. Viene utilizzato per inizializzare i membri dei dati dei nuovi oggetti in generale. Il costruttore in C++ ha lo stesso nome della classe o della struttura. Il costruttore viene invocato al momento della creazione dell'oggetto. Costruisce i valori, cioè fornisce i dati per l'oggetto, motivo per cui è noto come costruttore.

Normalmente, abbiamo il costruttore di default (senza parametri):

```
class orario {
public:
  orario(); // costruttore senza parametri
  ...       // detto anche costruttore di default
};
```

```
orario::orario() {  // definizione del costruttore
  sec = 0;          // di default
}
```

```
orario mezzanotte;  // viene invocato il
                    // costruttore di default
cout << mezzanotte.Ore() << endl; // stampa 0
```

È anche possibile definire più costruttori, purché differiscano nel numero e/o tipo dei parametri. Si fa quindi overloading dell'identificatore del metodo con lo stesso nome.
Il sovraccarico di funzioni/overloading è definito come il processo di avere due o più funzioni con lo stesso nome, ma con parametri diversi, noto come sovraccarico di funzioni in C++.

```
class orario {
public:
  orario();          // costruttore di default
  orario(int,int); // costruttore ore-minuti
  orario(int,int,int); // ore-minuti-secondi
  ...
};
```

```
orario::orario() {  // costruttore di default
  sec = 0;
};
```

```
orario::orario(int o, int m) {
  if (o < 0 || o > 23 || m < 0 || m > 59)
    sec = 0;
  else sec = o * 3600 + m * 60;
};
```

```
orario::orario(int o, int m, int s) {
  if (o < 0 || o > 23 || m < 0 || m > 59 ||
      s < o || s > 59) sec = 0;
  else sec = o * 3600 + m * 60 + s;
};
```

Esempi d'uso dei costruttori:

```
orario adesso_preciso(14,25,47);
     // usa il costruttore ore-minuti-secondi
orario adesso(14,25);
     // usa il costruttore ore-minuti
orario mezzanotte;
     // usa il costruttore senza parametri
orario mezzanotte2;
    // usa il costruttore senza parametri
orario troppo(27,25);
     // usa il costruttore ore-minuti
```

```
cout << adesso_preciso.Secondi() << endl;
                                  // stampa 47
cout << adesso.Minuti() << endl;
                                  // stampa 25
cout << mezzanotte.Ore() << endl;
                                  // stampa 0
cout << troppo.Ore() << endl;
                                  // stampa 0
```

I costruttori si possono anche invocare nel seguente modo:

```
orario adesso_preciso = orario(14,25,47);
orario adesso = orario(14,25);
orario mezzanotte = orario();
```

Si tratta in questo caso di invocazioni del **costruttore di copia**

Il costruttore di copia è un costruttore che crea un oggetto inizializzandolo con un oggetto della stessa classe, creato in precedenza. Il costruttore di copia viene utilizzato per

- Inizializzare un oggetto da un altro dello stesso tipo.
- Copiare un oggetto per passarlo come argomento a una funzione.
- Copiare un oggetto per restituirlo da una funzione.

Se un costruttore di copia non è definito in una classe, è il compilatore stesso a definirlo. Se la classe ha variabili puntatore e ha allocazioni dinamiche di memoria, è indispensabile avere un costruttore di copia.

Scritto da Gabriel

Nell'assegnazione:

```
orario t;
t = orario(12,33,25);
```

Il costruttore crea un oggetto temporaneo anonimo (senza l-value) della classe orario. Vuol dire che "lo crea solo per assegnare il proprio valore".
In C++ un l-valore è qualcosa che punta a una specifica posizione di memoria. D'altra parte, gli r-valori è qualcosa che non punta da nessuna parte. In generale, gli r-valori sono temporanei e di breve durata, mentre gli l-valori hanno una vita più lunga perché esistono come variabili.

Un'espressione come **`orario(12,33,25)`** può essere considerata come un "valore" del tipo **`orario`**. È un r-valore non indirizzabile.

Quando si invoca un costruttore, si ha una creazione dinamica sullo heap con la keyword "new".

```
orario* ptr = new orario;
orario* ptr1 = new orario(14,25);

cout << ptr->Ore() << endl;  // stampa 0
cout << ptr1->Ore() << endl; // stampa 14
```

L'operatore new è un operatore che denota una richiesta di allocazione di memoria sull'Heap. Se la memoria disponibile è sufficiente, l'operatore new inizializza la memoria e restituisce l'indirizzo della memoria appena allocata e inizializzata alla variabile puntatore.

Metodo 1 (usando new)

- Alloca la memoria per l'oggetto nell'archivio libero (che spesso è la stessa cosa dell'heap).
- Richiede di cancellare esplicitamente l'oggetto in un secondo momento. (Se non lo si cancella, si potrebbe creare una perdita di memoria).
- La memoria rimane allocata finché non viene cancellata. (cioè si potrebbe restituire un oggetto creato con new)

L'esempio riportato nella domanda comporta una perdita di memoria, a meno che il puntatore non venga eliminato, e dovrebbe essere sempre eliminato, indipendentemente dal percorso di controllo seguito o dal lancio di eccezioni.

Metodo 2 (senza usare new)

- Alloca la memoria per l'oggetto sullo stack (dove vanno tutte le variabili locali) In genere c'è meno memoria disponibile per lo stack; se si allocano troppi oggetti, si rischia lo stack overflow.
- Non sarà necessario cancellarla in seguito.
- La memoria non viene più allocata quando esce dallo scope. (cioè non si dovrebbe restituire un puntatore a un oggetto sullo stack).
- Per quanto riguarda il metodo da utilizzare, si sceglie quello che funziona meglio per l'utente, in base ai vincoli di cui sopra.

Riassunto della classe orario.

```
class orario {
public:
  orario();              // costruttore di default
  orario(int,int);       // costruttore ore-minuti
  orario(int,int,int); // costruttore ore-minuti-secondi
  int Ore();             // selettore delle ore
  int Minuti();          // selettore dei minuti
  int Secondi();         // selettore dei secondi
private:
  int sec;               // campo dati
};
```

Costruttori: tipi, explicit, const e const saga, static

Attenzione ai due seguenti casi:

```
class orario {
public:
// nessuna definizione di costruttori
  ...
};

orario o; // OK: viene invocato il
          // costruttore standard di default
```


```
class orario {
public:
  orario(int,int); // solo costruttore ore-minuti
  ...
};

orario o; // Errore: manca un costruttore di default
```

Altri esempi:

```
orario adesso(11,55);    // costruttore ore-minuti
orario copia;            // costruttore di default
copia = adesso;          // assegnazione
orario copia1 = adesso; // costruttore di copia
                         // analogo a int x = y;
orario copia2(adesso);   // costruttore di copia
                         // analogo a int x(y);
```

Il costruttore di copia crea un nuovo oggetto *copia1* in cui copia, campo dati per campo dati, l'oggetto *adesso*. L'assegnazione assegna il valore di ogni campo dati dell'oggetto *adesso* al corrispondente campo dati dell'oggetto preesistente *copia*. Di seguito, il costruttore di copia e l'operatore di assegnazione.

```
C(const C&)
C& operator=(const C&)
```

I costruttori possono essere usati come conversione implicita di tipo.

```
orario s;
s = 8;
```

Con l'assegnazione `s = 8;` succede che:
1) viene invocato il costruttore **`orario(int)`** con parametro attuale 8 che crea un oggetto anonimo temporaneo della classe **`orario`**
2) l'oggetto temporaneo viene assegnato all'oggetto **`s`**
3) viene "deallocato" l'oggetto temporaneo

```
orario s,t;
s = 8;        // OK: equivale a s = orario(8)
t = 8+12;     // OK: equivale a t = orario(8+12)
```

Attenzione anche agli *argomenti di default* nelle funzioni.
L'ordine nel chiamarli deve essere esattamente uguale al prototipo della funzione e se un parametro viene inizializzato nella firma della funzione, *anche tutti quelli dopo di lui devono essere inizializzati*.

```
void F(double x, int n = 3, string s); // ILLEGALE!

void G(double x, int n = 3, string s = "ciao"); // OK

G(3.2); // OK: G(3.2,3,"ciao");
G();    // NO
G(3,3); // OK: G(3,3,"ciao");
G(3,3,"pippo"); // OK: G(3,3,"pippo");
```

L'esempio del costruttore a più parametri, tra le altre cose, in grado anche di agire come convertitore di tipo da *int* ad *orario*.

```
orario::orario(int o = 0, int m = 0, int s = 0)
{
  if (o < 0 || o > 23 || m < 0 || m > 59 ||
      s < o || s > 59) sec = 0;
  else sec = o * 3600 + m * 60 + s;
};
```

Proseguendo con la classe *orario*:

```
class orario {
public:
  // costruttore a tre parametri con argomenti di default
  orario(int =0,int =0,int =0);
  int Ore();              // selettore delle ore
  int Minuti();           // selettore dei minuti
  int Secondi();          // selettore dei secondi
private:
  int sec;
};


// attenzione alla sintassi
orario::orario(int o, int m, int s) {
  if (o < 0 || o > 23 || m < 0 || m > 59 ||
      s < 0 || s > 59) sec = 0;
  else sec = o * 3600 + m * 60 + s;
}

int orario::Ore() { return sec / 3600; }
int orario::Minuti() { return (sec / 60) % 60; }
int orario::Secondi() { return sec % 60; }
```

```
#include<iostream>
#include "orario.cpp"
using namespace std::cout; using namespace std::endl;

int main() {
  orario s;
  s = 6; // equivale a: s = orario(6,0,0);
  cout << s.Ore() << ':' << s.Minuti() << ':'
       << s.Secondi() << endl;
  orario t = 5+2*2; // equivale a: orario t = orario(5+2*2,0,0);
  cout << t.Ore() << ':' << t.Minuti() << ':'
       << t.Secondi() << endl;
  orario r(12,45); // equivale a: orario r(12,45,0);
  cout << r.Ore() << ':' << r.Minuti() << ':'
       << r.Secondi() << endl;
  orario a; // equivale a: orario a(0,0,0);
  cout << a.Ore() << ':' << a.Minuti() << ':'
       << a.Secondi() << endl;
  orario b(7); // equivale a: orario b(7,0,0);
  cout << b.Ore() << ':' << b.Minuti() << ':'
       << b.Secondi() << endl;
}
```

Ecco anche gli operatori espliciti di conversione, come nel caso d'uso ad interi:

```
class orario {
public:
  operator int() {return sec;} // orario ⇒ int
  ...
};

orario o(14,37);
int x = o;     // OK: viene richiamato
               // implicitamente il metodo int() su o
```

Per bloccare la conversione a qualche tipo, si usa la keyword *explicit*, che permette di non richiamare implicitamente il costruttore del tipo.

```
class orario {
public:
  explicit orario(int); // costruttore esplicito
                        // con un solo parametro
  ...
};

orario o = 8; // Errore: non viene richiamato
              // implicitamente il costruttore
              // orario(8)
```

Ora, aggiungiamo ad *orario* due nuovi metodi pubblici:

```
class orario {
public:
  ...
  orario UnOraPiuTardi();
  void  AvanzaUnOra();
private:
  ...
};
```

```
orario orario::UnOraPiuTardi() {
  orario aux;
  aux.sec = (sec + 3600) % 86400;
  return aux;
};
```

```
void orario::AvanzaUnOra() {
  sec = (sec + 3600) % 86400;
};
```

Queste provocano dei side-effects, vale a dire delle modifiche, quando chiamate, all'oggetto di invocazione, che si ripercuote quando chiamato altrove.

```
orario mezzanotte;
cout << mezzanotte.Ore();    // stampa 0
orario adesso(15);
cout << adesso.Ore();        // stampa 15
adesso = mezzanotte.UnOraPiuTardi();
cout << adesso.Ore();        // stampa 1
cout << mezzanotte.Ore();    // stampa 0
mezzanotte.AvanzaUnOra();
cout << mezzanotte.Ore();    // stampa 1
```

Lo standard dettaglia:
"La lettura di un oggetto designato da un valore volatile l-value, la modifica di un oggetto, la chiamata di una funzione di I/O della libreria o la chiamata di una funzione che esegue una qualsiasi di queste operazioni sono tutti side-effects, ovvero cambiamenti nello stato dell'ambiente di esecuzione."

Riprendiamo con i metodi ed oggetti di invocazione costanti.

Quando un metodo NON produce side-effects, viene messa la keyword const alla fine della firma del metodo.

```
class orario {
public:
  void StampaSecondi() const;
  ...
};
```

```
void orario::StampaSecondi() const {
  std::cout << sec << std::endl;
};
```

Il compilatore controlla che nella definizione del metodo dichiarato `const` non compaia alcuna istruzione che possa provocare side-effect sull'oggetto di invocazione: assegnazioni ai campi dati dell'oggetto di invocazione ed invocazioni di metodi non costanti.

Spiegazione: in un metodo costante di una classe `C`, il puntatore `this` ha tipo `const C*`

Qualora invece la keyword const venga messa all'inizio di una variabile, metodo, ecc, si intende che si può usare solo per metodi dichiarati costanti (const davanti).

Anche un oggetto può essere dichiarato costante:

```
const orario LE_DUE(14); // è l'analogo di
const int tre = 3;
```

Attenzione: un oggetto costante si può usare come oggetto di invocazione **soltanto** per metodi dichiarati costanti.

```
const orario LE_TRE(15);
LE_TRE.StampaSecondi(); // OK: stampa 54000
orario t;
t = LE_TRE.UnOraPiuTardi(); // Errore! Non compila!
   // UnOraPiuTardi() non è stato
   // dichiarato come metodo costante
```

Eccezione (ovvia): i costruttori sono dei metodi non dichiarati costanti che possono venire invocati su oggetti dichiarati costanti!

Normalmente, il prof è molto preciso su questo.
Se il metodo non fa side-effects, si metta SEMPRE il const.

Nell'esempio che segue, l'alias è un riferimento che punta alla stessa area di memoria di un puntatore/oggetto; esso punta ad un valore variabile, non costante. Questa la differenza tra ciò che funziona e ciò che non è permesso. Una serie di esempi su cui ragionare.
Per vedere se corretto o sbagliato:

- Se non costante, deve puntare ad una variabile
- Se costante, punta ad un valore o ad un oggetto const/di copia (&) o ad un oggetto dereferenziato (*&) costante

```
int x=2;
int& a = x;  // ALIAS
int& a1 = 2; // ILLEGALE
a=5;
int y=3;
a=y;         // LEGALE, r-valore di y assegnato a
             // a l-valore di x
```

```
int x=2;
int* p = &x;
*p=5;
int y=3;
p=&y;        // LEGALE
```

```
int x=2;
int * const p = &x;
*p=5;    // LEGALE
int y=3;
p=&y;    // ILLEGALE
```

```
int x=2;
int & const r = x; // ILLEGALE (tipo illegale)
```

```
int x=2;
const int* p = &x;
*p=5;    // ILLEGALE
int y=3;
p=&y;    // LEGALE
```

```
int x=2;
const int *const p = &x;
*p=5;    // ILLEGALE
int y=3;
p=&y;    // ILLEGALE
```

```
int x=2;
const int& r = x; // RIFERIMENTO A TIPO COSTANTE
r=5;              // ILLEGALE
int y=3;
r=y;              // ILLEGALE
```

```
const int& r = 4; // LEGALE
r=5;              // ILLEGALE
int y=3;
r=y;              // ILLEGALE
```

```
const int & const r=2; // ILLEGALE
```

I successivi esempi hanno *fun* con un oggetto costante per riferimento (&) e funzionano. Se per valore, non funzionano.

```
void fun(const int& r);
// PASSAGGIO PER RIFERIMENTO (A TIPO) COSTANTE
int x=2;
fun(x); // LEGALE
fun(4); // LEGALE
```

```
const int& fun() { return 4; /* LEGALE */ }
// RITORNA RIFERIMENTO (A TIPO) COSTANTE

fun()=5;     // ILLEGALE
```

```
void fun_ref(const int& r);
// VERSUS
void fun_ptr(const int* p);
int x=2;
fun_ref(x);
// VERSUS
fun_ptr(&x);

fun_ref(4);  // LEGALE
// VERSUS
fun_ptr(&4); // ILLEGALE
```

```
void fun1(const Big& r); // PER RIFERIMENTO COSTANTE
// VERSUS
void fun2(Big v);        // PER VALORE

Big b(...);
fun1(b);     // copia di un riferimento a Big
// VERSUS
fun2(b);     // costruttore di copia di Big
```

Const correctness: usare la keyword *const* per prevenire modifiche agli oggetti quando possibile.
Questo è utile per la cosiddetta *type safety*, quindi utilizzare i tipi in modo corretto ed evitare conversioni non sicure (pericoloso).

I riferimenti in C++ non esistono; normalmente, tutto passa per i puntatori.
Nelle parole di Bjarne Stroustrup, creatore del C++:
"Come un puntatore, un riferimento è un alias per un oggetto, di solito è implementato per contenere un indirizzo macchina di un oggetto e non impone un sovraccarico di prestazioni rispetto ai puntatori, ma differisce da un puntatore per questo:
- Si accede a un riferimento esattamente con la stessa sintassi del nome di un oggetto.
- Un riferimento si riferisce sempre all'oggetto a cui è stato inizializzato.
- Non esiste un ''riferimento nullo'' e si può assumere che un riferimento si riferisca a un oggetto.
Anche se un riferimento è in realtà un puntatore, non dovrebbe essere usato come un puntatore ma come un alias."

Scritto da Gabriel

Esercizio utile di comprensione sui const:

```
class C {
private:
  int x;
public:
  C(int n = 0) {x=n;}
  C F(C obj) {C r; r.x = obj.x + x; return r;}
  C G(C obj) const {C r; r.x = obj.x + x; return r;}
  C H(C& obj) {obj.x += x; return obj;}
  C I(const C& obj) {C r; r.x = obj.x + x; return r;}
  C J(const C& obj) const {C r; r.x = obj.x + x; return r;}
};

int main() {
  C x, y(1), z(2); const C v(2);
  z=x.F(y); // OK
//! v.F(y); // ILLEGALE: "passing const C as this discards qualifiers"
  v.G(y); // OK
  (v.G(y)).F(x); // OK
  (v.G(y)).G(x); // OK
//! x.H(v); // ILLEGALE: "no matching function for call to C::H(const C&)"
//! x.H(z.G(y)); // ILLEGALE (!!): no matching function for call to C::H(C)
  x.I(z.G(y)); // OK (nota bene!)
  x.J(z.G(y)); // OK
  v.J(z.G(y)); // OK
}
```

Overloading, static, operatori di confronto e primi esempi

Le considerazioni principali sono due. Una è la spesa per copiare l'oggetto passato e la seconda è l'ipotesi che il compilatore può fare quando l'oggetto è un oggetto locale.

Ad esempio, nella prima forma, nel corpo di f non si può assumere che a e b non facciano riferimento allo stesso oggetto; quindi il valore di a deve essere riletto dopo qualsiasi scrittura su b, per sicurezza. Nella seconda forma, a non può essere modificato tramite una scrittura su b, poiché è locale alla funzione, quindi queste riletture non sono necessarie.

```
void f(const Obj& a, Obj& b)
{
  // a e b potrebbero fare riferimento allo stesso oggetto
}

void f(Obj a, Obj& b)
{
  // a è locale, b non può essere un riferimento ad a
}
```

Ad esempio: Nel primo esempio, il compilatore può assumere che il valore di un oggetto locale non cambi quando viene effettuata una chiamata non correlata. Senza informazioni su h, il compilatore non può sapere se un oggetto a cui quella funzione ha un riferimento (tramite un parametro di riferimento) non viene modificato da h. Ad esempio, quell'oggetto potrebbe far parte di uno stato globale che viene modificato da h.

```
void g(const Obj& a)
{
  // ...
  h(); // il valore di a potrebbe cambiare
  // ...
```

Scritto da Gabriel

```
}

void g(Obj a)
{
  // ...
  h(); // è improbabile che il valore di a cambi
  // ...
}
```

Sfortunatamente, questo esempio non è solido. È possibile scrivere una classe che, ad esempio, aggiunge un puntatore a sé stessa a un oggetto di stato globale nel suo costruttore, in modo che anche un oggetto locale di tipo classe possa essere alterato da una chiamata di funzione globale. Ciononostante, ci sono ancora potenzialmente più opportunità di ottimizzazioni valide per gli oggetti locali, dato che non possono essere soggetti ad aliasing direttamente da riferimenti passati o da altri oggetti preesistenti.

Il passaggio di un parametro tramite riferimento *const* dovrebbe essere scelto quando la semantica dei riferimenti è effettivamente richiesta, oppure come miglioramento delle prestazioni solo se il costo del potenziale aliasing è superiore al costo della copia del parametro.

```
class C {
  int a[1000]; // 4000 bytes
};

bool byValue(C x) {return true;}
bool byConstReference(const C& x) {return true;}

int main() {
  C obj;
  for(int i=0; i<10000000; i++) byValue(obj);          // 3.368 sec
  for(int i=0; i<10000000; i++) byConstReference(obj); // 0.031 sec
                                                       // 108x
}
```

Notevole anche il confronto a livello temporale, come si può notare.

Definire un metodo **`OraDiPranzo()`** che ritorna sempre uno specifico oggetto della classe **`orario`** che rappresenta l'orario canonico del pranzo.

Tentativo: un metodo costante **`OraDiPranzo()`** che ritorna un oggetto della classe **`orario`**

```
class orario {
public:
  orario OraDiPranzo() const;
  ...
};
```

```
orario orario::OraDiPranzo() const {
  return orario(13,15);
};
```

Ciò sarebbe evidentemente inutile: essendo valore costante, non ha senso passare un numero (tra le altre cose locale, solo per avere un valore costante).
La soluzione è definire il metodo dell'ora di pranzo come static, che serve a dichiarare una variabile che permane nello stesso valore per tutto il programma, come segue:

```
class orario {
public:
  static orario OraDiPranzo(); // metodo statico
  // il modificatore const non ha senso
  // per un metodo statico perchè il metodo
  // OraDiPranzo non ha un oggetto di invocazione !
  ...
};
```

```
orario orario::OraDiPranzo() {
  return orario(13,15);
};
```

Qualora chiamata esternamente, si deve utilizzare lo scoping.
Attenzione: non ha senso this nei metodi static, essendo riferimento costante solo all'oggetto attuale di invocazione.

```
cout << "Si pranza alle "
   << orario::OraDiPranzo().Ore() << " e "
   << orario::OraDiPranzo().Minuti() << " minuti\n";
```

L'inizializzazione dei campi dati statici si deve fare all'esterno della classe ed è sempre richiesta:

L'inizializzazione dei campi dati statici si deve fare all'esterno della classe ed è **sempre** richiesta.

Section 9.4.2, Static data members, of the C++ standard states:

If a `static` data member is of `const` integral or `const` enumeration type, its declaration in the class definition can specify a *const-initializer* which shall be an integral constant expression.

```
class orario {
public:
 ...
 static int Sec_di_una_Ora;
 static int Sec_di_un_Giorno;

 ...
};
```

```
// esternamente alla classe orario
int orario::Sec_di_una_Ora = 3600;
int orario::Sec_di_un_Giorno = 86400;
```

Unica copia in memoria dei campi dati statici

```
class orario {
public:
 ...
 static const int Sec_di_una_Ora = 3600;
 ...
};
```

Legale

Esempio utile: contare gli oggetti istanziati fino a quel momento.

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

class C {
  int dato; // privato
public:
  C(int); // costruttore ad un argomento
  static int cont; // campo dati statico pubblico
};

int C::cont = 0; // inizializzazione campo dati statico

C::C(int n) {cont++; dato=n;} //definizione costruttore

int main(){
  C c1(1), c2(2);
  cout << C::cont; // stampa: 2
}
```

Notare che per eseguire la somma degli orari potremmo pensare di definire un metodo *somma* come segue:

Metodo `orario somma(orario)`

```
orario orario::Somma(orario o) const {
  orario aux;
  aux.sec = (sec + o.sec) % 86400;
  // Notare che con o.sec si accede ad un campo
  // dati privato del parametro o
  return aux;
}
```

```
int main {
  ...
  orario ora(22,45);
  orario DUE_ORE_E_UN_QUARTO(2,15);
  ora = ora.Somma(DUE_ORE_E_UN_QUARTO);
  ...
}
```

Quello che conviene fare è il cosiddetto overloading, quindi ridefinire il comportamento di un operatore/metodo per quella apposita classe per riutilizzarlo sempre (da cui il sovraccarico/overloading), per tutte le situazioni di quel tipo.

```
class orario {
public:
  orario operator+(orario) const; // operator è una keyword
  ...
};
```

```
orario orario::operator+(orario o) const {
  orario aux;
  aux.sec = (sec + o.sec) % 86400;
  return aux;
}
```

```
int main {
  ....
  orario ora(22,45);
  orario DUE_ORE_E_UN_QUARTO(2,15);
  ora = ora + DUE_ORE_E_UN_QUARTO;
  ....
}
```

L'overloading degli operatori segue questa struttura e queste regole

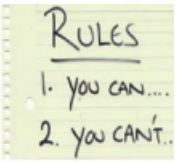
Overloading di un operatore OP

- **operatorOP**
- come metodo oppure come funzione esterna
- se è un metodo allora l'oggetto di invocazione è il primo argomento

Il C++ permette di sovraccaricare circa 40 operatori (unari e binari) tra i quali (lista completa anche su Wikipedia):

```
+ - * / % == != < <= > >= ++ --
<<  >> = -> [] () & new delete
```

Regole per l'overloading degli operatori



1) Non si possono cambiare:
- posizione (prefissa/infissa/postfissa)
- numero operandi
- precedenze e associatività

2) Tra gli argomenti deve essere presente almeno un tipo definito dall'utente

3) Gli operatori "**=**", "**[]**" e "**->**" si possono sovraccaricare solo come metodi (interni)

4) Non si possono sovraccaricare gli operatori "**.**", "**::**", "**sizeof**", "**typeid**", i cast e l'operatore condizionale ternario "**? :**"

5) Gli operatori "**=**", "**&**" e "**,**" hanno una versione standard

Caso particolare: operatore condizionale ternario.

Funziona così:
(condizione) ? vera-clausola : falsa-clausola
È più comunemente usato nelle operazioni di assegnazione, sebbene abbia anche altri usi. L'operatore ternario ? è un modo per abbreviare una clausola if-else e in altri linguaggi è chiamato anche istruzione immediate-if (IIf(condition,true-clause,false-clause) in VB, per esempio).

Ad esempio:

bool Three = SOME_VALUE;
int x = Three? 3 : 0;

è uguale a

bool Three = QUALCHE VALORE;
int x;
if (Three) x = 3;
else x = 0;

```
booleanExpr ? expr1 : expr2;
```

```
// ESEMPI

orario oraApertura = (day == SUNDAY) ? 15 : 9;

int max(int x, int y) {
  return x>y ? x : y;
}
```

Scritto da Gabriel

L'operatore virgola separa delle espressioni (i suoi argomenti) e ritorna il valore solamente dell'ultima espressione a destra, valutando tutte le altre espressioni per i loro effetti collaterali.

```
int main() {
  int a = 0, b = 1, c = 2, d = 3, e = 4;
  a = (b++, c++, d++, e++);
  cout << "a = " << a << endl; // stampa 4
  cout << "c = " << c << endl; // stampa 3
}
```

La ragione della stampa è semplice: *a* considera l'incremento di tutte le variabili, mentre *c* solo della propria.

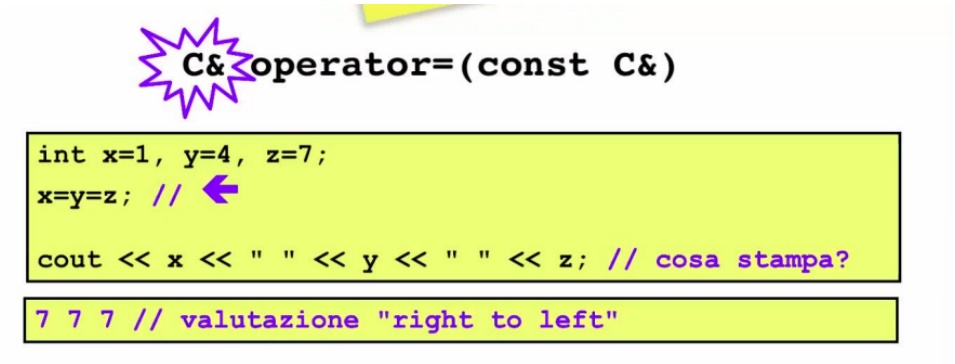
Scrivere una definizione degli operatori "-", "==", ">" e "<" per la classe **orario**. Scrivere una definizione degli operatori "+", "-", "==", ">" e "<" per la classe **complesso**.

```
bool orario::operator==(const orario& o)   //OPERATORE ==
{
   if(sec==o.sec)
      return true;
   else
      return false;
}

bool orario::operator>(orario o)      //OPERATORE >
{
   if(sec>o.sec)
      return true;
   else
      return false;
}

bool orario::operator<(orario o)   //OPERATORE <
{
   if(sec<o.sec)
      return true;
   else
      return false;
}

orario orario::operator-(orario o)     //OPERATORE -
{
   orario aux;
   aux.sec=(sec-o.sec)%86400;
   return aux;
}

bool operator==(complesso c){
      return (reale == c.reale && immaginario == c.immaginario);
   }
```

Scritto da Gabriel

```
bool operator-(complesso c){
   return (reale - c.reale && immaginario - c.immaginario);
}
bool operator>(complesso c){
   return (reale > c.reale && immaginario > c.immaginario);
}

bool operator <(complesso c){
   return (reale < c.reale && immaginario < c.immaginario);
}
bool operator+(complesso c){
   return (reale + c.reale && immaginario + c.immaginario);
}
```

Vediamo ora l'operatore di assegnazione, tale che prende di copia ogni singolo oggetto da sinistra a destra e vi assegna un valore specifico:



Altri esempi dello stesso tipo:

```
int x=1, y=4, z=7;
(x=y)=z; //

cout << x << " " << y << " " << z; // cosa stampa?
```

```
7 4 7 // valutazione "left to right"
```

```
int x=1, y=4, z=7;
x=y=z; //

cout << x << " " << y << " " << z; // cosa stampa?
```

7 7 7

```
int& f(int& a) {a=3; return a;}
int x=5, y=8;
f(y=x); //

cout << x << " " << y; // cosa stampa?
```

5 3

Il costruttore di copia *C(const C&)* viene invocato automaticamente in tre casi:

1 - quando un oggetto viene dichiarato ed inizializzato con un altro oggetto della stessa classe, come nei seguenti due casi:

```
orario adesso(14,30);
orario copia = adesso; // inizializza copia
orario copia1(adesso); // inizializza copia1
```

2 - quando un oggetto viene **passato per valore** come parametro di una funzione, come in:

```
ora = ora.Somma(DUE_ORE_E_UN_QUARTO);
```

dove `Somma` è la funzione:

```
orario orario::Somma(orario o) const {
  orario aux;
  aux.sec = (sec + o.sec) % 86400;
  return aux;
}
```

3 - quando una funzione **ritorna per valore** tramite l'istruzione `return` un oggetto, come in

```
return aux;
```

nella funzione precedente.

Attenzione al caso 3:
Il compilatore *g++* di default ottimizza evitando di creare un temporaneo anonimo usato solo per inizializzare un altro oggetti dello stesso tipo. Per evitare tale ottimizzazione, è possibile tramite un apposito *flag* di compilazione vedere (a fini didattici) l'uso di questi temporanei (normalmente comporta una/due stampe in più).
Il flag in questione è:

```
g++ -fno-elide-constructors
```

Modo semplice di usarlo (se si volesse):

- Il compiler online: https://www.onlinegdb.com/online_c++_compiler
- Espandere la finestra il larghezza e cliccare in alto a destra la rotellina delle impostazioni
- Selezionare "Extra Compiler Flags"
- Inserire il flag discusso

Esempi di uso:

```
C fun(C a) {return a;}

int main() {
  C c;
  fun(c);
  // 2 invocazioni del costr. di copia

  C y = fun(c);
  // g++ => 2 sole invocazioni del costr. di copia e non 3!

  C z; z = fun(c);
  // 2 invocazioni del costr. di copia

  fun(fun(c));
  // g++ => 3 sole invocazioni del costr. di copia e non 4!
}
```

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

class Puntatore {
public:
  int* punt;
};

int main() {
  Puntatore x, y;
  x.punt = new int(400);
  y = x;
  cout << "*(y.punt) = " << *(y.punt) << endl;
  *(y.punt) = 100;
  cout << "*(x.punt) = " << *(x.punt) << endl;
}
```

*(y.punt) = 400
*(x.punt) = 100

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

class Vettore {
public:
  int vett[10];
};

int main() {
  Vettore x, y;
  x.vett[2] = 200;
  y = x;
  cout << "y.vett[2] = " << y.vett[2] << endl;
  y.vett[2] = 0;
  cout << "x.vett[2] = " << x.vett[2] << endl;
}
```

y.vett[2] = 200
x.vett[2] = 200

(Ciascun elemento viene assegnato coerentemente col suo tipo)

The implicitly-defined assignment operator for class X performs memberwise assignment of its subobjects. ... Each subobject is assigned in the manner appropriate to its type:
...
-- if the subobject is an array, **each element is assigned**, in the manner appropriate to the element type

```
int main() {
  int a[10];
  int b[10];
  b=a; // ILLEGALE: array type int[10] is not assignable
}
```

Un altro operatore importante di cui fare overloading: quello di stampa.

```
std::ostream& orario::operator<<(std::ostream& os) const
{
  return os << Ore() << ':' << Minuti()
            << ':' << Secondi();
}
```

Vista così la funzione viene chiamata come funzionalità classica e in un modo poco comodo:

```
orario le_tre(15,0);
orario le_quattro(16,0);
le_quattro << ((le_tre << cout)
           << " vengono prima delle ");
```

Spiegazione: si tratta delle seguenti chiamate

```
le_quattro.operator<<(cout);
le_tre.operator<<(cout);
```

Vorremmo invece, come al solito:

```
cout << le_tre << " vengono prima delle " << le_quattro;
```

La definiamo quindi come funzione esterna.

Overloading di **<<** come **funzione esterna** alla classe:

```
ostream& operator<<(ostream& os,const orario& o) {
  return os << o.Ore() << ':' << o.Minuti()
            << ':' << o.Secondi();
}
```

```
orario le_tre(15,0), dodici(12,0);

cout << "Adesso sono le " << le_tre << endl;
// stampa: Adesso sono le 15:0:0

cout << "Fra dodici ore saranno le "
     << le_tre + dodici << endl;
// stampa: Fra dodici ore saranno le 3:0:0
```

Confrontano le due definizioni, abbiamo un accesso alla parte private con la prima definizione (chiamata del metodo), nel secondo caso abbiamo l'oggetto che di copia chiama ciò che gli serve:

```
ostream& orario::operator<<(ostream& os) const {
  return os << Ore() << ':' << Minuti()
            << ':' << Secondi();
}
```

```
ostream& operator<<(ostream &os,const orario &o) {
  return os << o.Ore() << ':' << o.Minuti()
            << ':' << o.Secondi();
}
```

Altro esempio:

```
orario orario::operator+(orario o) const; // operatore +
orario::orario(int o); // costruttore ad un parametro
```

Si riporta che nel primo dei due esempi sotto riportati, funziona il primo dei due casi, in quanto il secondo considera un oggetto costante di tipo *int* e non un oggetto di tipo orario (t), tale che non sia ammessa una conversione implicita (perché l'operatore "+" è definito come funzione interna).

```
orario t(12,20), s;
s = t + 4;  // OK
```

Invece non funziona

```
s = 4 + t;  // ERRORE: operatore non definito
```

Quando invece, l'operatore "+" è definito come funzione esterna, allora le cose cambiano (per il meglio):

Operatore + come funzione esterna

```
// operatore + esterno
orario operator+(const orario& t, const orario& s);
orario::orario(int o); // costruttore ad un parametro
```

```
orario t(12,20), s;
s = t + 4;  // OK:conversione del secondo parametro
```

```
s = 4 + t;  // OK:conversione del primo parametro
```

Naturalmente in entrambi i casi funziona

```
s = 4 + 5;  // OK: + tra interi e poi
            // conversione sul risultato
```

È poco stabile definire l'accesso alle variabili di orario con una semplice variabile che di copia utilizza i membri privati presenti come segue:

```
orario operator+(const orario& t, const orario& s)
{
  orario aux;
  aux.sec = (t.sec + s.sec) % 86400;
  return aux;
}
```

L'implementazione di basso livello utilizza l'interfaccia pubblica (con dei metodi get()) come segue, migliorando l'accesso alle variabili presenti:

```
orario operator+(const orario& t, const orario& s) {
  int sec = t.Secondi() + s.Secondi();
  int min = t.Minuti() + s.Minuti() + sec / 60;
  sec = sec % 60;
  int ore = t.Ore() + s.Ore() + min / 60;
  min = min % 60;
  ore = ore % 24;
  return orario(ore,min,sec);
}
```

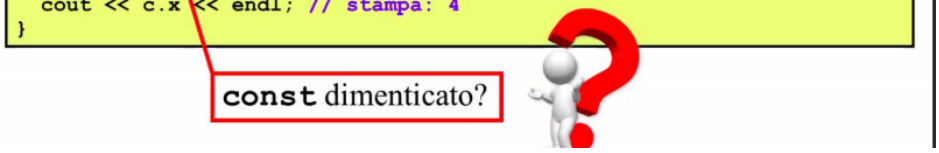
Ora, una serie di cosa stampa:

```
class C {
public:
  C() {}
  C(const C& r) {cout << "*";}
};

C f(C a) {
  C b(a);
  C c = b;
  return c;
}

int main() {
  C x;
  C y = f(f(x));
}
```

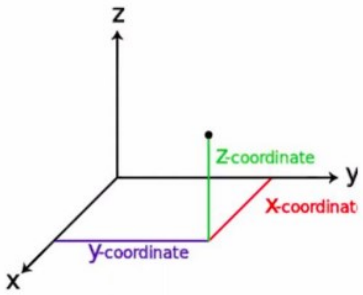
Stampa → * * * * *

In quest'altro metodo, il *const* non sarebbe necessario (in quanto *x* si riferisce ad un *int* e non ad un *const int*.

```
class C {
public:
  int x;

  C() { x=8; }

  void f(int& a) const { a=4; } // metodo costante

  void m() { f(x); }            // metodo non costante
};

int main() {
  C c;
  cout << c.x << endl; // stampa: 8
  c.m();
  cout << c.x << endl; // stampa: 4
}
```



Definire una classe `Point` i cui oggetti rappresentano punti nello spazio (x,y,z). Includere un costruttore di default, un costruttore a tre argomenti che inizializza un punto, selettori delle coordinate cartesiane, un metodo `negate()` che trasforma un punto nel suo negativo, una funzione `norm()` che restituisce la distanza del punto dall'origine, l'overloading degli operatori di somma e di output. Separare interfaccia ed implementazione della classe.

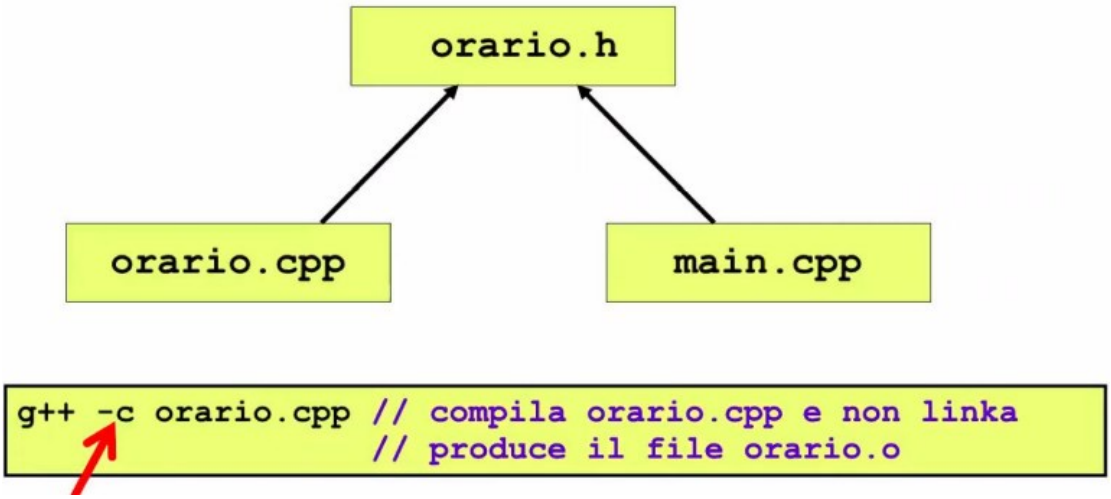


```
// file Point.h
#include<iostream>

class Point {
private:
  double _x, _y, _z;
public:
  // costruttore a 3-2-1-0 parametri
  Point(double x =0.0, double y =0.0, double z =0.0);
  double getX() const;
  double getY() const;
  double getZ() const;
  void negate();
  double norm() const;

};

Point operator+(const Point& p1, const Point& p2);

std::ostream& operator<<(std::ostream& os, const Point& p);

// file Point.cpp
// implementazioni
#include<cmath>
#include<iostream>
// #include "Point.h"

Point::Point(double x, double y, double z) {
  _x = x; _y = y; _z=z;
}

double Point::getX() const {return _x;}

double Point::getY() const {return _y;}

double Point::getZ() const {return _z;}

void Point::negate() {
  _x *= -1.0; _y *= -1.0; _z *= -1.0;
}

double Point::norm() const {
  return sqrt(_x*_x + _y*_y + _z*_z);
}

std::ostream& operator<<(std::ostream& os, const Point& p){
  return os << "(" << p.getX() << "," << p.getY() << ","
            << p.getZ() << ")";
}

Point operator+(const Point& p1, const Point& p2) {
  return Point(p1.getX()+p2.getX(),
               p1.getY()+p2.getY(),
```

```
        p1.getZ()+p2.getZ());
}
```

Modularizzazione dei programmi in file, Makefile, Modularizzazione delle classi, Relazione has-a

L'esempio della classe orario è un caso molto utile, infatti il comando compila orario e produce un file ".o", usato per la compilazione di file multipli. All'utilizzatore basta fornire il file "orario.h" da includere in "main.cpp" per la compilazione ed il file "orario.o" da aggiungere al codice compilato di "main.cpp" affinché il linker generi l'eseguibile.



Ora parliamo del preprocessore.

I preprocessori sono le direttive che danno istruzioni al compilatore per preelaborare le informazioni prima che inizi la compilazione vera e propria.
Tutte le direttive del preprocessore iniziano con # e prima di una direttiva del preprocessore in una riga possono comparire solo caratteri di spazio bianco. Le direttive del preprocessore non sono istruzioni C++, quindi non terminano con un punto e virgola (;).

La direttiva *#include* è già presente in tutti gli esempi. Questa macro viene utilizzata per includere un file di intestazione nel file sorgente.
Esistono numerose direttive del preprocessore supportate dal C++, come #include, #define, #if, #else, #line, ecc. Vediamo le direttive più importanti.

- Il *preprocessore #define*

La direttiva del preprocessore #define crea costanti simboliche. La costante simbolica è chiamata macro e la forma generale della direttiva è la seguente

#define macro-name "replacement-text"
Quando questa riga appare in un file, tutte le occorrenze successive di macro in quel file saranno sostituite da replacement-text prima che il programma venga compilato. Ad esempio

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

#define PI 3.14159
```

```
int main () {
  cout << "Valore di PI :" << PI << endl;

  restituire 0;
}
```

Ora, facciamo la preelaborazione di questo codice per vedere il risultato, supponendo di avere il file del codice sorgente. Compiliamolo quindi con l'opzione -E , poi reindirizziamo il risultato a test.p. Ora, se si controlla test.p, ci saranno molte informazioni e in fondo si troverà il valore sostituito nel modo seguente -

```
$gcc -E test.cpp > test.p

...
int main () {
  cout << "Valore di PI :" << 3,14159 << endl;
  restituire 0;
}
```

- *Macro simili a funzioni*

È possibile utilizzare #define per definire una macro che accetta gli argomenti come segue

```
#include <iostream>
usando lo spazio dei nomi std;

#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {
  int i, j;

  i = 100;
  j = 30;

  cout <<"Il minimo è " << MIN(i, j) << endl;

  return 0;
}
```

Se si compila e si esegue il codice sopra descritto, si otterrà il seguente risultato
Il minimo è 30

- *Compilazione condizionale*

Esistono diverse direttive che possono essere utilizzate per compilare porzioni selettive del codice sorgente del programma. Questo processo è chiamato compilazione condizionale.
Il costrutto del preprocessore condizionale è simile alla struttura di selezione 'if'. Considerate il seguente codice del preprocessore

```
#ifndef NULL
  #define NULL 0
#endif
```

Definiti alcuni casi d'uso dei preprocessore, parliamo del Makefile.
Make è uno strumento UNIX utilizzato per semplificare la creazione di eseguibili da diversi moduli di un progetto. Ci sono varie regole che vengono specificate come voci di destinazione nel makefile. Lo strumento make legge tutte queste regole e si comporta di conseguenza.

Ad esempio, se una regola specifica una dipendenza, lo strumento make la includerà ai fini della compilazione. Il comando make viene usato nel makefile per costruire moduli o per ripulire i file.

La sintassi generale di make è:
%make target_label (#target_label è un target specifico nel makefile)

- Makefile C++

Un makefile non è altro che un file di testo utilizzato o referenziato dal comando 'make' per costruire i target. Un makefile contiene anche informazioni come le dipendenze a livello di sorgente per ogni file e le dipendenze dell'ordine di compilazione. Vediamo ora la struttura generale di un makefile.

Un makefile inizia tipicamente con le dichiarazioni delle variabili, seguite da una serie di voci relative ai target per la costruzione di target specifici. Questi target possono essere file .o oppure altri file eseguibili in C o C++ e file .class in Java.
Si può anche avere un insieme di voci di target per l'esecuzione di un insieme di comandi specificati dall'etichetta del target.
Quindi un makefile generico è quello mostrato di seguito:

```
# commento

target: dipendenza1 dipendenza2 ... dipendenza
    comando <tab

# (nota: la <tab> nella riga di comando è necessaria per il funzionamento di make)
```

Un semplice esempio di makefile è mostrato di seguito.

```
# a build command to build myprogram executable from myprogram.o and
mylib.lib
all:myprogram.o mylib.o
     gcc –o myprogram myprogram.o mylib.o
clean:
     $(RM) myprogram
```

Nel makefile sopra riportato, abbiamo specificato due etichette di destinazione, la prima è l'etichetta 'all' per costruire l'eseguibile dai file oggetto myprogram e mylib. La seconda etichetta di destinazione 'clean' rimuove tutti i file con il nome 'myprogram'.

Vediamo un'altra variante del makefile.

```
# the compiler: gcc for C program, define as g++ for C++
 CC = gcc

  # compiler flags:
  #  -g     - this flag adds debugging information to the executable file
  #  -Wall  - this flag is used to turn on most compiler warnings
 CFLAGS  = -g -Wall

  # The build target
 TARGET = myprogram

 all: $(TARGET)
```

Scritto da Gabriel

```
$(TARGET): $(TARGET).c
	$(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c

clean:
	$(RM) $(TARGET)
```

Come mostrato nell'esempio precedente, in questo makefile si fa uso della variabile 'CC' che contiene il valore del compilatore che si sta utilizzando (GCC in questo caso). Un'altra variabile 'CFLAGS' contiene i flag del compilatore che utilizzeremo.

La terza variabile 'TARGET' contiene il nome del programma per il quale dobbiamo costruire l'eseguibile.

Il vantaggio di questa variante del makefile è che basta cambiare i valori delle variabili utilizzate ogni volta che si cambia il compilatore, i flag del compilatore o il nome del programma eseguibile.

Un makefile consiste di alcune regole così descritte:

```
TARGET : DEPENDENCIES ...
COMMAND
```

Di solito TARGET è il nome dell'eseguibile o del file oggetto da ricompilare, ma può essere anche una azione (es. **clean**). È una sorta di identificatore dell'azione da compiere: alla chiamata **bash$ make clean** verrà eseguito il TARGET ``**clean**".

Le DEPENDENCIES sono usate come input per generare l'azione TARGET e di solito sono più di una. Più genericamente vengono citati i file o le azioni, cioè i TARGET, da cui dipende il completamento dell'azione TARGET.

Un COMMAND è invece il comando da eseguire; può essere più di uno, e di solito si applica sulle DEPENDENCIES

Altro esempio (slide del prof):

```
# Commento: variabile CC è il comando che invoca il compilatore
CC=g++
# CCFLAGS: flags per il compilatore
CCFLAGS=-Wall -march=x86-64

my_prog : main.o kbd.o video.o help.o print.o
$(CC) $(CCFLAGS) -o my_prog main.o kbd.o video.o \\
help.o print.o

main.o : main.cpp config.h
$(CC) $(CCFLAGS) -c main.cpp -o main.o

kbd.o : kbd.cpp config.h
$(CC) $(CCFLAGS) -c kbd.cpp -o kbd.o

video.o : video.cpp config.h defs.h
$(CC) $(CCFLAGS) -c video.cpp -o video.o

help.o : help.cpp help.h
$(CC) $(CCFLAGS) -c help.cpp -o help.o

print.o : print.cpp
$(CC) $(CCFLAGS) -c print.cpp -o print.o

clean :
rm *.o
echo "pulizia completata"
```

Normalmente, per creare ed eseguire i nostri programmi utilizziamo i cosiddetti *IDE/Integrated Development Environment*. Un ambiente di sviluppo integrato (IDE) è una suite software che riunisce gli strumenti di base necessari per scrivere e testare il software.

Gli sviluppatori utilizzano numerosi strumenti per la creazione, la costruzione e il collaudo del codice software. Gli strumenti di sviluppo spesso includono editor di testo, librerie di codice, compilatori e piattaforme di test. Senza un IDE, uno sviluppatore deve selezionare, distribuire, integrare e gestire tutti questi strumenti separatamente. Un IDE riunisce molti di questi strumenti di sviluppo in un unico framework, applicazione o servizio. Il set di strumenti integrati è progettato per semplificare lo sviluppo del software e può identificare e ridurre al minimo gli errori di codifica e i refusi.

Alcuni IDE sono open source, mentre altri sono offerte commerciali. Un IDE può essere un'applicazione indipendente o può far parte di un pacchetto più ampio.

Caratteristiche [modifica | modifica wikitesto]

Normalmente è uno strumento software che consiste di più componenti, da cui appunto il nome *integrato*:

- un editor di codice sorgente;
- un compilatore e/o un interprete;
- un tool di building automatico;
- (solitamente) un debugger.

svn, git

A volte è integrato anche con un sistema di controllo di versione e uno o più tool per semplificare la costruzione di una GUI.

Alcuni IDE, rivolti allo sviluppo di software orientato agli oggetti, comprendono anche un navigatore di classi, un *analizzatore di oggetti* e un *diagramma della gerarchia delle classi*.

IDE noti ed usati:

- Eclipse (caso Java)
- NetBeans
- Visual Studio
- XCode (Apple)
- Qt Creator (usato da noi poveri cristi per il progetto)

Memory leak

Essi si verificano in C++ quando i programmatori allocano la memoria usando la parola chiave new e dimenticano di deallocarla usando la funzione delete() o l'operatore delete[]. Una delle maggiori perdite di memoria si verifica in C++ con l'uso di un operatore delete errato.
L'operatore delete dovrebbe essere usato per liberare un singolo spazio di memoria allocato, mentre l'operatore delete[] dovrebbe essere usato per liberare un array di valori di dati.
Svantaggio della perdita di memoria:
Se un programma ha perdite di memoria, l'utilizzo della memoria aumenta in modo satirico, poiché tutti i sistemi hanno una quantità limitata di memoria e la memoria è costosa. Di conseguenza, si creano problemi.

Uno strumento utile per trovarli è *Valgrind*, noto come strumento per trovare errori di lavoro con la memoria. Ma oltre a questo, contiene anche una serie di utilità aggiuntive per la profilazione delle prestazioni, la ricerca di errori di sincronizzazione nei programmi multithreading e l'analisi del consumo di memoria. È utile parlarne in quanto integrato in Qt Creator e nei vari IDE.

Un concetto molto utile in programmazione ad oggetti è l'esempio delle *relazioni is-a* e le *relazioni has-a*. Comprendiamo le relazioni IsA e HasA in C++ con alcuni esempi. È utile per comprendere gli specificatori di accesso o l'accessibilità dei membri di una classe. Per questo motivo, date un'occhiata all'esempio seguente.

```
class Rettangolo{
    Some data members
    Some function members
}
```

Questa è una classe chiamata Rettangolo. Supponiamo che i dati abbiano alcuni membri all'interno di questa classe. Poi, abbiamo una classe Cuboid che eredita dalla classe Rectangle come segue.

```
Class Cuboide : public Rettangolo{
    Some data members
    Some function members
}
```

Questa è la classe Cuboid che eredita dalla classe Rectangle. Si supponga inoltre che anche questa classe abbia alcuni membri al suo interno. Ora scriviamo un'altra classe come segue.

```
class Tabella{
     Rettangolo top;
     legs Int;
}
```

Questa è la classe Tabella, che non è ereditata da nessuna classe. Questa classe ha due membri di dati: l'oggetto *top* della classe Rectangle e la variabile di tipo intero *legs*.

Scritto da Gabriel

Per prima cosa abbiamo creato una classe chiamata Rettangolo. Poi abbiamo creato un'altra classe chiamata Cuboide, ereditata dalla classe Rettangolo, e quindi abbiamo creato un'altra classe chiamata Tavolo. All'interno della classe Tabella, abbiamo creato un oggetto di tipo Rettangolo e una variabile intera. La classe Cuboide è ereditata dalla classe Rettangolo. Possiamo quindi dire che un cuboide è un rettangolo? Si. Quindi, la relazione tra la classe Rettangolo e la classe Cuboide è la relazione "is-a".

Poi, la nostra classe Tavolo ha un piano d'appoggio di tipo Rettangolo. La classe Tavolo ha un oggetto della classe Rettangolo. Possiamo quindi dire che la classe Tavolo ha un Rettangolo? Sì, la classe Tavolo ha un Rettangolo. Quindi, la relazione tra la classe Tabella e la classe Rettangolo è la relazione 'Ha A'.

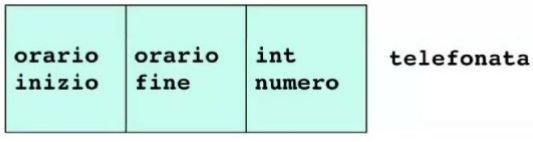
Quindi, possiamo usare la nostra classe in due modi: 'is-a' e 'has-a'. Questo è comune nella programmazione orientata agli oggetti, ma non nel C++. Una classe può essere utilizzata in due modi. Uno è che una classe può essere derivata, il che significa che possiamo scrivere classi figlie che ereditano da quella classe. Il secondo è che si possono usare gli oggetti di quella classe. Quindi, ci sono due modi per utilizzare una classe. O si può creare l'oggetto e usarlo o si può ereditare da quella classe.

La relazione has-a è utile per descrivere le caratteristiche di una classe specifica.

Nel nostro esempio, ci può essere una classe *Telefonata* che *possiede* l'orario come dettaglio implementativo. Vediamo anche cosa possiede in memoria in quel momento.

```
// file telefonata.h
#ifndef TELEFONATA_H
#define TELEFONATA_H
#include <iostream>
#include "orario.h"

class telefonata {
private:
  orario inizio, fine; // relazione has-a
  int numero;
public:
  telefonata(orario,orario,int); //p.valore (per ora)
  telefonata();
  orario Inizio() const;
  orario Fine() const;
  int Numero() const;
  bool operator==(const telefonata&) const;
};

ostream& operator<<(ostream&,const telefonata&);
#endif
```

In memoria



Descriviamo l'andamento della memoria per un motivo fondamentale: definire il comportamento del costruttore.

Comportamento del costruttore

Consideriamo una classe **C** con campi dati **x1,...,xk** di qualsiasi tipo, possibilmente qualche altra classe. L'ordine dei campi dati è determinato dall'ordine in cui essi appaiono nella definizione della classe **C**.

Supponiamo di definire nella classe **C** un qualsiasi costruttore come segue:

```
C(Tipo_1,...,Tipo_n) {//codice}
```

Il comportamento di tale costruttore è il seguente:

(1) Per ogni campo dati **xj** di **tipo non classe Tj** (ovvero di tipo primitivo o derivato), viene allocato un corrispondente spazio in memoria per contenere un valore di tipo **Tj** ma il valore viene lasciato indefinito;

(2) Per ogni campo dati **xi** di **tipo classe Ti** viene invocato il costruttore di default **Ti()**;

(3) Alla fine, viene eseguito il **codice** del corpo del costruttore.

Nota: i punti (1) e (2) vengono eseguiti per tutti i campi dati **x1,...,xk** seguendo il loro ordine di dichiarazione.

A seguito di questo, un cosa stampa e un esempio in cui manca un chiaro comportamento di un oggetto B e una giusta definizione del costruttore standard

```
class C {
private:
  int x;
public:
  C() {cout << "C0 "; x=0;}
  C(int k) {cout << "C1 "; x=k;}
};
class D {
private:
  C c;
public:
  D() {cout << "D0 "; c = C(3);}
};
class E {
private:
  char c;
  C c1;
public:
  D d;
  C c2;
};
int main(){
  E x; cout << "UNO\n";
  D y; cout << "DUE\n";
  E* p = &x; cout << "TRE\n";
  D& a = y; cout << "QUATTRO";
}
```

Stampe:
C0 C0 D0 C1 C0 UNO
C0 D0 C1 DUE
TRE
QUATTRO

```
class B {
private:
  int x;
public:
  B(int a) {x=a;}
};

class C {
private:
  int x;
  B b;
public:
  C() {x=5; b=B(4)}
};

int main(){
  C z;
}
// ERRORE in compilazione:
// In method C::C() no matching function for call
// to B::B(). Candidates are: B::B(int)  B::B(const B&)
```

In *telefonata*, quindi:

```
// file telefonata.cpp
#include "telefonata.h"

telefonata::telefonata(orario i,orario f, int n) {
  inizio = i;
  fine = f;
  numero = n;
}

telefonata::telefonata() {numero = 0;}
// Attenzione: gli altri campi dati di tipo orario vengono
// inizializzati tramite il costruttore di default

orario telefonata::Inizio() const {return inizio;}

orario telefonata::Fine() const {return fine;}

int telefonata::Numero() const {return numero;}
```

```
bool telefonata::operator==(const telefonata& t) const {
  return inizio == t.inizio &&
         fine == t.fine &&
         numero == t.numero;
}
```

```
bool telefonata::operator==(const telefonata& t) const {
  return inizio == t.inizio &&
         fine == t.fine &&
         numero == t.numero;
}

ostream& operator<<(ostream& s, const telefonata& t) {
  return s << "INIZIO " << t.Inizio()
           << " FINE " << t.Fine()
           << " NUMERO CHIAMATO " << t.Numero();
}
```

Esercizio

La classe **`telefonata`** presenta l'inconveniente di non poter rappresentare numeri telefonici che iniziano con lo 0.

Scegliere una **rappresentazione alternativa** per il numero telefonico che risolva l'inconveniente e riscrivere le definizioni dei metodi per tale rappresentazione.

Versione definitiva di telefonata

Telefonata.h

```
#ifndef telefonata_h
#define telefonata_h

#include <iostream>
#include "orario.h"

using std::ostream;

class telefonata {
public:
   telefonata(orario,orario,int);
   telefonata();
   orario Inizio() const;
   orario Fine() const;
   int Numero() const;
   bool operator==(const telefonata&)
const;
private:
   orario inizio, fine;
   int numero;
};

ostream& operator<<(ostream&, const telefonata&);

#endif
```

Invocazione **esplicita** del "costruttore di copia" per il campo dati costante `numero`.

```
telefonata::telefonata() : numero(0) {}
```

LEGAL

È possibile richiamare esplicitamente un costruttore per qualsiasi campo dati:

```
telefonata::telefonata(orario i, orario f, int n)
                      : inizio(i), fine(f), numero(n) {}
```

Telefonata.cpp

```
#include "telefonata.h"

telefonata::telefonata(orario i, orario f, int n) {
   inizio = i; fine = f; numero = n;
```

```
}
telefonata::telefonata() {numero = 0;}

orario telefonata::Inizio() const {return inizio;}
orario telefonata::Fine() const {return fine;}
int telefonata::Numero() const {return numero;}

bool telefonata::operator==(const telefonata& t) const {
   if (inizio == t.inizio && fine == t.fine && numero == t.numero) {
      return true;
   }
   else return false;
}

ostream& operator<<(ostream& s, const telefonata& t) {
   return s << "Inizio " << t.Inizio() << " Fine " << t.Fine() << " Numero " << t.Numero();
}
```

Problema dell'inclusione multipla di file header

Il problema è che il file di intestazione è incluso (direttamente e indirettamente) nella stessa unità di traduzione. Si dovrebbe usare un modo per evitare l'inclusione multipla dello stesso file di intestazione nei cpp. Preferisco #pragma una volta all'inizio del file di intestazione: non è uno standard, ma è supportato da tutti i principali compilatori. Altrimenti si può optare per le buone vecchie protezioni per gli include:

```
#ifndef C.h
#define C.h
// Il contenuto dell'header è qui
#endif
```

o con pragma (serve allo stesso scopo):

```
#pragma once
// Il contenuto dell'intestazione qui
```

```
// file "C.h"
class C {
public:
  int x;
  C(int k=4) {x=k;}
};
```

```
// file "D.h"
#include<iostream>
#include "C.h"

class D {
public:
  int x;
  D(int k=6) {x=k;}
  void print(const C& c) const {std::cout << x + c.x;}
};
```

```
// file "main.cpp"
#include "C.h"
#include "D.h"

int main() {
  C c(3); D d(4);
  d.print(c);
}
// main.cpp non compila!
// g++: "redefinition of class C"
```

Una buona pratica (inclusa già di default alla creazione dei file header in Qt Creator), quindi, sono le cosiddette *direttive di compilazione*.

Operatore molto utile è il cosiddetto *defined*, che verifica se un simbolo è stato definito o meno.

```
#ifdef identificatore

#ifndef identificatore

// sono equivalenti a

#if defined identificatore

#if !defined identificatore
```

```
#ifdef identificatore
  text
#endif

#ifndef identificatore
  text
#endif

#ifdef identificatore1
  text1
#elif defined identificatore2
  text2
#endif
```

Le direttive stabiliscono anche specifici comportamenti tra i vari sistemi operativi, ad esempio:

```
#define LINUX
#ifdef LINUX
// istruzioni per versione Linux
#elif defined MacOS
// istruzioni per versione MacOS
#endif
```

Una precisazione importante: nell'esempio di *telefonata*, si ha il concetto di lista di inizializzazione, vale a dire, l'insieme dei parametri (nell'esatto ordine in cui si presentano) del costruttore.

Scritto da Gabriel

Essa inizia con i due punti e presenta una inizializzazione membro a membro di ogni parametro del costruttore.

```
telefonata::telefonata(orario i, orario f, int n)
                      : inizio(i), fine(f), numero(n) {}
```

La differenza nel costruttore di *telefonata* con la creazione del costruttore di copia è evidente:

- Creazione safe dei parametri
- Non c'è spreco di memoria con variabili locali
- Va bene perché non ci sono campi costanti

Nel primo costruttore (peggiore):

- 2 costruttori di copia di orario, 1 di copia per int
- 2 costruttori di default per orario, 1 di default per int
- 2 assegnazioni di default di orario, 1 costr. di default tra int

Come vengono eseguite le seguenti istruzioni:

```
orario t1(12,25,33), t2(12,47,12);
telefonata telefonata_a_carlo(t1,t2,333333);
```

usando il precedente costruttore

```
telefonata::telefonata(orario i, orario f, int n)
  {inizio = i; fine = f; numero = n; }
```

e usando il nuovo costruttore

```
telefonata::telefonata(orario i, orario f, int n)
            : inizio(i), fine(f), numero(n) {}
```

nell'ipotesi che non ci siano campi costanti e quindi siano permesse entrambe le definizioni.

Nel secondo costruttore (migliore):

- 2 costr. di default di orario, 1 di copia di int
- 2 costr. di copia di orario, 1 di copia per int

La cosiddetta lista di inizializzazione viene qui riportata, con la successiva logica di uso e costruzione:

Passiamo i parametri come riferimenti costanti.

```
telefonata::telefonata(const orario& i,
            const orario& f, const int& n)
            : inizio(i), fine(f), numero(n) {}
```

Ciò corrisponde quindi alle seguenti istruzioni:

```
orario inizio(t1); orario fine(t2); int numero(333333);
```

Lista di inizializzazione del costruttore.

In una classe **C** con lista ordinata di campi dati **x1,...,xk**, un costruttore con lista di inizializzazione per i campi dati $x_{i1},...,x_{ij}$ è definito tramite la seguente sintassi:

```
C(T1,...,Tn): xi1(...),..., xij(...) {\\ codice };
```

Il comportamento del costruttore è il seguente:

(A) Ordinatamente per ogni campo dati **xi** ($1 \le i \le k$) viene richiamato un costruttore:
- o esplicitamente tramite una chiamata ad un costruttore **xi(...)** definita nella lista di inizializzazione
- oppure implicitamente (non appare nella lista di inizializzazione) tramite una chiamata al costruttore di default **xi()**

(B) Quindi viene eseguito il codice del costruttore.

Alcune precisazioni da notare:

[1] Naturalmente, parliamo di costruttori e costruttori di copia anche per campi dati di tipo non classe (primitivo o derivato): la lista di inizializzazione può includere inizializzazioni anche per questi campi dati.

[2] La chiamata implicita al costruttore di default standard per un campo dati di tipo non classe, come al solito, alloca lo spazio in memoria ma lascia indefinito il valore.

[3] L'ordine in cui vengono invocati i costruttori, esplicitamente o implicitamente, è sempre determinato dalla lista ordinata dei campi dati, qualsiasi sia l'ordine delle chiamate nella lista di inizializzazione.

Un altro esempio pratico con un bel cosa stampa:

```
class D {
public:
  D() {cout << "D0 ";}
  D(int a) {cout << "D1 ";}
};
class E {
private:
  D d;
public:
  E(): d(3) {cout << "E0 ";}
  E(double a, int b) {cout << "E2 ";}
  E(const E& a): d(a.d) {cout << "Ec ";}
};
class C {
private:
  int z;
  E e;
  D d;
  E* p;
public:
  C(): p(0), e(), z(4) {cout << "C0 ";}
  C(int a, E b): e(3.7,2), p(&b), z(1), d(3) {cout << "C1 ";}
  C(char a, int b): e(), d(2), p(&e)  {cout << "C2 ";}
};
int main() {
  E e; cout << "UNO\n";
  C c; cout << "DUE\n";
  C c1(1,e); cout << "TRE\n";
  C c2('b',2); cout << "QUATTRO";
}
```

Stampe:
D1 E0 UNO
D1 E0 DO C0 DUE
Ec DO E2 D1 C1 TRE
D1 E0 D1 C2 QUATTRO

Usando invece campi *di tipo riferimento T&*, si verifica sul seguente frammento di codice, la definizione *di un costruttore di default legale per C (richiesta).*

```
class E {
private:
  int x;
public:
  E(int z=0): x(z) {}
};

class C {
private:
  int z;
  E x;
  const E e;
  E& r;
  int* const p;
  public:
  C();
```

Soluzione

```
C::C(): e(1), r(x), p(new int(0)) {}
```

Esercizio: confrontare gli output dei seguenti due programmi.

```
class C {
public:
  C() {cout << "C0 "; }
  C(const C&) {cout <<"Cc ";}
};

class D {
public:
  C c;
  D() {cout << "D0 ";}
};

int main(){
  D x; cout << endl;
  // stampa ...
  D y(x); cout << endl;
  // stampa ...
}
```

```
class C {
public:
  C() {cout << "C0 "; }
  C(const C&) {cout <<"Cc ";}
};

class D {
public:
  C c;
  D() {cout << "D0 ";}
  D(const D&) {cout <<"Dc ";}
};

int main(){
  D x; cout << endl;
  // stampa ...
  D y(x); cout << endl;
  // stampa ...
}
```

Stampe:

Codice a sx
C0 D0
Cc

Codice a dx
C0 D0
C0 Dc

Spiegazione: il costruttore di copia standard di una classe **D** invoca ordinatamente per ogni campo dati **x** di **D** il corrispondente costruttore di copia del tipo di **x**.

La struttura standard di costruttori di copia e standard è quindi:

Costruttori standard

Possiamo quindi specificare precisamente il comportamento dei costruttori standard di default e di copia per una classe **C** con lista ordinata di campi dati **x1,...,xk**

Costruttore di default standard:

```
C(): x1(),...,xk() {}
```

Costruttore di copia standard:

```
C(const C& obj): x1(obj.x1),...,xk(obj.xk) {}
```

Esercizio evergreen: Razionali

Esercizio

Definire, separando interfaccia ed implementazione, una classe `Raz` i cui oggetti rappresentano un numero razionale $\frac{num}{den}$ (naturalmente, i numeri razionali hanno sempre un denominatore diverso da 0). La classe deve includere:

1. opportuni costruttori;
2. un metodo `Raz inverso()` con il seguente comportamento: se l'oggetto di invocazione rappresenta $\frac{n}{m}$ allora `inverso` ritorna un oggetto che rappresenta $\frac{m}{n}$;
3. un operatore esplicito di conversione al tipo primitivo `double`;
4. l'overloading degli operatori di somma e moltiplicazione;
5. l'overloading dell'operatore di incremento postfisso che, naturalmente, dovrà incrementare di 1 il razionale di invocazione;
6. l'overloading dell'operatore di uguaglianza;
7. l'overloading dell'operatore di output su `ostream`;
8. un metodo `Raz unTerzo()` che ritorna il razionale 0.3333...

```
#include <iostream>
#include <cmath>

class Raz {
private:
  int num, den; // INV globale: den!=0, numero razionale num/den

  // riduzione naive
  void reduce() {
    if(num%den==0){ num=num/den; den=1;}
    else if(den%num==0){den=den/num; num=1;}
    else {
      int i;
      if(abs(num)<abs(den)) i=num/2;
      else i=den/2;
      do {
        if(num%i==0 && den%i==0){num=num/i; den=den/i;}
        i--; // naive
      } while(i>=2);
    }
  }

public:
  // conversione int => Raz bloccata da explicit
  explicit Raz(int n=0, int d=1): num(d==0 ? 0 : n), den(d==0 ? 1 : d) {}

  operator double() const {
    return static_cast<double>(num)/static_cast<double>(den);
  }

  Raz inverso() const {
    return Raz(den,num); // inverso di Raz(0,1) e' Raz(0,1)
  }

  Raz operator+(const Raz& r) const {
    return Raz(num*r.den + den*r.num,den*r.den);
  }

  Raz operator*(const Raz& r) const {
    return Raz(num*r.num,den*r.den);
  }

  Raz operator++(int) { // incremento postfisso
    Raz loc(*this);
    num += den;
    return loc;
  }
```

Scritto da Gabriel

```
  Raz& operator++()  {  // incremento prefisso
    num += den;
    return *this;
  }

  bool operator==(const Raz& r) const {
    return num*r.den == r.num*den;
  }

  static Raz unTerzo() {
    return Raz(1,3);
  }
};

std::ostream& operator<<(std::ostream& os, const Raz& r) {
  return os << "il razionale come double è: " << r.operator double();
}


int main() {
  Raz x(2,3), y(2), z, u(1,8), v(3,2), w(8,4);

  std::cout << x+y+v*u << std::endl; // 2.85417
  std::cout << u.inverso() << std::endl; // 8
  std::cout << (y == w) << std::endl; // true
  std::cout << y++ << " " << ++w << std::endl; // 2 3
  std::cout << (++y + Raz::unTerzo()) << std::endl; // 4.33333
  std::cout << 2 + Raz(1,2) << std::endl; // 2.5
}
```

Container, interferenza, shallow/deep copy, distruttori e distruttori profondi, lifetime, array

In informatica, un *contenitore* è una classe o una struttura di dati le cui istanze sono collezioni di altri oggetti. In altre parole, memorizzano gli oggetti in modo organizzato, seguendo specifiche regole di accesso.
La dimensione del contenitore dipende dal numero di oggetti (elementi) che contiene. Le implementazioni sottostanti (ereditate) dei vari tipi di contenitore possono variare in termini di dimensioni, complessità e tipo di linguaggio, ma in molti casi offrono flessibilità nella scelta dell'implementazione giusta per ogni scenario.
Le strutture dati dei contenitori sono comunemente utilizzate in molti tipi di linguaggi di programmazione.

I contenitori possono essere caratterizzati dalle seguenti tre proprietà:

- accesso, ovvero il modo di accedere agli oggetti del contenitore. Nel caso degli array, l'accesso avviene con l'indice dell'array. Nel caso delle pile, l'accesso avviene secondo l'ordine LIFO (last in, first out) e nel caso delle code secondo l'ordine FIFO (first in, first out);
- storage, cioè il modo di memorizzare gli oggetti del contenitore;
- attraversamento, cioè il modo di attraversare gli oggetti del contenitore.

Le classi di contenitori devono implementare metodi simili a *CRUD* (create, read, update, delete) per fare quanto segue:

- creare un contenitore vuoto (costruttore);
- inserire oggetti nel contenitore
- cancellare oggetti dal contenitore;
- cancellare tutti gli oggetti del contenitore (clear);

Scritto da Gabriel

- accedere agli oggetti del contenitore;
- accedere al numero di oggetti presenti nel contenitore (count).

I contenitori sono talvolta implementati insieme agli iteratori (oggetti che li scorrono)
Nella libreria STL ci sono vari tipi di contenitori (per comodità ben riportati):

1) Contenitori di sequenze

I contenitori di sequenze implementano strutture di dati a cui si può accedere in modo sequenziale.

- array: Array contiguo statico (modello di classe).
- vettore: Array contiguo dinamico (modello di classe)
- deque: Coda a doppia estremità (modello di classe)
- forward_list: Elenco collegato singolarmente (modello di classe)
- list: Lista doppiamente collegata (modello di classe)

2) Contenitori associativi

I contenitori associativi implementano strutture di dati ordinati che possono essere ricercati rapidamente (complessità O(log n)).

- Set: Collezione di chiavi uniche, ordinate per chiavi
- (modello di classe)
- Mappa: Collezione di coppie chiave-valore, ordinate per chiavi, le chiavi sono uniche (modello di classe).
- multiset: Collezione di chiavi, ordinate per chiavi (modello di classe).
- multimap: Raccolta di coppie chiave-valore, ordinate per chiavi (modello di classe)

3) Contenitori associativi non ordinati

I contenitori associativi non ordinati implementano strutture di dati non ordinati (hashed) che possono essere ricercati rapidamente (complessità O(1) ammortizzata, O(n) nel caso peggiore).

- unordered_set: Collezione di chiavi uniche, con hash per chiavi. (modello di classe)
- unordered_map: Collezione di coppie chiave-valore, hash per chiavi, le chiavi sono uniche. (modello di classe)
- unordered_multiset: Collezione di chiavi, con hash per chiavi (modello di classe)
- unordered_multimap: Collezione di coppie chiave-valore, hash per chiavi (modello di classe)

4) Adattatori di contenitori

Gli adattatori di contenitori forniscono un'interfaccia diversa per i contenitori sequenziali.

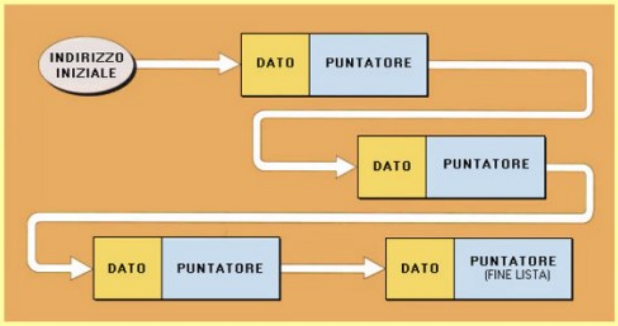
- stack: Adatta un contenitore per fornire stack (struttura dati LIFO) (modello di classe).
- coda: Adatta un contenitore per fornire una coda (struttura dati FIFO) (modello di classe).
- priority_queue: Adatta un contenitore per fornire una coda di priorità (modello di classe).

Possiamo usare queste conoscenze per espandere la classe *bolletta* aggiungendo funzionalità da container (rimozione/aggiunta telefonata, verifica se vuoto, ecc.):

```
class bolletta {
public:
  bolletta();
  bool Vuota() const;
  void Aggiungi_Telefonata(const telefonata&);
  void Togli_Telefonata(const telefonata&);
  telefonata Estrai_Una();
```

Proviamo a creare una *linked list,* struttura dati evergreen.
Una linked list è una struttura di dati lineare, in cui gli elementi non sono memorizzati in posizioni di memoria contigue. Gli elementi di un elenco collegato sono collegati mediante puntatore.
In parole povere, un elenco collegato è costituito da nodi in cui ogni nodo contiene un campo dati e un riferimento (link) al nodo successivo dell'elenco.

```
class bolletta {
public:
  bolletta();
  bool Vuota() const;
  void Aggiungi_Telefonata(const telefonata&);
  void Togli_Telefonata(const telefonata&);
  telefonata Estrai_Una();
private:
    ....
    ...
    ....
```




```
};
#endif
```

```
#ifndef BOLLETTA_H          // file bolletta.h
#define BOLLETTA_H
#include "telefonata.h"
class bolletta {
public:
  bolletta();
  bool Vuota() const;
  void Aggiungi_Telefonata(const telefonata&);
  void Togli_Telefonata(const telefonata&);
  telefonata Estrai_Una();
private:
  class nodo {          // nodo: classe interna privata
  public:
    nodo();
    nodo(const telefonata&, nodo*);
    telefonata info;
    nodo* next;
  };
  nodo* first; // puntatore al primo nodo della lista
};
#endif
```

```
// file bolletta.cpp
#include "bolletta.h"

bolletta::nodo::nodo() : next(0) {}
    // costruttore di default per il campo dati info

bolletta::nodo::nodo(const telefonata& t, nodo* s)
                     : info(t), next(s) {}

bolletta::bolletta(): first(0) {}

bool bolletta::Vuota() const {
  return first == 0;
}

void bolletta::Aggiungi_Telefonata(const telefonata& t){
  first = new nodo(t,first);
  // aggiunge in testa alla lista
}
```

```
void bolletta::Togli_Telefonata(const telefonata& t) {
  nodo* p = first, *prec = nullptr;
  while (p && !(p->info == t)) {
    prec = p;
    p = p->next;
  } // p==0 (not found) o p punta al nodo da rimuovere
  if (p) { // ho trovato t
    if (!prec) // p punta al primo nodo
      first = p->next;
    else // p punta ad un nodo successivo al primo
      prec->next = p->next;
    delete p; // attenzione: deallocare!
  }
}
```

```
telefonata bolletta::Estrai_Una() {
  //Precondizione: bolletta non vuota (!(first==0))
  nodo* p = first;
  first = first->next;
  telefonata aux = p->info; // costruttore di copia
  delete p;                 // attenzione: deallocare!
  return aux;
}
```

Attenzione:
Aggiungi_Telefonata() e Togli_Telefonata() possono provocare side-effects alla bolletta di invocazione.

```
int main() {
  bolletta b1; // costruttore senza argomenti

  telefonata t1(orario(9,23,12),orario(10,4,53),2121212);
  telefonata t2(orario(11,15,4),orario(11,22,1),3131313);

  b1.Aggiungi_Telefonata(t1);
  b1.Aggiungi_Telefonata(t2);
  cout << b1; // supponiamo di avere l'output di bolletta

  bolletta b2;
  b2 = b1;

  b2.Togli_Telefonata(t1);
  cout << b1 << b2;
}
```

```
TELEFONATE IN BOLLETTA: (NB: b1)
1) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313
2) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212

// dopo b2.Togli_Telefonata(t1);

TELEFONATE IN BOLLETTA: (NB: b1)
1) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313

TELEFONATE IN BOLLETTA: (NB: b2)
1) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313
```

Si porta all'attenzione del lettore quello che succede qui sotto: il problema dell'aliasing.

L'aliasing dei puntatori è un tipo di dipendenza nascosta dai dati che può verificarsi in C, C++ o in qualsiasi altro linguaggio che utilizza i puntatori per gli indirizzi degli array nelle operazioni aritmetiche. I dati dell'array identificati dai puntatori in C possono sovrapporsi, perché il linguaggio C pone pochissime restrizioni sui puntatori. Due puntatori apparentemente diversi possono puntare a posizioni di memoria dello stesso array (aliasing). Di conseguenza, quando si eseguono calcoli basati su loop utilizzando i puntatori, possono verificarsi dipendenze dai dati, poiché i puntatori possono potenzialmente puntare a regioni della memoria che si sovrappongono.

Tuttavia, affinché la vettorizzazione abbia luogo, il compilatore deve dimostrare che non sono possibili dipendenze di dati (sovrapposizioni). Questo è difficile da fare quando si usano i puntatori, il che significa che il compilatore è spesso costretto a essere conservativo sulla vettorizzazione. (Fortran 90/95 è meno vulnerabile a questo problema perché un puntatore deve essere esplicitamente associato a una variabile di destinazione di dimensioni note prima di poter essere utilizzato).

A titolo di esempio, si consideri la seguente funzione C. È sicuro vettorializzare il ciclo for?

```
void compute(double *a, double *b, double *c) {
   for (i=1; i<N; i++) {
      a[i] = b[i] + c[i];
   }
}
```

Scritto da Gabriel

A prima vista, questo sembra essere un ciclo semplice, facilmente vettorializzabile e senza apparenti dipendenze dai dati. Tuttavia, cosa succederebbe se lo invocassimo come segue:

```
/* Dati gli array p e q */
compute(p, p-1, q);
```

Nella nostra funzione compute(), i puntatori a e b si riferiscono entrambi a regioni sovrapposte di p. In effetti, b è solo p sfalsato di un indice di array. L'i-esimo elemento di a corrisponde all'i-esimo elemento di p. L'i-esimo elemento di b corrisponde (e funge da alias) all'i-1esimo elemento di p. Di conseguenza, possiamo sostituire questi valori e costruire un ciclo equivalente:

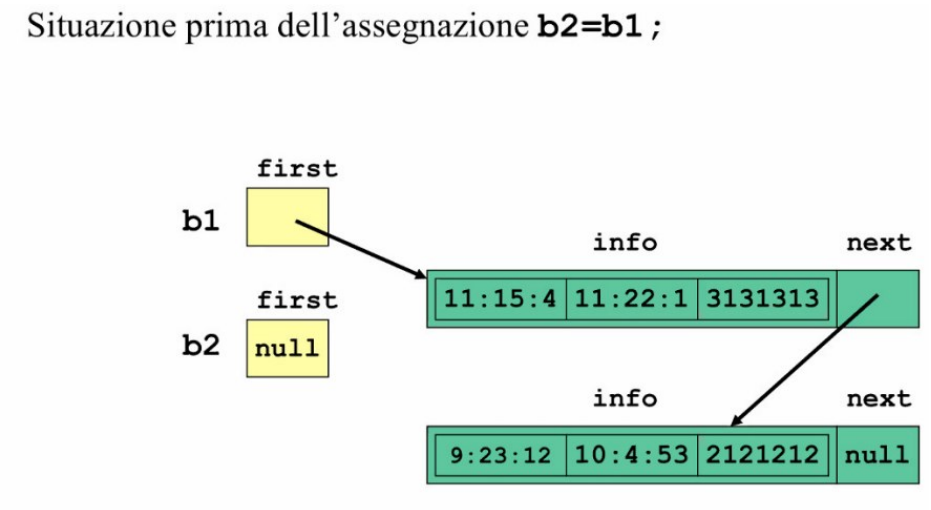
```
for (i=1; i<N; i++) {
    p[i] = p[i-1] + q[i];
}
```

È ora evidente che il ciclo ha una dipendenza di lettura e scrittura. Come abbiamo visto in precedenza, ciò significa che il ciclo non è vettoriale.

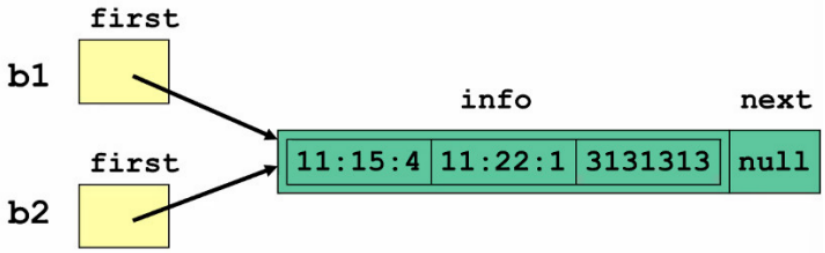
I puntatori rendono chiaramente più difficile il lavoro del compilatore nel dedurre la presenza di potenziali dipendenze, a causa dell'effetto aliasing. Come vedremo in una sezione successiva sui suggerimenti (o garanzie) del compilatore, il programmatore può risolvere questo problema contrassegnando il codice con la parola chiave *restrict*, per assicurare al compilatore che gli argomenti dei puntatori non si riferiscano mai a regioni di memoria sovrapposte.

I compilatori moderni sono spesso intelligenti per i cicli semplici come quello sopra descritto. Possono interrogare l'indirizzo iniziale e l'indirizzo finale (ad esempio, a[1] e a[N-1], e b[1] e b[N-1], ecc. Il compilatore Intel può persino produrre codice "multiversione" contenente sia versioni vettorizzate che non vettorizzate del ciclo, consentendo di scegliere la variante corretta in fase di esecuzione in base ai valori di input! Tuttavia, se ci sono troppe manipolazioni nell'indicizzazione (ad esempio, a[i+j]), i compilatori non cercheranno di approfondire l'analisi e non vettorizzeranno il ciclo.

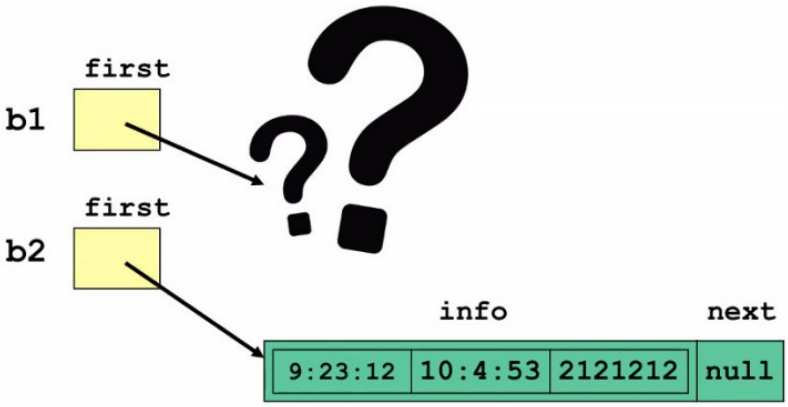
Sull'esempio del prof:



Situazione dopo `b2.Togli_Telefonata(t1);`



Se invece della telefonata `t1` avessimo rimosso da `b2` la telefonata `t2` la situazione sarebbe stata anche peggiore:



Chiaro quindi come l'aliasing abbia due cause:

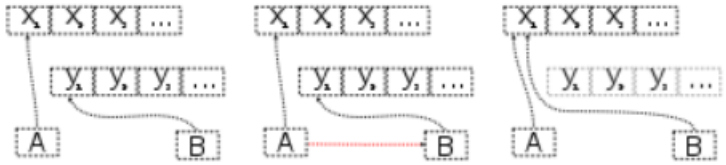
1) condivisione di memoria
2) funzioni con side effects

L'assegnazione standard esegue una *shallow copy*, cioè copiare tutti i valori del campo A che si vuole copiare e anche il suo tipo primitivo. La stessa cosa esegue il costruttore di copia standard.
Le shallow copy duplicano il meno possibile. Una copia superficiale di una collezione è una copia della struttura della collezione, non degli elementi. Con una copia superficiale, due collezioni condividono i singoli elementi.

Una soluzione a questo è la *deep copy*, in grado di duplicare non solo la struttura ma anche ogni singolo elemento (dati). Non si dipende dalla variabile, ma ci si mette di più ed è più costoso.

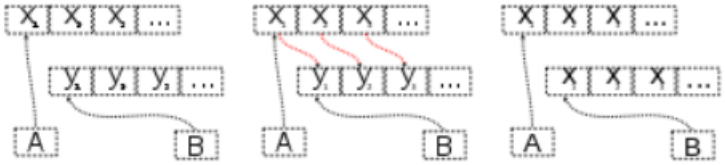
Segue un esempio su due variabili A e B.

Shallow:



Le variabili A e B si riferiscono ad aree di memoria diverse, ma quando B viene assegnata ad A le due variabili si riferiscono alla stessa area di memoria. Le modifiche successive al contenuto di una delle due variabili si riflettono immediatamente sul contenuto dell'altra, poiché ne condividono il contenuto.

Deep:



Le variabili A e B si riferiscono ad aree di memoria diverse, quando B viene assegnata ad A i valori dell'area di memoria a cui punta A vengono copiati nell'area di memoria a cui punta B. Le modifiche successive al contenuto di una delle due variabili rimangono uniche per A o B; il contenuto non viene condiviso.

In questo contesto

- si ridefinisce l'operatore di assegnazione (*assegnazione profonda*) per copiare struttura e contenuto, assegnando di copia.

Nella classe *bolletta*:

```
class bolletta {
public:
  bolletta& operator=(const bolletta& y);
  ...
};
```

Ora dobbiamo *copiare* e *distruggere* elemento per elemento (versioni *iterative e ricorsive*):

Aggiungiamo a `bolletta` due *metodi statici* di utilità che dichiariamo privati

```
class bolletta {
  ...
private:
  static nodo* copia(nodo*);
  static void distruggi(nodo*);
  ...
};
```

```
// implementazione iterativa
bolletta::nodo* bolletta::copia(nodo* p) {
  if (!p) return 0;
  nodo* primo = new nodo(p->info,0);
  // primo punta al primo nodo della copia della lista
  nodo* q = primo;
  // q punta all'ultimo nodo della lista finora copiata
  while (p->next) {
    p = p->next;
    q->next = new nodo(p->info,0);
    q = q->next;
  }
  return primo;
}
```

```
// implementazione iterativa
void bolletta::distruggi(nodo* p) {
  if(p!=nullptr) {
    // scorro tutta la lista deallocando ogni nodo
    nodo* q = p;
    while (p!=nullptr) {
      p = p->next;
      delete q;   // dealloco il nodo puntato da q
      q = p;
    }
  }
}
```

```
bolletta::nodo* bolletta::copia(nodo* p) {
  if (!p) return 0; // caso base: lista vuota
  else
  // passo induttivo:
  // per induzione copia(p->next) è la copia della coda
  // di p, e quindi inserisco una copia del primo nodo
  // di p in testa alla lista copia(p->next)
  return new nodo(p->info,copia(p->next));
}
```

```
void bolletta::distruggi(nodo* p) {
  // caso base: lista vuota, nulla da fare
  if (p) {
  // passo induttivo:
  // per induzione distruggi(p->next) dealloca
  // la coda di p, e quindi rimane da deallocare
  // solamente il primo nodo di p
    distruggi(p->next);
    delete p; // dealloco il nodo puntato da p
  }
}
```

Primo tentativo di definizione di `operator=`

```
bolletta& bolletta::operator=(const bolletta& b) {
  first = copia(b.first); // operator= tra puntatori
  return *this; // ritorna l'oggetto di invocazione
};
```

Problema:
1)Viene puntata la memoria di first prima della sua assegnazione

Soluzione: si pulisce lo *heap* e si copia (ciò permette di evitare la copia di oggetti già presenti oppure, come qui, non ancora presenti)

```
bolletta& bolletta::operator=(const bolletta& b) {
  distruggi(first); // pulizia dello heap
  first = copia(b.first);
  return *this;
}
```

Altro problema: *this* e l'oggetto di copia si riferiscono allo stesso contesto e si può avere un problema di interferenza dell'oggetto puntato.

(2) possibile assegnazione b = b; con:

```
bolletta& bolletta::operator=(const bolletta& b) {
  distruggi(first); // pulizia dello heap
  first = copia(b.first);
  return *this;
}
```

Soluzione:

```
bolletta& bolletta::operator=(const bolletta& b) {
  if (this != &b) {    // != tra puntatori
    distruggi(first); // pulizia dello heap
    first = copia(b.first);
  }
  return *this;
}
```

In *bolletta*, il costruttore di copia viene così ridefinito:

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

```
bolletta::bolletta(const bolletta& b) :
  first(copia(b.first)) {}
```

Scritto da Gabriel

Definiamo una funzione esterna

orario Somma_Durate(bolletta b)

che restituisca la somma delle durate delle telefonate in **b**.

```
orario Somma_Durate(bolletta b) {// NOTA: b passato per valore
  orario durata; // costruttore di default di orario
  while (!b.Vuota()) {
    // estrae dal primo nodo della lista
    telefonata t = b.Estrai_Una();
    durata = durata + t.Fine() - t.Inizio();
  } // vincolo: durata < 24 ore !
  return durata;
}
```

```
int main() {
  bolletta b1;

  telefonata t1(orario(9,23,12),orario(10,4,53),2121212);
  telefonata t2(orario(11,15,4),orario(11,22,1),3131313);

  b1.Aggiungi_Telefonata(t2);
  b1.Aggiungi_Telefonata(t1);

  cout << b1;

  cout << "LA SOMMA DELLE DURATE è "
       << Somma_Durate(b1) << endl;

  cout << b1;
}
```

Con il seguente output:

```
TELEFONATE IN BOLLETTA:
1) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212
2) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313

LA SOMMA DELLE DURATE è 0:38:38

TELEFONATE IN BOLLETTA:
1) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212
2) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313
```

Cosa succede e perché dà questo output?

(1) **b1** è passato per valore in **Somma_Durate(b1)**

(2) **b** oggetto locale a **Somma_Durate()** è una copia profonda di **b1**

(3) **Estrai_Una()** provoca side effects su **b**

La funzione esterna **`Chiamate_A`** rimuove dalla **`bolletta b`** passata per riferimento tutte le telefonate a **`num`** e restituisce una nuova bolletta contenente le telefonate tolte.

```
bolletta Chiamate_A(int num, bolletta& b) {
  bolletta selezionate, resto; // oggetti locali
  while (!b.Vuota()) {
    telefonata t = b.Estrai_Una();
    if (t.Numero() == num)
      selezionate.Aggiungi_Telefonata(t);
    else
      resto.Aggiungi_Telefonata(t);
  }
  b = resto; // overloading di operator= in bolletta
  return selezionate;
}
```

```
int main() {
  bolletta b1;
  telefonata t1(orario(9,23,12),orario(10,4,53),2121212);
  telefonata t2(orario(11,15,4),orario(11,22,1),3131313);
  telefonata t3(orario(12,17,5),orario(12,22,8),2121212);
  telefonata t4(orario(13,46,5),orario(14,0,33),3131313);
  b1.Aggiungi_Telefonata(t4);
  b1.Aggiungi_Telefonata(t3);
  b1.Aggiungi_Telefonata(t2);
  b1.Aggiungi_Telefonata(t1);
  cout << b1;
  bolletta b2 = Chiamate_A(2121212, b1);
  cout << b1;
  cout << b2;
}
```

OUTPUT

```
bolletta b2 = Chiamate_A(2121212, b1);
```

```
TELEFONATE IN BOLLETTA: (NB: b1)
1) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212
2) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313
3) INIZIO 12:17:5 FINE 12:22:8 NUMERO 2121212
4) INIZIO 13:46:5 FINE 14:0:33 NUMERO 3131313

TELEFONATE IN BOLLETTA: (NB: b1)
1) INIZIO 13:46:5 FINE 14:0:33 NUMERO 3131313
2) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313

TELEFONATE IN BOLLETTA: (NB: b2)
1) INIZIO 12:17:5 FINE 12:22:8 NUMERO 2121212
2) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212
```

```
bolletta Chiamate_A(int num, bolletta& b) {
  bolletta selezionate, resto;
  while (!b.Vuota()) {
    telefonata t = b.Estrai_Una();
    if (t.Numero == num)
      selezionate.Aggiungi_Telefonata(t);
    else
      resto.Aggiungi_Telefonata(t);
  }
  b = resto;
  return selezionate;
}
```

I due oggetti locali **`selezionate`** e **`resto`** esistono solo durante l'esecuzione della funzione **`Chiamate_A()`**.

Due problemi:

1) Memoria di *resto* e *selezionate* non è deallocata
2) *return selezionate:* oggetto ritornato costruito di copia, la memoria non viene deallocata

Si introduce il concetto di *lifetime* di un oggetto (vita). Ciò è molto utile per gli oggetti che devono essere eliminati (*garbage collection*).
Durante la creazione di un oggetto, è necessario stabilire la sua posizione in memoria, prima di inizializzarlo. Inizializzare significa inserire un valore nella posizione. La vita di un oggetto inizia subito dopo l'inizializzazione. Quando un oggetto muore, la sua posizione (memoria), che l'oggetto occupava, viene rilasciata e quindi il computer viene spento o la memoria viene occupata (utilizzata) da un altro oggetto. Rilasciare una memoria significa rendere non valido l'identificatore o il puntatore che la occupava. La vita di un oggetto termina quando la sua memoria viene rilasciata.
È necessario un certo tempo per creare un oggetto. È necessario un certo tempo per uccidere un oggetto. Quando si parla di un oggetto, sono coinvolte due cose: la posizione, che è la memoria, e il valore. Il significato di durata e di durata della memorizzazione sono simili, ma la durata è vista più dal punto di vista della posizione che dal punto di vista del valore. La durata della memorizzazione è il tempo che intercorre tra il momento in cui un luogo viene associato a un oggetto e il momento in cui il luogo viene dissociato dall'oggetto.

Le variabili che rimangono *per tutta la durata di un programma* sono le variabili statiche, *che rimangono sullo stack staticamente*.
Il *tempo di vita delle variabili* è quello riportato.
(utile perché serve a capire la logica dei costruttori e distruttori dopo).

Tempo di vita di una variabile

(1) Variabili di classe automatica (**Call stack**)

(2) Variabili di classe statica (**Memoria statica**)

- campi dati statici (P2)
- variabili globali (P1, almost deprecated)
- variabili statiche in corpo di funzione (bad practice)

(3) Variabili dinamiche (**Heap**)

Ora, introduciamo uno dei concetti fondamentali di P2: i distruttori.
Il distruttore è una funzione membro dell'istanza che viene invocata automaticamente ogni volta che un oggetto sta per essere distrutto. In altre parole, un distruttore è l'ultima funzione che viene richiamata prima che un oggetto venga distrutto.

- Anche il distruttore è una funzione membro speciale, come il costruttore. Il distruttore distrugge gli oggetti della classe creati dal costruttore.
- Il distruttore ha lo stesso nome del nome della classe preceduto dal simbolo tilde (~) (Alt+0256 in Windows, <Alt-Gr> + ì oppure *Pagina Giù* su Linux con layout italiano)
- Non è possibile definire più di un distruttore.
- Il distruttore è solo un modo per distruggere l'oggetto creato dal costruttore. Pertanto, il distruttore non può essere sovraccaricato.
- Il distruttore non richiede alcun argomento né restituisce alcun valore.
- Viene chiamato automaticamente quando l'oggetto esce dall'ambito.
- Il distruttore libera lo spazio di memoria occupato dagli oggetti creati dal costruttore.
- Nel distruttore, gli oggetti vengono distrutti al contrario della loro creazione.

Se l'oggetto viene creato usando new o il costruttore usa new per allocare memoria che risiede nell'heap o nel free store, il distruttore deve usare delete per liberare la memoria.

Come per i costruttori, esiste *sempre* il distruttore standard: l'invocazione procede *nell'ordine inverso rispetto alla dichiarazione dei campi*.
È accettabile il distruttore standard dentro bolletta? No, va ridefinito.

Distruttore della classe `bolletta`:

```
class bolletta {
public:
  ....
  ~bolletta();
  ....
};
```

```
//first punta alla testa della lista
bolletta::~bolletta() { distruggi(first); }
```

Scritto da Gabriel

Fondamentale: regole dei distruttori

Le *regole di invocazione dei distruttori* sono le seguenti:

- per gli **oggetti di classe statica**, al termine del programma (all'uscita dalla funzione principale `main`())
- per gli **oggetti di classe automatica** definiti in un blocco (ad esempio una funzione), all'uscita dal blocco in cui sono definiti. In particolare, ciò vale per i parametri formali di una funzione
- per gli **oggetti dinamici** (allocati sullo heap), quando viene invocato l'operatore di **`delete`** su un puntatore ad essi.
- per gli oggetti che sono **campi dati** di un oggetto **`x`**, quando **`x`** viene distrutto.
- gli **oggetti con lo stesso tempo di vita**, tipicamente oggetti definiti nello stesso blocco oppure oggetti statici di una classe, vengono distrutti nell'ordine inverso a quello in cui sono stati creati

In particolare, il distruttore viene invocato:

1. sui **parametri** di una funzione **passati per valore** all'uscita dalla funzione
2. sulle **variabili locali** di una funzione all'uscita dalla funzione
3. **sull'oggetto anonimo ritornato per valore** da una funzione non appena esso sia stato usato (se qualche ottimizzazione prolunga il tempo di vita dell'oggetto anonimo ritornato significa che non viene invocato il distruttore)

Ordine di distruzione per g++ e clang: [2], [3], [1]

Comportamento del distruttore

Supponiamo che la lista ordinata dei campi dati di una classe **C** sia x1,…,xn. Quando viene distrutto un oggetto di tipo **C**, viene invocato automaticamente il distruttore della classe **C**, standard oppure ridefinito, con il seguente comportamento:

1. innanzitutto viene eseguito il corpo del distruttore della classe **C**, se questo esiste
2. vengono quindi richiamati i distruttori per i campi dati nell'ordine inverso alla loro lista di dichiarazione, cioè xn,…,x1. Per un campo dati di tipo non classe (cioè primitivo o derivato) viene semplicemente rilasciata la memoria (è il "distruttore standard" dei tipi non classe) mentre per i tipi classe viene invocato il distruttore, standard oppure ridefinito.

Distruttore standard: semplicemente ha il corpo vuoto. Quindi il distruttore standard di una classe **C** si limita a richiamare i distruttori per i campi dati di **C** in ordine inverso di dichiarazione

Il distruttore di **`bolletta`** richiama il metodo statico privato

```
bolletta::distruggi(nodo* p) {
  if (p) {
    distruggi(p->next);
    delete p;
  }
}
```

che a sua volta esegue l'operatore **`delete`** su tutti gli elementi della lista.

[1] L'istruzione "`delete p;`" provoca l'invocazione del distruttore della classe `nodo`, e poichè esso non è stato ridefinito viene richiamato il distruttore standard di `nodo`.

[2] Il distruttore standard della classe `nodo` richiama a sua volta per ogni campo dati di `nodo`, cioè `info` e `next`, i rispettivi distruttori. Il campo puntatore `next` viene semplicemente deallocato, mentre per il campo `info` viene invocato il distruttore della classe `telefonata`, e siccome neanch'esso è stato ridefinito verrà richiamato il distruttore standard di `telefonata`.

[3] Il distruttore standard della classe `telefonata` dealloca l'intero `numero` e richiama il distruttore della classe `orario` per i campi `inizio` e `fine`, che sarà quindi il distruttore standard di `orario`.

[4] Il distruttore standard della classe `orario` dealloca il campo dati intero `sec`.

Fondamentale il controllo di avere un puntatore valido:

From the C++0x draft Standard.

$5.3.5/2 - "[...]In either alternative, the value of the operand of delete may be a null pointer value. [..."

Of course, no one would ever do 'delete' of a pointer with NULL value, but it is safe to do. Ideally one should not have code that does deletion of a NULL pointer. But it is sometimes useful when deletion of pointers (e.g. in a container) happens in a loop. Since delete of a NULL pointer value is safe, one can really write the deletion logic without explicit checks for NULL operand to delete.

Modo più elegante: ridefinire il distruttore della classe `nodo` in modo che esso provveda a distruggere l'oggetto puntato da `next`

```
bolletta::nodo::~nodo()
// invocazione automatica distruttore di telefonata
// deallocazione automatica puntatore next
{
  if (next != 0)
    delete next; // chiamata ricorsiva
}
```

In questo modo il distruttore di `bolletta` diventa:

```
bolletta::~bolletta() { if(first) delete first; }
```

Queste sono le premesse giuste per dei cosa stampa (con il flag *-fno-elide-constructors*).
Senza l'opzione produce un output diverso.

```
class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "~C ";}
};

// funzione esterna, pass. per valore
C F(C p) { return p; }

C w("3"); // variabile globale

class D {
public:
  static C c; // campo dati statico
};
C D::c("4");

int main() {
  C x("5"), y("6"); D d;
  y=F(x); cout << "UNO\n";
  C z=F(x); cout << "DUE\n";
}
```

```
5~C 5~C UNO
5~C 5~C DUE
5~C 5~C 5~C 4~C 3~C
```

```
class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "~C ";}
};

C  F(C& p) { return p; } // passaggio per riferimento

C w("3");

class D {
public:
  static C c;
};
C D::c("4");

int main() {
  C x("5"), y("6"); D d;
  y=F(x); cout << "UNO\n";
  C z=F(x); cout << "DUE\n";
}
```

```
5~C UNO
5~C DUE
5~C 5~C 5~C 4~C 3~C
```

```
class C {
public:
  string s;
  C(string x="1"): s(x) {}
  ~C() {cout << s << "~C ";}
};

C& F(C& p) { return p; } //passaggio e ritorno per riferimento

C w("3");

class D {
public:
  static C c;
};
C D::c("4");

int main() {
  C x("5"), y("6"); D d;
  y=F(x); cout << "UNO\n";
  C z=F(x); cout << "DUE\n";
}
```

```
UNO
DUE
5~C 5~C 5~C 4~C 3~C
```

Attenzione alla *rule of three* della modellazione di una classe:

- distruttore
- costruttore di copia
- operatore di assegnazione (con parametro const T&)

Breve parentesi sugli *array*:
Un array è una struttura di dati che contiene un gruppo di elementi. In genere questi elementi sono tutti dello stesso tipo di dati, come ad esempio un numero intero o una stringa. Gli array sono comunemente utilizzati nei programmi informatici per organizzare i dati in modo che un insieme di valori correlati possa essere facilmente ordinato o ricercato.

Esistono due tipi principali di array: gli array statici e gli array dinamici.

Gli array statici vengono creati sullo stack e hanno una durata automatica: non è necessario gestire manualmente la memoria, ma vengono distrutti al termine della funzione in cui si trovano. Hanno necessariamente una dimensione fissa al momento della compilazione:
int foo[10];

Gli array creati con l'operatore new[] hanno una durata di memoria dinamica e sono memorizzati sull'heap (tecnicamente il "free store"). In fase di esecuzione possono avere qualsiasi dimensione, ma è necessario allocarli e liberarli da soli, poiché non fanno parte dello stack frame:

int foo = new int[10];*
delete[] pippo;

Anch'essi, chiaramente, devono avere operazioni di costruzione e distruzione.

```
int arrayStatico[5] = {3,2,-3};
int* arrayDinamico = new int[5];
arrayDinamico[0]=3;
*(arrayDinamico+1) = 2;
delete[] arrayDinamico;
```

Un esempio in codice:

```
class C {
public:
  int i;
  C(int x=3): i(x) {}
  ~C() {cout << i << "~C ";}
};

int main() {
  C a[4] = {C(1), C(), C(8)};
  // distruzione oggetti anonimi, stampa: 8~C 3~C 1~C
  cout << a[0].i << a[1].i << a[2].i << a[3].i << endl;
  // stampa: 1383
}
// all'uscita stampa: 3~C 8~C 3~C 1~C
```

Dichiarazioni di classi, friend, nested classes, iteratori ed esempi

Da un punto di vista di programmazione ad oggetti, si intende nascondere la parte privata della dichiarazione di C all'utente finale.

```
class C {
public:
  // parte pubblica
private:
  // parte privata
};
```

Quanto segue sarebbe una dichiarazione incompleta della classe, con un *handle*/*puntatore* ad essa pronta per l'uso:

```
// file C_handle.h
class C_handle {
public:
  // parte pubblica
private:
  class C_privata; // dichiarazione incompleta
  C_privata* punt; // solo punt. e ref. per dich.incompleta
};
```

oggetti allocati sullo heap

Nell'implementazione separata di `C_handle` sarà definita la classe `C_privata` contenente la parte privata di `C`

```
// file C_handle.cpp
class C_handle::C_privata {
  // parte privata
};
```

Una classe C può dichiarare ed usare puntatori e riferimenti ad una classe D che viene meramente dichiarata (messa lì, ma poi dovrà essere definita). Tale pratica è la *dichiarazione incompleta* della classe *D*.

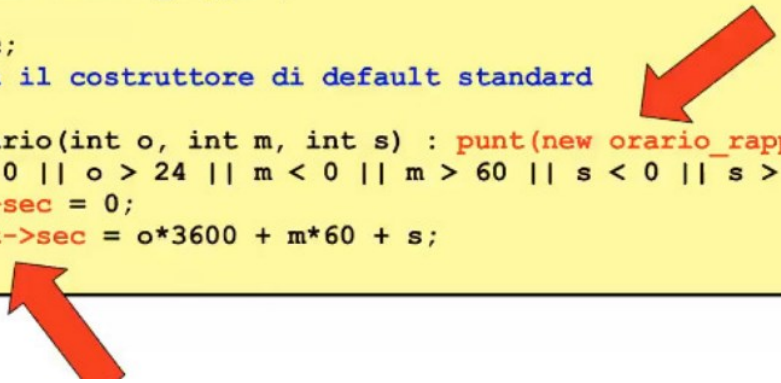
Attenzione: In una classe **C** una dichiarazione che la classe **D** è amica di **C** funge anche da dichiarazione incompleta di **D**.

```
class D; // dichiar.incompleta

class C {
  D* p;                 // campo dati
  D* m();          // tipo di ritorno
  void n(...,D*,...); // parametro
  D& f(...);       // tipo di ritorno
...
};

class D {
... // definizione della classe
};
```

L'esempio d'uso è quindi avere un puntatore, per il momento, alla classe che ci interessa privata:

```
// File orario.h
#ifndef ORARIO_H
#define ORARIO_H
#include <iostream>
using std::ostream;

class orario {
  public:
    orario(int =0,int =0,int =0);
    int Ore() const;
    int Minuti() const;
    int Secondi() const;
    void AvanzaUnOra();
  private:
    class orario_rappr;
    orario_rappr* punt; // oggetti allocati sullo heap
};
ostream& operator<<(ostream&,const orario&);
#endif
```

```
// File orario.cpp
#include "orario.h"

class orario::orario_rappr {
  public:
    int sec;
}; // basta il costruttore di default standard

orario::orario(int o, int m, int s) : punt(new orario_rappr)
{  if (o < 0 || o > 24 || m < 0 || m > 60 || s < 0 || s > 60)
     punt->sec = 0;
  else punt->sec = o*3600 + m*60 + s;
}
```



```
int orario::Ore() const
{ return punt->sec / 3600; }

int orario::Minuti() const
{ return (punt->sec - (punt->sec / 3600)*3600) / 60; }

int orario::Secondi() const
{ return punt->sec % 60; }

void orario::AvanzaUnOra()
{ punt->sec = (punt->sec + 3600) % 86400; }

ostream& operator<<(ostream& os,const orario& t)
{ return os << t.Ore() << ':' << t.Minuti() << ':'
    << t.Secondi(); }
```

Quanto presentato non è proprio la scelta migliore; ci potrebbero essere problemi di interferenza (codice riportato). Sarà quindi necessario ridefinire adeguatamente assegnazione, costruttore di copia e distruttore di *orario*.

```
orario t1(17,11,27), t2;
cout << t1 << ' ' << t2 << endl;
// stampa: 17:11:27 0:0:0
t2=t1; // stesso puntatore punt in t2 e t1
t1.AvanzaUnOra(); // interferenza
cout << t1 << ' ' << t2 << endl;
// stampa: 18:11:27 18:11:27
```

Attenzione al comportamento dell'operatore di stampa: deve usare un campo privato e addirittura per valore:

```
ostream& operator<<(ostream& os, bolletta b) {
  // NOTA BENE: b passato per valore
  os << "TELEFONATE IN BOLLETTA" << endl;
  int i = 1;
  while (!b.Vuota()) {
    os << i << ") " << b.Estrai_Una() << endl;
    i++;
  }
  return os;
}
```

Perché si avrebbe un problema?

Il primo parametro è uno stream. Poiché non si ha accesso all'oggetto *stream* (non è possibile modificarlo), questi non possono essere operatori membri, ma devono essere esterni alla classe. Pertanto, devono accessi come *friend* della classe o avere accesso a un metodo pubblico che esegua lo streaming per voi.
È inoltre consuetudine che questi oggetti restituiscano un riferimento a un oggetto stream, in modo da poter concatenare le operazioni di stream.
Non è possibile farlo come funzione membro, perché il parametro implicito this è il lato sinistro dell'operatore <<.

Una *funzione friend* in C++ è una funzione dichiarata al di fuori di una classe, ma in grado di accedere ai membri privati e protetti della classe. Nella programmazione possono verificarsi situazioni in cui si desidera che due classi condividano i loro membri. Questi membri possono essere membri di dati, funzioni di classe o modelli di funzione.

```
// file bolletta.h
class bolletta {
  ...
  // funzione esterna dichiarata friend
  friend ostream& operator<<(ostream&,const bolletta&);
  ...
};
```

```
// nel file bolletta.cpp
ostream& operator<<(ostream& os, const bolletta& b) {
  os << "TELEFONATE IN BOLLETTA" << endl;
  bolletta::nodo* p = b.first; // per "amicizia"!
  int i = 1;
  while (p) {
    os << i++ << ") " << p->info << endl;
    p = p->next;
  }
  return os;
}
```

Accesso alle classi annidate/Nested classes access

Lo standard C++ del 2003 cita che:
“I membri di una classe annidata non hanno un accesso speciale ai membri di una classe racchiusa, né alle classi o alle funzioni che hanno concesso l'amicizia a una classe racchiusa; devono essere rispettate le consuete regole di accesso. I membri di una classe racchiusa non hanno accesso speciale ai membri di una classe annidata; devono essere rispettate le consuete regole di accesso”

In C++ 98 si ha che:
“I membri di una classe annidata non hanno un accesso speciale ai membri di una classe racchiusa, né alle classi o alle funzioni che hanno concesso l'amicizia a una classe racchiusa; devono essere rispettate le consuete regole di accesso.”

Da C++ 11:
“Una classe annidata è un membro e come tale ha gli stessi diritti di accesso di ogni altro membro della classe. I membri di una classe annidata non hanno un accesso speciale ai membri di una classe racchiusa, né alle classi o alle funzioni che hanno concesso l'amicizia a una classe racchiusa; devono essere rispettate le consuete regole di accesso”

Le regole di accesso specificano anche che:
“Un membro di una classe può anche accedere a tutti i nomi per cui la classe ha accesso”

Scritto da Gabriel

In conclusione: i membri come *friend* accedono agli oggetti privati quando dichiarati come tali ed altri oggetti della classe.
Normalmente, la relazione di amicizia non è simmetrica e non è transitiva (quindi *deve essere dichiarata* dalla classe contenuta (iteratore) rispetto alla classe container (contenitore)).

Parliamo ora degli iteratori, che servono a scorrere ed accedere agli elementi di una collezione.
Gli iteratori vengono utilizzati per puntare agli indirizzi di memoria dei contenitori STL. Sono utilizzati principalmente nelle sequenze di numeri, caratteri ecc. Riducono la complessità e il tempo di esecuzione del programma.
Vari tipi di iteratori sono spesso forniti tramite l'interfaccia di un contenitore. Sebbene l'interfaccia e la semantica di un dato iteratore siano fisse, gli iteratori sono spesso implementati in termini di strutture sottostanti all'implementazione di un contenitore e sono spesso strettamente accoppiati al contenitore per consentire la semantica operativa dell'iteratore. Un iteratore esegue la traversata e dà accesso agli elementi dei dati in un contenitore, ma non esegue l'iterazione.

Applicandolo al nostro esempio, abbiamo bisogno di variabili che puntano ai vari oggetti dei vari contenitori definiti:

```
class contenitore {
private:
  class nodo {
  public: // per convenienza nell'esempio
    int info;
    nodo* next;
    nodo(int x, nodo* p): info(x), next(p) {}
  };
  nodo* first; // puntatore al primo nodo della lista

public:
  contenitore(): first(0) {}
  void aggiungi_in_testa(int x) {first = new nodo(x,first);}
}
```

Segue la classe *iteratore* che ha come oggetti gli indici dei nodi della classe *contenitore*.

```
class iteratore {
private:
  contenitore::nodo* punt; // nodo puntato dall'iteratore
public:
  bool operator==(const iteratore& i) const {
       return punt == i.punt;
  }
  bool operator!=(const iteratore& i) const {
       return punt != i.punt;
  }
  iteratore& operator++() { // operator++ prefisso
    if (punt) punt = punt->next; return *this;
  }
// se it punta all'ultimo nodo, da ++it non si torna indietro
// nessun costruttore per il momento
};
```

Problema: *nodo* ed i suoi membri non sono accessibili dall'esterno.

```
class contenitore {
// friend class iteratore; // non necessaria da C++03
private:
  class nodo {
    ...
  };
  nodo* first;
public:
  class iteratore { // classe annidata nella parte pubblica
  private:
    contenitore::nodo* punt;
    ...
  };
  contenitore();
  void aggiungi_nodo(int);
}
```

```
class contenitore {

private:
  class nodo { ... };
  nodo* first;
public:
  class iteratore {                  // necessaria
    friend class contenitore; // dichiarazione di amicizia
    ...
  }; // Attenzione, va definita prima dei metodi che la usano
  ...
  iteratore begin() const;    // "costruttore di iteratore"
  iteratore end() const;      // "costruttore di iteratore"
  // operatore di subscripting
  int& operator[](const iteratore&) const;
};
```

```
contenitore::iteratore contenitore::begin() const {
  iteratore aux;    // costruttore di default standard
  aux.punt = first; // per amicizia ho accesso a punt
  return aux;
}

contenitore::iteratore contenitore::end() const {
  iteratore aux;
  aux.punt = 0; // per amicizia
  return aux;
}

int& contenitore::operator[](const contenitore::iteratore& it) const {
  return it.punt->info; // per amicizia, nessun controllo su it.punt
}
```

Possiamo ora utilizzare gli iteratori della classe contenitore come nel seguente esempio di funzione esterna:

```
int somma_elementi(const contenitore& c) {
  int s=0;
  for(contenitore::iteratore it=c.begin(); it!=c.end(); ++it)
    s += c[it];
  return s;
}
```

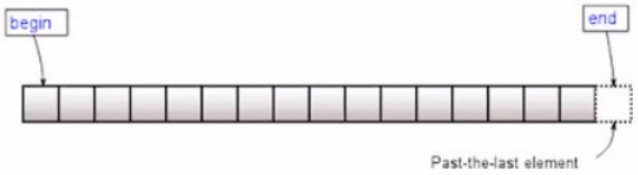
Esempi utili della creazione della classe *iterator*: i metodi di accesso all'inizio (begin()) e fine (end()) rispetto agli elementi.

Dichiarazione della classe `iteratore`:

```
class iteratore {
  friend class bolletta;
public:
  bool operator==(const iteratore&) const;
  bool operator!=(const iteratore&) const;
  iteratore& operator++();        // ++ prefisso
  iteratore operator++(int);      // ++ postfisso
private:
  bolletta::nodo* punt;
};
// notare che non ci sono costruttori espliciti
```

Metodi `begin()` ed `end()` e l'overloading dell'operatore di indicizzazione `[]`.

```
class bolletta {
public:
  ...
  iteratore begin() const;
  iteratore end() const;
  telefonata& operator[](const iteratore&) const;
  // tipo di ritorno per riferimento
  ...
private:
  ...
};
```



```
// bolletta.h
#ifndef BOLLETTA_H
#define BOLLETTA_H
#include "telefonata.h"

class bolletta {
private:
  class nodo {
  public:
    nodo();
    nodo(int x, nodo* p): info(x), next(p) {}
    telefonata info;
    nodo* next;
    ~nodo(); // distruttore "ricorsivo"
  };
  nodo* first; // puntatore al primo nodo della lista
  static nodo* copia(nodo*);
  static void distruggi(nodo*);

// continua ...
```

```
// ... continuazione
public:
  class iteratore {
  friend class bolletta;
  private:
    bolletta::nodo* punt; // nodo puntato dall'iteratore
  public:
    bool operator==(const iteratore&) const;
    bool operator!=(const iteratore&) const;
    iteratore& operator++(); // operator++ prefisso
    iteratore operator++(int); // operator++ postfisso
  }; // end classe iteratore
  bolletta();
  ~bolletta(); // distruzione profonda
  bolletta(const bolletta&); // copia profonda
  bolletta& operator=(const bolletta&); // assegnazione profonda
  bool Vuota() const;
  void Aggiungi_Telefonata(const telefonata& t);
  void Togli_Telefonata(const telefonata& t)
  telefonata Estrai_Una();
  // metodi che usano iteratore
  iteratore begin() const;
  iteratore end() const;
  telefonata& operator[](const iteratore&) const;
};
#endif
```

```
// bolletta.cpp
#include "bolletta.h"

bool bolletta::iteratore::operator==(const iteratore&) const {
  return punt == i.punt;
}

bool bolletta::iteratore::operator!=(const iteratore&) const {
  return punt != i.punt;
}

bolletta::iteratore& bolletta::iteratore::operator++() {// prefisso
  if (punt) punt = punt->next;    //side-effect
  return *this;
} // NB: se punt==0 non fa nulla

bolletta::iteratore bolletta::iteratore::operator++(int) {// postfisso
  iteratore aux = *this;
  if (punt) punt = punt->next;     //side-effect
  return aux;
}
```

```
// continuazione file bolletta.cpp

bolletta::iteratore bolletta::begin() const {
  bolletta::iteratore aux;
  aux.punt = first;  // amicizia
  return aux;
}

bolletta::iteratore bolletta::end() const {
  bolletta::iteratore aux;
  aux.punt = nullptr;  // amicizia
  return aux;
}

telefonata& bolletta::operator[](const bolletta::iteratore& it) const {
  return (it.punt)->info; // amicizia
  // NB: nessun controllo it.punt != 0
}
```

L'utente esterno può calcolare la somma delle durate delle telefonate di una bolletta con la seguente funzione:

```
orario Somma_Durate(const bolletta& b) {// per riferimento
  orario durata;
  for(bolletta::iteratore it = b.begin(); it != b.end(); ++it)
    durata = durata + (b[it].Fine() - b[it].Inizio());
  return durata;
}
```

Esercizio:

Ridefinire l'operatore "**telefonata& operator*()**" come metodo della classe **iteratore** cosicchè la funzione **Somma_Durate** possa essere scritta nel seguente modo:

```
orario Somma_Durate(const bolletta& b) { // per riferimento
  orario durata;
  for (bolletta::iteratore it = b.begin(); it != b.end(); ++it)
    durata = durata + ((*it).Fine() - (*it).Inizio());
  return durata;
}
```

```
telefonata& operator*(telefonata& t, int n)   //OPERATORE *
{
   orario aux;
   aux = t.Inizio() + orario(0, 0, n);
   t.fine = aux;
   return t;
}
```

Prima di descrivere l'operatore di selezione (->), descriviamo l'operatore prefisso e l'operatore postfisso *con relativo overloading*:

L'operatore di postfisso decremento significa che l'espressione viene valutata prima utilizzando il valore originale della variabile e poi la variabile viene decrementata (diminuita). L'operatore di incremento del prefisso significa che la variabile viene prima incrementata e poi l'espressione viene valutata utilizzando il nuovo valore della variabile.
In Pre-incremento/Decremento e Post-incremento/Decremento, la differenza si basa solo su un parametro fittizio nella funzione sovraccaricata.

operator++() => Incremento prefisso
operator--() => Decremento prefisso

operator++(int) => Incremento postfisso
operatore--(int) => Decremento postfisso

Un esempio concreto (classe Point):

```
// increment_and_decrement1.cpp
class Point
{
public:
  // Declare prefix and postfix increment operators.
  Point& operator++();      // Prefix increment operator.
  Point operator++(int);    // Postfix increment operator.

  // Declare prefix and postfix decrement operators.
  Point& operator--();      // Prefix decrement operator.
```

Scritto da Gabriel

```
  Point operator--(int);    // Postfix decrement operator.

  // Define default constructor.
  Point() { _x = _y = 0; }

  // Define accessor functions.
  int x() { return _x; }
  int y() { return _y; }
private:
  int _x, _y;
};

// Define prefix increment operator.
Point& Point::operator++()
{
  _x++;
  _y++;
  return *this;
}

// Define postfix increment operator.
Point Point::operator++(int)
{
  Point temp = *this;
  ++*this;
  return temp;
}

// Define prefix decrement operator.
Point& Point::operator--()
{
  _x--;
  _y--;
  return *this;
}

// Define postfix decrement operator.
Point Point::operator--(int)
{
  Point temp = *this;
  --*this;
  return temp;
}

int main()
{
}
```

Tornando a noi, *l'operatore di selezione* ritorna un puntatore ad una classe (oppure un oggetto di una classe per cui è stato ridefinito ->.

Definire l'overloading di **->** come metodo di iteratore in modo tale che Somma_Durate() possa essere scritta nel seguente modo:

```
orario Somma_Durate(const bolletta& b) { // per riferimento
  orario durata;
  for (bolletta::iteratore it = b.begin(); it != b.end(); ++it)
    durata = durata + (it->Fine() - it->Inizio());
  return durata;
}
```

telefonata& bolletta::iteratore::operator->(bolletta::iteratore it) const
{
return (it.p)->info;
}

telefonata bolletta::iteratore::operator*(bolletta::iteratore it) const*
{
return &(it.p)->info;
}

```
public:
  class iteratore {
  friend class bolletta;
  private:
    bolletta::nodo* punt; // nodo puntato dall'iteratore
  public:
    bool operator==(const iteratore&) const;
    bool operator!=(const iteratore&) const;
    iteratore& operator++(); // operator++ prefisso
    iteratore operator++(int); // operator++ postfisso
    telefonata* operator->() const {return &(punt->info);}
    telefonata& operator*() const {return punt->info;}
  };
  bolletta();
  ~bolletta(); // distruzione profonda
  bolletta(const bolletta&); // copia profonda
  bolletta& operator=(const bolletta&); // assegnazione profonda
  bool Vuota() const;
  void Aggiungi_Telefonata(const telefonata& t);
  void Togli_Telefonata(const telefonata& t)
  telefonata Estrai_Una();
  // metodi che usano iteratore
  iteratore begin() const;
  iteratore end() const;
  telefonata& operator[](const iteratore&) const;
};
```

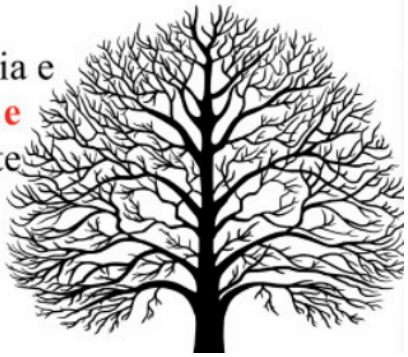
```
telefonata* operator->() const {return &(punt->info);}
telefonata& operator*() const {return punt->info;}
```

Question: perché questi metodi const sono compatibili con tipi di ritorno non const?

Answer:
Questi metodo sono di “lettura/scrittura”; conseguentemente, non per forza il tipo sotto deve essere const, devono solo leggerlo e avremo un riferimento const T&, essendo passato per riferimento.

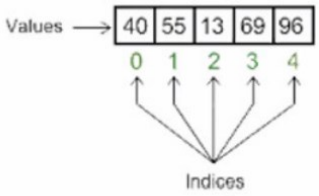
```
class Nodo {
private:
  Nodo(char c='*', Nodo* s=0, Nodo* d=0): info(c), sx(s), dx(d) {}
  char info;
  Nodo* sx;
  Nodo* dx;
};
class Tree {
public:
  Tree(): root(0) {}
  Tree(const Tree&); // dichiarazione costruttore di copia
private:
  Nodo* root;
};
```

Gli oggetti della classe Tree rappresentano alberi binari ricorsivamente definiti di char. Si ridefiniscano assegnazione, costruttore di copia e distruttore di Tree come **assegnazione, copia e distruzione profonda**. Scrivere esplicitamente eventuali dichiarazioni friend che dovessero essere richieste da tale definizione.



```
/*
Gli oggetti della classe Tree rappresentano alberi binari ricorsivamente definiti di char. Si
ridefiniscano assegnazione, costruttore di copia e distruttore di Tree come assegnazione, copia e
distruzione profonda. Scrivere esplicitamente eventuali dichiarazioni friend che dovessero essere
richieste da tale definizione.
*/

class Nodo {
  friend class Tree;
private:
  Nodo(char c='*', Nodo* s=0, Nodo* d=0): info(c), sx(s), dx(d) {}
  char info;
  Nodo* sx;
  Nodo* dx;
};

class Tree {
private:
  Nodo* root;
  static Nodo* copia(Nodo* r) {
    // caso base: albero vuoto
    if(r==nullptr) return nullptr;
    // passo induttivo: dalle copie profonde ritornate induttivamente da
    // copia(r->sx) e copia(r->dx), costruisco la copia dell'albero radicato in *r
    // nel seguente modo:
    return new Nodo(r->info,copia(r->sx),copia(r->dx));
  }
  static void distruggi(Nodo* r) {
    // caso base: albero vuoto
    if(r!=nullptr) {
      // passo induttivo: albero non vuoto
      distruggi(r->dx);
      distruggi(r->sx);
      delete r;
    }
  }

public:
  Tree(): root(nullptr) {}
  Tree(const Tree& t): root(copia(t.root)) {} // copia profonda
  Tree& operator=(const Tree& t) {
    if(this != &t) {
      distruggi(root);
      root = copia(t.root);
    }
    return *this;
  }

  ~Tree() {if(root) distruggi(root);}
};

int main() {
  Tree t1,t2;
  t1=t2;
  Tree t3=t2;
}
```

Definire una classe **Vettore** i cui oggetti rappresentano array di interi. Vettore deve includere un costruttore di default, una operazione di concatenazione che restituisce un nuovo vettore v1+v2, una operazione di append v1.append(v2), l'overloading dell'uguaglianza, dell'operatore di output e dell'operatore di indicizzazione. Deve inoltre includere il costruttore di copia profonda, l'assegnazione profonda e la distruzione profonda.




```
/* ESERCIZIO:
Definire una classe vettore i cui oggetti rappresentano array di interi.
vettore deve includere un costruttore di default, una operazione di
concatenazione che restituisce un nuovo vettore v1+v2, una operazione di
append v1.append(v2), l'overloading dell'uguaglianza, dell'operatore di
output e dell'operatore di indicizzazione.  Deve inoltre includere il
costruttore di copia profonda, l'assegnazione profonda e la distruzione profonda.
*/

#include<iostream>

class Vettore {
private:
  int* a;
  unsigned int size; // size ≧ 0  (garantito da unsigned int)
  // vettore vuoto IFF a==nullptr && size == 0
  // vettore non vuoto IFF a!=nullptr && size>0
public:
  // unsigned int => Vettore
  Vettore(unsigned int s =0, int init=0): a(s==0 ?  nullptr : new int[s]), size(s) {
    for(int j=0; j<size; ++j) a[j]=init;
  }

  Vettore(const Vettore& v): a(v.size == 0 ? nullptr : new int[v.size]), size(v.size) {
    for(unsigned int j=0; j<size; ++j) a[j]=v.a[j];
  }

  Vettore& operator=(const Vettore& v) {
    if (this != &v) {
      delete[] a; // attenzione: delete[] e NON delete
      size = v.size;
      a = size == 0 ? nullptr : new int[size];
      for (int i = 0; i < size; i++) a[i] = v.a[i];
    }
    return *this;
  }

  ~Vettore() {if(a) delete[] a;}

  Vettore& append(const Vettore& v) {
    if (v.size != 0){
      int* aux = new int[size+v.size];
      for (int i = 0; i < size; i++) aux[i] = a[i];
      for (int i = 0; i < v.size; i++) aux[size+i] = v.a[i];
      size += v.size;
      delete[] a; // FONDAMENTALE
      a = aux;
    }
    return *this;
  }
```

```
Vettore operator+(const Vettore& v) const {
  Vettore aux(*this);
  aux.append(v);
  return aux;
}

bool operator==(const Vettore& v) const {
  if(this == &v) return true;
  if(size!=v.size) return false;
  // size == v.size >= 0
  for(int i=0;i<size;i++)
    if(a[i]!=v.a[i]) return false;
  // forall i in [0,size-1], a[i]==v.a[i]
  return true;
}

int& operator[](unsigned int i) const {
  return *(a+i);
}

unsigned int getSize() const {
  return size;
} //FINE CLASSE
```

```
};


std::ostream& operator<<(std::ostream& os, const Vettore& v) {
  os << '[';
  int i = 0;
  while(i<(v.getSize())) {
    os << v[i] << ((i!=v.getSize()-1)  ? ',' : ']' );
    i++;
  }
  if(v.getSize()==0) os << ']';
  return os;
}

int main() {
  Vettore v1(4), v2(3,2), v3(5,-3);
  v1 = v2+v3;
  v2.append(v2);
  v3.append(v1).append(v3);
  std::cout << v1 << std::endl;
  std::cout << v2 << std::endl;
  std::cout << v3 << std::endl;
}
```

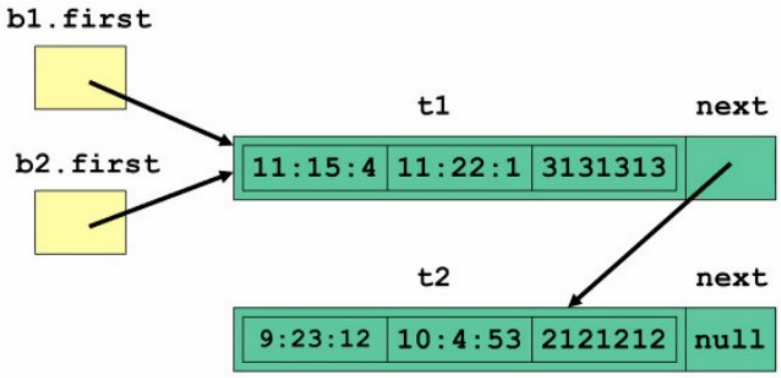
Visti questi due esercizi, prendiamo un cosa-stampa:

```
class S {
public:
  string s;
  S(string t): s(t) {}
};
class N {
private:
  S x;
public:
  N* next;
  N(S t, N* p): x(t), next(p) {cout << "N2 ";}
  ~N() {if (next) delete next; cout << x.s + "~N ";}
};
class C {
  N* pointer;
public:
  C(): pointer(0) {}
  ~C() {delete pointer; cout << "~C ";}
  void F(string t1, string t2 = "pippo") {
    pointer = new N(S(t1),pointer);
    pointer = new N(t2,pointer);
  }
};
main(){
  C* p = new C; cout << "UNO\n";
  p->F("pluto","paperino"); p->F("topolino"); cout <<"DUE\n";
  delete p; cout <<"TRE\n";
}
```


```
NESSUNA STAMPA UNO
N2 N2 N2 N2 DUE
pluto~N paperino~N topolino~N pippo~N ~C TRE
NESSUNA STAMPA
```

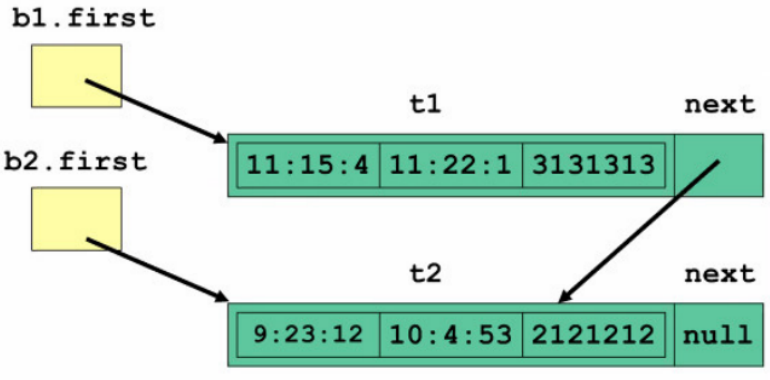
Ci facciamo una domanda → Le copie profonde sono davvero una soluzione ottimale?
Sono una soluzione costosa, talvolta inutile.

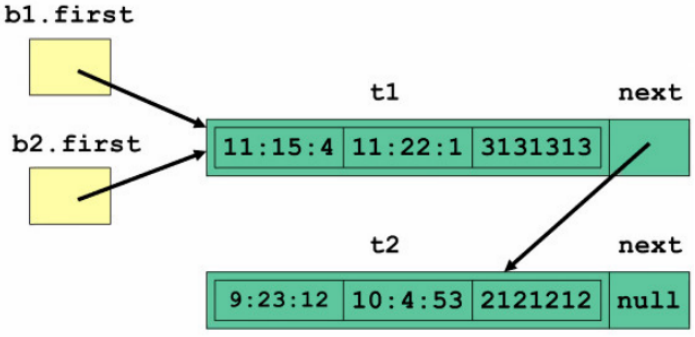
Situazione di condivisione di memoria



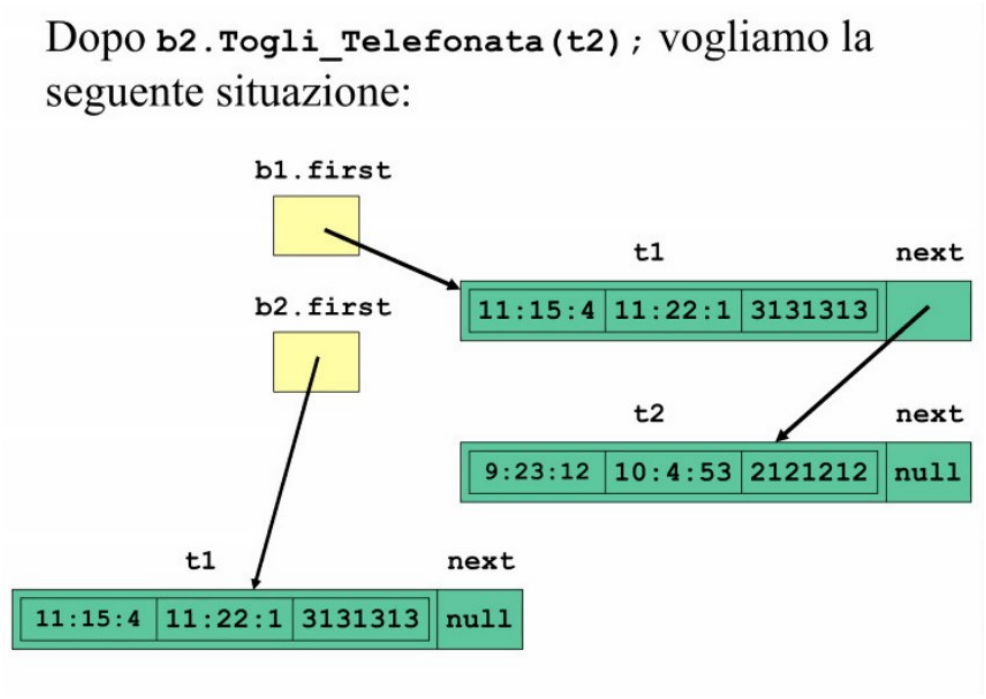
Cosa si può fare per mantenere uno stato di condivisione della memoria consistente dopo `b2.Togli_Telefonata(t1);` ?

Vorremmo la seguente situazione dopo `b2.Togli_Telefonata(t1);`





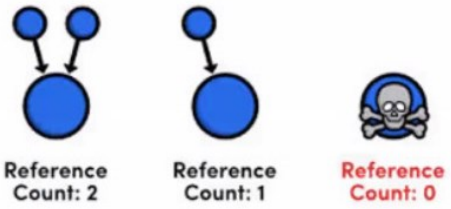
Invece cosa si può fare dopo `b2.Togli_Telefonata(t2);` ?



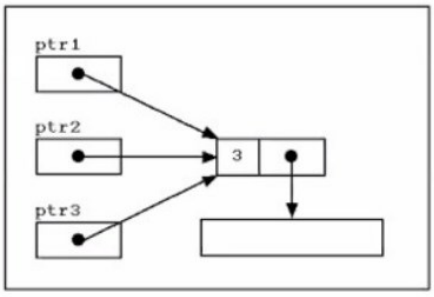
Problemi vari delle deep copy:

- Se la struttura dati da copiare può avere dei cicli, un'implementazione di copia profonda deve tenere traccia degli oggetti già copiati per evitare una copia infinita.
- Se non creiamo la copia profonda in modo corretto, la copia punterà all'originale, con conseguenze disastrose (puntatori nulli, oggetti deallocati, pesante computazionalmente se il puntatore nullo è dopo molti elementi, ad esempio).

È quindi utile tenere traccia della vita degli oggetti. Un modo di farlo è la *reference counting*.



Spesso è difficile tracciare manualmente la vita di un oggetto. Questo è particolarmente vero quando l'oggetto viene utilizzato ampiamente nel programma. Sarebbe utile avere una gestione della durata di vita in qualche modo automatica. Una volta terminato l'utilizzo dell'oggetto, questo dovrebbe essere eliminato. Il reference counting è una di queste tecniche.



Questo metodo consiste semplicemente nel mantenere un contatore aggiuntivo insieme a ogni oggetto creato. Il contatore è il numero di riferimenti che esistono all'oggetto, nel caso del C/C++ quanti puntatori fanno riferimento a questo oggetto. Ogni volta che un puntatore viene copiato, il contatore viene incrementato e ogni volta che un puntatore esce dallo scope o viene resettato, il contatore viene decrementato. Quando il conteggio raggiunge lo zero, l'oggetto viene cancellato, poiché non viene più utilizzato.

Quindi:
Si incapsula in una classe il puntatore nodo* e si ridefinisce assegnazione, costruttore di copia e distruttore. Si definisce un cosiddetto *smart pointer*, i quali dovranno essere dotati di una interfaccia pubblica che permette all'utente di utilizzarli come fossero puntatori ordinari.

Scritto da Gabriel

Uno Smart Pointer è una classe wrapper su un puntatore con un operatore come * e -> overloaded entrambi. Gli oggetti della classe Smart Pointer hanno l'aspetto di normali puntatori. Ma, a differenza dei normali puntatori, possono deallocare e liberare la memoria dell'oggetto distrutto. (essi sono implementati da C++ 11).

L'idea è di prendere una classe con un puntatore, un distruttore e operatori sovraccaricati come * e ->. Poiché il distruttore viene chiamato automaticamente quando un oggetto esce dall'ambito, la memoria allocata dinamicamente viene automaticamente cancellata (o il conteggio dei riferimenti può essere decrementato).

Ne esistono di vari tipi (nominati solo a scopo didattico, presenti ad esempio in un insieme di librerie utili C++, come Boost):

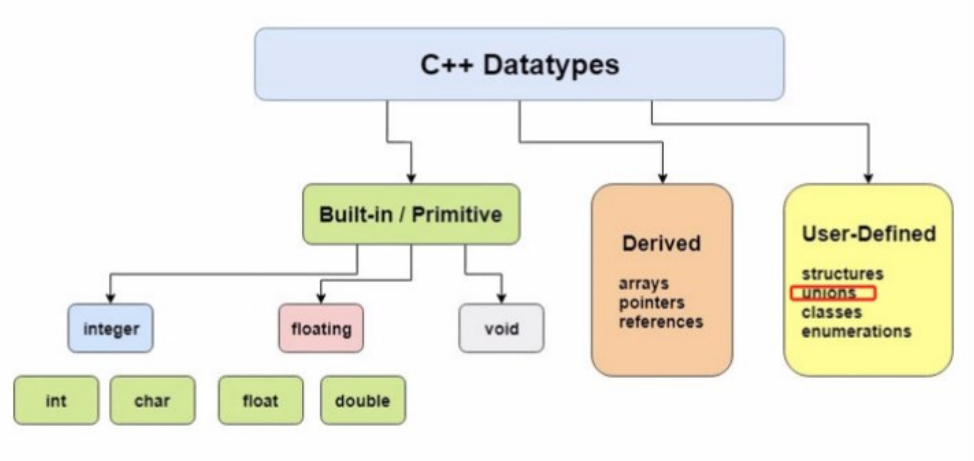
- *unique_ptr*. unique_ptr memorizza un solo puntatore. È possibile assegnare un oggetto diverso rimuovendo l'oggetto corrente dal puntatore. Notate il codice sottostante. All'inizio, il puntatore unique_pointer punta a P1. Ma poi si rimuove P1 e si assegna P2, per cui il puntatore ora punta a P2.

- *shared_ptr*. Usando shared_ptr, più di un puntatore può puntare a questo oggetto alla volta e manterrà un contatore di riferimenti usando il metodo use_count().

- *weak_ptr*. È molto più simile a shared_ptr, ma non mantiene un contatore di riferimenti. In questo caso, un puntatore non avrà una posizione di forza sull'oggetto. Il motivo è che se più puntatori mantengono l'oggetto e chiedono altri oggetti, possono formare un deadlock.

Tipi di dati C++, tipi di conversioni, tipi di cast, Template, template di classe, Friend e classi annidate

I tipi di dati di C++ sono i seguenti:



Per esempio, vediamo i *tipi primitivi*, cioè tutti i tipi predefiniti:

Data Type	Size (bytes)	Size (bits)	Value Range
unsigned char	1	8	0 to 255
signed char	1	8	-128 to 127
char	1	8	either
unsigned short	2	16	0 to 65,535
short	2	16	-32,768 to 32,767
unsigned int	4	32	0 to 4,294,967,295
int	4	32	-2,147,483,648 to 2,147,483,647
unsigned long	8	64	0 to 18,446,744,073,709,551,616
long	8	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8	64	0 to 18,446,744,073,709,551,616
long long	8	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	32	3.4E +/- 38 (7 digits)
double	8	64	1.7E +/- 308 (15 digits)
long double	8	64	1.7E +/- 308 (15 digits)
bool	1	8	false or true

Seguono i *tipi integrali*, in grado di gestire tutte le quantità numeriche:

Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
short	short int	at least 16	16	16	16	16
short int						
signed short						
signed short int						
unsigned short	unsigned short int					
unsigned short int						
int	int	at least 16	16	32	32	32
signed						
signed int						
unsigned	unsigned int					
unsigned int						
long	long int	at least 32	32	32	32	64
long int						
signed long						
signed long int						
unsigned long	unsigned long int					
unsigned long int						
long long	long long int (C++11)	at least 64	64	64	64	64
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int (C++11)					
unsigned long long int						

Successivamente, i tipi a virgola mobile:

Floating point types

`float` - single precision floating point type. Usually IEEE-754 32 bit floating point type

`double` - double precision floating point type. Usually IEEE-754 64 bit floating point type

`long double` - extended precision floating point type. Does not necessarily map to types mandated by IEEE-754. Usually 80-bit x87 floating point type on x86 and x86-64 architectures.

Spesso serve operare le conversioni di tipo, tale da poter passare facilmente da un tipo all'altro, di qualsiasi natura, per poter avere a che fare in modo vero e proprio con oggetti.

Conversioni di tipo

- Conversioni **implicite** (coercions)
- Conversioni **esplicite**

- Conversioni **predefinite** dal linguaggio
- Conversioni **definite dall'utente**

- Conversioni **con/senza perdita** di informazione (narrow/wide conversions)

La conversione implicita è normalmente una semplice inizializzazione, tale che compili:

An expression **e** is said to be ***implicitly convertible*** to **T** if and only if **T** can be copy-initialized from **e**, that is, the declaration

```
T t=e;
```

can be compiled

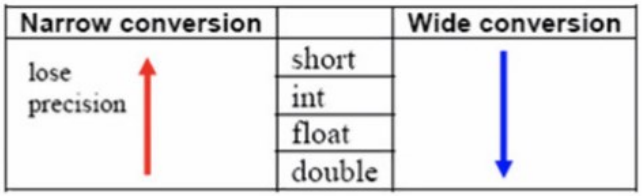
Vediamo le varie conversioni implicite (safe/castless conversions)
(notare che il puntatore generico è void*)

```
T& => T          // e non viceversa     int& x = 5;

T[] => T*           int[2] a={3,1}; int* p = a;

T* => void*      // generic pointer: int* p=&x; void* q=p;

T => const T      int x=5; const int y=x;
const NPR => NPR // NPR = Tipo NON Puntatore o
                 //                 Riferimento
                 // In particolare: C* const => C*
const int x = 5; int y = x;
int* const p = &z; int* q = p;

T* => const T*      int* p = &x; const int* q = p;

T => const T&          int x=4; const int& r = x;
// TRA TIPI PRIMITIVI
bool => int
float => double => long double
char => short int => int => long
unsigned char => ... => unsigned long
```

Esistono due tipi di conversioni: quelle *narrow* (strette) e quelle *wide* (larghe).

- Una conversione wide modifica un valore in un tipo di dati che può consentire qualsiasi valore possibile dei dati originali. Le conversioni di ampliamento conservano il valore di partenza, ma possono modificarne la rappresentazione. Ciò si verifica se si converte da un tipo integrale a Decimale o da Char a String.

- Una conversione narrow cambia un valore in un tipo di dati che potrebbe non essere in grado di contenere alcuni dei valori possibili. Ad esempio, un valore frazionario viene arrotondato quando viene convertito in un tipo integrale e un tipo numerico convertito in booleano viene ridotto a Vero o Falso.




```
// esempio di narrowing conversion
double d = 3.14;
int x = static_cast<int>(d);
// esempio di wide conversion (coercion)
char c = 'a';
int x = static_cast<int>(c);
// esempio di conversione T* => void*
void* p;
p=&d;
// per la conversione di void* serve uno static_cast
double* q = static_cast<double*>(p);
```

Altri tipi di cast:

```
const_cast <Type> (puntatore/riferimento)
```

`const_cast` permette di convertire un puntatore o un riferimento ad un tipo `const T` ad un puntatore o riferimento a `T` (quindi perdendo l'attributo `const`).

```
const int i = 5;
int* p = const_cast<int*> (&i);

void F(const C& x) {
  x.metodoCostante();
  const_cast<C&>(x).metodoNonCostante();
}

int j = 7;
const int* q = &j; // OK, cast implicito
```

```
reinterpret_cast <T*> (puntatore)
```

`reinterpret_cast` si limita a reinterpretare a basso livello la sequenza di bit con cui è rappresentato il valore puntato da `puntatore` come fosse un valore di tipo `T`. Questo tipo di cast è particolarmente pericoloso

```
Classe c;
int* p = reinterpret_cast<int*>(&c);
const char* a = reinterpret_cast<const char*>(&c);
string s(a);
cout << s;
```

Prima di introdurre il caso d'uso dei template, prendiamo l'esempio di una funzione che ritorna il minimo tra due variabili di un tipo che ammette il test booleano *operator<*.

```
int min (int a, int b) {
  return a < b ? a : b;
}
```

```
float min (float a, float b) {
  return a < b ? a : b;
}
```

```
orario min (orario a, orario b) {
  return a < b ? a : b;
}
```

```
string min (string a, string b) {
  return a < b ? a : b;
}
```

Una soluzione attraente ma subdolamente pericolosa potrebbe essere la definizione di **macro** per il preprocessore.

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

il preprocessore sostituisce:

`min(10,20)` con `10 < 20 ? 10 : 20`

`min(3.5,2.1)` con `3.5 < 2.1 ? 3.5 : 2.1`

Non abbiamo ottenuto una "vera funzione"

```
min(++i,--j)
```

verrebbe espansa dal preprocessore con:

```
++i < --j ? ++i : --j
```

e provocherebbe una doppia applicazione di ++ o --.

```
int i=3, j=6;
cout << min(++i,--j);
// stampa 5
// NON stampa 4
```

Il paradigma è quello della programmazione generica; creare degli algoritmi con dei tipi che saranno specificati successivamente (types-to-be-specified-later) e poi istanziati (instantiated).

Qui arrivano i *template*, letteralmente dei modelli.
Un template è uno strumento semplice ma molto potente in C++. L'idea semplice è quella di passare il tipo di dati come parametro, in modo da non dover scrivere lo stesso codice per tipi di dati diversi. Ad esempio, una società di software potrebbe aver bisogno di ordinare() per diversi tipi di dati. Invece di scrivere e mantenere più codici, possiamo scrivere un solo sort() e passare il tipo di dati come parametro.
Il C++ aggiunge due nuove parole chiave per supportare i modelli: 'template' e 'typename'. La seconda parola chiave può sempre essere sostituita dalla parola chiave 'class'.

Ci può essere il tipo a cui si riferisce/che modella il template → istanziazione esplicita
oppure potrebbe essere implicito → istanziazione implicita

Definizione della funzione `min` come template:

```
template <class T> // oppure: template <typename T>
T min(T a, T b) {
  return a < b ? a : b;
}
```

Quindi:

```
int main() {
  int i,j,k;
  orario r,s,t;
  ...
  // istanziazione implicita del template
  k = min(i,j);
  t = min(r,s);
  // oppure: istanziazione esplicita del template
  k = min<int>(i,j);
  t = min<orario>(r,s);
}
```

- I parametri di un template possono essere:

- **Parametri di tipo**: si possono istanziare con un tipo qualsiasi

- **Parametri valore di qualche tipo**: si possono istanziare con un valore costante del tipo indicato

- Un template **non è codice compilabile**: istanziazione **implicita** o **esplicita** di un template di funzione

- **Processo di deduzione degli argomenti** di un template nella istanziazione implicita (dove il **tipo di ritorno non si considera mai**)

Attenzione: nell'istanziazione implicita il **tipo di ritorno** dell'istanza del template non viene **mai considerato nella deduzione degli argomenti** (essendo opzionale l'uso del valore ritornato)

```
int main() {
  double d; int i,j;
  ...
  d = min(i,j);   // istanzia int min(int,int)
                  // e quindi usa la conversione
                  // int => double
}
```

Importante anche l'algoritmo di deduzione dei template e le successive tipologie di conversione operate caso per caso (segue).

L'algoritmo di deduzione degli argomenti di un template procede esaminando tutti i parametri attuali passati al template di funzione da sinistra verso destra. Se si trova uno stesso parametro **T** del template che appare più volte come parametro di tipo, l'argomento del template dedotto per **T** da ogni parametro attuale deve essere **esattamente** lo stesso.

```
int main() {
  int i; double d, e;
  ...
  e = min(d,i);
  // NON COMPILA
  // Si deducono due diversi argomenti del
  // template: double e int
}
```

L'istanziazione dei parametri di tipo **deve essere univoca**.

Nell'algoritmo di deduzione degli argomenti sono ammesse **quattro tipologie di conversioni** dal tipo dell'argomento attuale al tipo dei parametri del template:

1) conversione da lvalue in rvalue, i.e. da `T&` a `T`;
2) da array a puntatore, i.e. da `T[]` a `T*`;
3) conversione di qualificazione costante, i.e. da `T` a `const T`;
4) conversione da rvalue a riferimento costante, i.e. da rvalue di tipo `T` a `const T&`

```
template <class T> void E(T x) {...};
template <class T> void F(T* p) {...};
template <class T> void G(const T x) {...};
template <class T> void H(const T& y) {...};

int main() {
  int i = 6; int& x = i;
  int a[3] = {4,2,9};
  E(x);          // (1): istanzia void E(int)
  F(a);          // (2): istanzia void F(int*)
  G(i);          // (3): istanzia void G(const int)
  H(7);          // (4): istanzia void H(const int&)
}
```

Istanziazione esplicita degli argomenti dei parametri del template di funzione. Nell'istanziazione esplicita è possibile applicare ***qualsiasi conversione implicita*** di tipo per i parametri attuali del template di funzione.

```
int main() {
  int i; double d, e; ...
  e = min<double>(d,i);
  // compila!
  // istanzia: double min(double,double)
  // e quindi converte implicitamente i
  // da int a double
}
```

Un parametro di una funzione può essere un *riferimento ad un array statico*. In questo caso, la dimensione costante dell'array è parte integrante del tipo del parametro e il compilatore controlla che la dimensione dell'array parametro attuale coincida con quella specificata nel tipo del parametro.

```
int min(int (&a)[3]) { // array di 3 int
  int m = a[0];
  for (int i = 1; i < 3; i++)
    if (a[i] < m) m = a[i];
  return m;
}

int ar[4] = {5,2,4,2};
cout << min(ar); // non compila!
```

```
template <class T, int size>
T min(T (&a)[size]) {
  T vmin = a[0];
  for (int i = 1; i < size; i++)
    if (a[i] < vmin) vmin = a[i];
  return vmin;
}
```

Parametro valore

```
int main() {
  int ia[20];
  orario ta[50];
  ...
  cout << min(ia);
  cout << min(ta);
  // oppure
  cout << min<int,20>(ia);
  cout << min<orario,50>(ta);

}
```

Come compilare un template:

- Compilazione per inclusione: Definizione del template in un "header file" che deve sempre essere incluso dal codice che necessita di istanziare il template. Non vi è quindi il concetto di compilazione separata di un template.
  Problemi:
  1) No information hiding → No soluzione
  2) Istanze multiple del template → Dichiarazioni esplicite di istanziazione (tipo a cui si riferirà l'intero template)

Dichiarazione esplicita di istanziazione del template di funzione:

```
template <class Tipo>
Tipo min(Tipo a, Tipo b) {
  return a < b ? a : b;
}
```

al tipo `int` ha la forma:

```
template int min(int,int);
```

Forza il compilatore a generare il codice dell'istanza del template relativa al tipo `int`.

Per evitare di istanziare implicitamente i template, esiste l'apposito flag:

-fno-implicit-templates Never emit code for non-inline templates which are instantiated implicitly (i.e. by use); only emit code for explicit instantiations. See Template Instantiation, for more information.

```
g++ -fno-implicit-templates
```

Dichiarazione esplicita di istanziazione del template di funzione:

```
// file min.h
template <class Tipo>
Tipo min(Tipo a, Tipo b) {
  return a < b ? a : b;
}
```

```
// file usedTemplates.cpp
#include "min.h"
template int min(int,int);
template orario min(orario,orario);
```

```
// file main.cpp
#include "min.h"
#include "orario.h"
int main() {
  cout << min(9,3) << min(3*16,50-2);
  cout << min(orario(4), orario(4,5,6));
}
```

```
g++ -fno-implicit-templates -c main.cpp
g++ -fno-implicit-templates -c usedTemplates.cpp
g++ main.o usedTemplates.o
```

- Compilazione per separazione: Dichiarazione del template separata dalla sua definizione. Era fattibile attraverso la keyword *export*, oggi non supportata dai compilatori.

Per analizzare il comportamento di template di classe, dunque un intera classe parte di template, prendiamo l'esempio della struttura dati queue/coda.

Una coda è definita come una struttura di dati lineare che è aperta a entrambe le estremità e le operazioni vengono eseguite in ordine First In First Out (FIFO).
Definiamo una coda come un elenco in cui tutte le aggiunte all'elenco vengono effettuate a un'estremità e tutte le cancellazioni dall'elenco vengono effettuate all'altra estremità. L'elemento che viene spinto per primo nell'ordine, l'operazione viene eseguita per prima su di esso.

Se, come si vede di seguito, volessimo usare sia code di interi che code di stringhe, senza i template avremmo bisogno di due classi separate.

```
class QueueInt {
public:
  Queue();
  ~Queue();
  bool empty() const;
  void add(const int&);
  int remove();
private:
  ...
};
```

```
class QueueString {
public:
  Queue();
  ~Queue();
  bool empty() const;
  void add(const string&);
  string remove();
private:
  ...
};
```

Qui si ha la potenza del template, con l'uso del *generic*o tipo *T*, tale da poter creare un container con qualsiasi tipo possibile. Di seguito, le caratteristiche che ognuno di questi deve avere:

```
template <class T>
class Queue {
public:
  Queue();
  ~Queue();
  bool empty() const;
  void add(const T&);
  T remove();
private:
  ...
};
```

- Parametri di tipo
- Parametri valore
- Parametri tipo/valore con **possibili valori di default**
- Solo **istanziazione esplicita**

```
Queue<int> qi;
Queue<bolletta> qb;
Queue<string> qs;
```

```
template <class Tipo = int, int size = 1024>
class Buffer {
...
};
```

```
Buffer<> ib;              // Buffer<int,1024>

Buffer<string> sb;        // Buffer<string,1024>

Buffer<string,500> sbs; // Buffer<string,500>
```

Proseguiamo nella scrittura del template Queue<T>:

```
template <class T>
class QueueItem {       // per ora classe esterna
public:                 // per ora tutto public
  QueueItem(const T&);
  T info;
  QueueItem* next;
};

template <class T>
class Queue {
public:
  Queue();               // Queue e non Queue<T>
  ~Queue();              // distruzione profonda
  bool empty() const;
  void add(const T&);
  T remove();
private:
  QueueItem<T>* primo;   // QueueItem<T>
  QueueItem<T>* ultimo; // e non QueueItem
};
```

Nella dichiarazione o definizione di un template (di classe o di funzione), possono comparire sia nomi di istanze di template di classe sia nomi di template di classe.

```
template <class T>
int fun(Queue<T>& qT, Queue<string> qs);
// Queue<T> template di classe associato
// Queue<string> istanza di template di classe
```

NB: Il compilatore genera una istanza di un template (di classe o funzione) solo quando è necessario.
Ad esempio, il compilatore non genera l'istanza Queue<int> quando incontra le due seguenti occorrenze del nome dell'istanza:

```
// basta una dichiarazione incompleta di template
template <class T> class Queue;

void Stampa(const Queue<int>& q) {
  Queue<int>* pqi = const_cast< Queue<int>* >(&q);
  ...
}
```

Invece il compilatore è costretto a generare l'istanza del template di classe con

```
template <class T> class Queue {
  // definizione della classe necessaria
};

void Stampa(Queue<int> q) {
  Queue<int> qi; // si genera l'istanza Queue<int>
                 // per la costruzione di default
}
```

Il compilatore è pure costretto a generare l'istanza `Queue<int>` con

```
template <class T> class Queue {
  // definizione
};

void Stampa(const Queue<int>& q) {
  Queue<int>* pqi = const_cast< Queue<int>* >(&q);
  pqi++;
  ...
}
```

perché l'istanza serve per calcolare la quantità `sizeof(Queue<int>)` di cui occorre incrementare il puntatore per eseguire `pqi++`.

In un template di classe, è possibile definire metodi inline:

```
template <class T>
class Queue {
  ...
public:
  Queue() : primo(0), ultimo(0) {}
  ...
};
```

La definizione esterna di un metodo di template richiede come sintassi:

```
template <class T>
class Queue {
  ...
public:
  Queue();
  ...
};
```

```
// definizione esterna
template <class T>
Queue<T>::Queue() : primo(0), ultimo(0) {}
```

Un metodo di un template di classe è un template di funzione. Esso non viene istanziato quando viene istanziata la classe ma *se e solo se* il programma usa effettivamente quel metodo.

Completiamo la definizione del template `Queue<T>`.

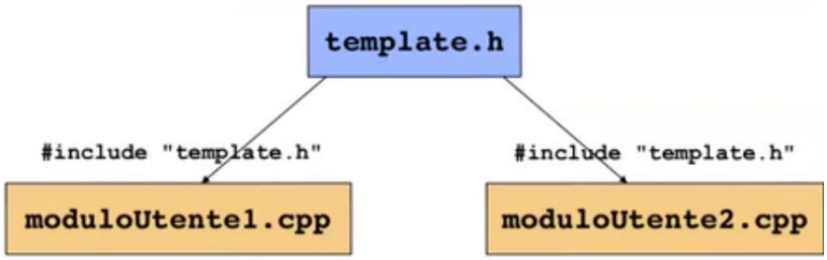
```
// file Queue.h
#ifndef QUEUE_H
#define QUEUE_H

template <class T>
class QueueItem {
public:
  // per gli scopi di Queue basta questo costruttore
  QueueItem(const T& val): info(val), next(0) {}
  T info;
  QueueItem* next;
};
```

```
// sempre nel file Queue.h

template <class T>
class Queue {
public:
  Queue() : primo(0), ultimo(0) {}
  bool empty() const;
  void add(const T&);
  T remove();
  /* Attenzione: distruttore, costruttore di copia e
     assegnazione profondi */
  ~Queue();
  Queue(const Queue&);
  Queue& operator=(const Queue&);
private:
  QueueItem<T>* primo;      // primo el. della coda
  QueueItem<T>* ultimo;     // ultimo el. della coda
};
```

```
// sempre nel file Queue.h

template <class T>
bool Queue<T>::empty() const {
  return (primo == 0);
}

template <class T>
void Queue<T>::add(const T& val) {
  if(empty())
    primo = ultimo = new QueueItem<T>(val);
  else {  // aggiunge in coda
    ultimo->next = new QueueItem<T>(val);
    ultimo = ultimo->next ;
  }
}
```

Viene fatto vedere che per lanciare un errore si può usare *std::cerr*.
Stream di errore standard (cerr): cerr è lo stream di errore standard che viene utilizzato per emettere gli errori. È un'istanza della classe ostream. Poiché lo stream di errore cerr non è bufferizzato, viene utilizzato quando è necessario visualizzare immediatamente il messaggio di errore e non lo si memorizza per visualizzarlo in seguito. L'oggetto della classe ostream rappresenta il flusso di errore standard orientato a caratteri stretti (di tipo char).
La "c" in cerr si riferisce a "carattere" e 'err' significa "errore", quindi cerr significa "errore di carattere". È sempre buona norma utilizzare cerr per visualizzare gli errori.

- Compilazione per inclusione del template di classe



Vediamo mediante un esempio quando vengono create le istanze dei template di classe e dei template dei metodi.

```
#include<iostream>
using std::cout; using std::endl;
#include "Queue.h" // il file Queue.h contiene
                   // le definizioni dei template
int main() {
  Queue<int>* pi = new Queue<int>;
// vengono istanziati la classe Queue<int> ed il suo
// costruttore Queue<int>() perché new deve costruire un
// oggetto della classe
  int i;
  for (i = 0; i < 10; i++) pi->add(i);
// vengono istanziati i metodi add<int>
// e empty<int>, la classe QueueItem<int> e il
// suo costruttore QueueItem<int>()
  for (i = 0; i < 10; i++)
    cout << pi->remove() << endl;
// viene istanziato il metodo remove<int> e
// il distruttore standard ~QueueItem<int>
}
```

Amicizie in template di classe

1) Primo modo: le classi/funzioni friend non template (prendono tutte le istanze)

Dichiarazione nel template di classe **C** di una classe o funzione friend **non template**

```
class A { ..... int fun(); ..... };

template<class T>
class C {
  friend int A::fun();
  friend class B;
  friend bool test();
};
```

La classe `B`, la funzione `test()` e il metodo `A::fun()` della classe `A` sono `friend` di **tutte le istanze** del template di classe `C`.

Essa possiede la seguente logica sintattica:

Dichiarazione nel template di classe **C** di un template di classe `A` o di funzione `fun` friend **associato**

```
template <class U1,...,class Uk> class A;
template <class V1,...,class Vj> void fun(...);

template <class T1,...,class Tn> class C {
     friend class  A<...,Tj,...>;
     friend void fun<...,Tj,...>(...);
};
```

$\subseteq \{T1, \ldots, Tn\}$


```
template<class T>
class C {
private:
  T t;
public:
  C(const T&);
  friend void f_friend<T>(const C<T>&);
  // amicizia associata
};

template<class T>
C<T>::C(const T& x) : t(x) {}

template<class T>
void f_friend(const C<T>& c){
  cout << c.t << endl; // per amicizia
}

int main(){
  C<int> c1(1); C<double> c2(2.5);
  f_friend(c1); // stampa: 1
  f_friend(c2); // stampa: 2.5
}
```

2) Secondo modo: template di classe/funzione friend *non associato* (quindi, esterna dal template in uso rispetto alla classe considerata).

```
template<class T>
class C {

  template<class Tp>        // amicizia con template
  friend int A<Tp>::fun(); // di metodo

  template<class Tp>        // amicizia con template
  friend class B;           // di classe

  template<class Tp>        // amicizia con template
  friend bool test(C<Tp>); // di funzione
};
```

Alcuni compilatori pre-standard non supportavano quest'ultima tipologia di dichiarazioni `friend`. Il compilatore GNU `g++` supporta i template di classe e di funzione friend non associati.

```
template <class T>
class C {
  template <class V>
  friend void fun(const C<V>&); // amicizia non associata
private:
  T x;
public:
  C(const T& y): x(y) {}
};

template <class T>
void fun(const C<T>& t) {
  cout << t.x << " ";  // per amicizia (associata)
  C<double> c(3.1);
  cout << c.x << endl; // per amicizia NON ASSOCIATA
}

int main(){
  C<int> c(4);
  C<string> s("pippo");
  fun(c); // stampa: 4 3.1, istanz.implicita fun<int>
  fun(s); // stampa: pippo 3.1, istanz.implicita fun<string>
}
```

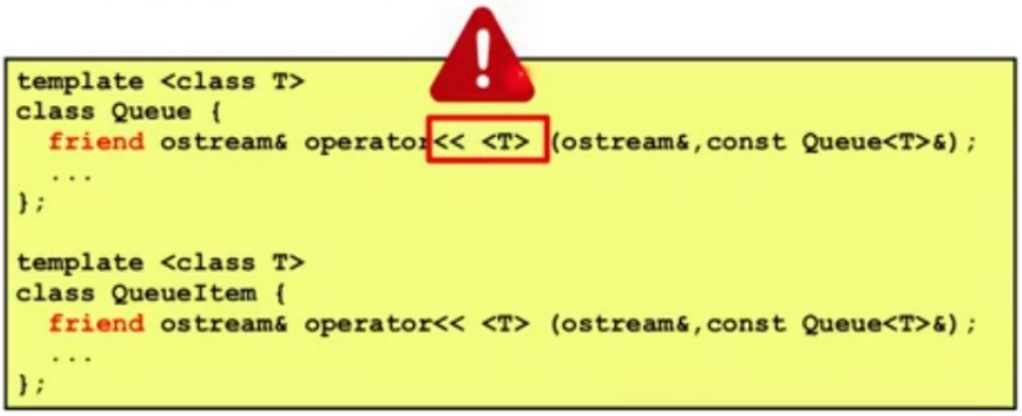
É naturale associare ad ogni istanza di *QueueItem* una sola istanza amica della classe Queue, ovvero quella associata.

```
template <class T>
class QueueItem {
friend class Queue<T>;
private:
  T info;
  QueueItem* next;
  QueueItem(const T& val) : info(val), next(0) {}
};
```

Caso particolare: l'operatore di stampa.
Serve un amicizia con tutti i tipi considerati (tipo container e sottotipo ed operatore di stampa definito sul sottotipo):

```
template <class T>
ostream& operator<<(ostream& os,const Queue<T>& q) {
  os << "(";
  QueueItem<T>* p = q.primo;  // amicizia con Queu
  for (; p != 0; p = p->next) // amicizia con QueueItem
    os << *p << " "; // operator<< per il tipo QueueItem
  os << ")" << endl;
  return os;
}
```

Dobbiamo quindi dichiarare `operator<<` come funzione amica associata sia della classe `Queue` che della classe `QueueItem`.



```
template <class T>
class Queue {
  friend ostream& operator<< <T> (ostream&,const Queue<T>&);
  ...
};

template <class T>
class QueueItem {
  friend ostream& operator<< <T> (ostream&,const Queue<T>&);
  ...
};
```

Inoltre, dobbiamo definire `operator<<` per il template di classe `QueueItem`.

```
template <class T>
ostream& operator<<(ostream& os,const QueueItem<T>& qi){
  os << qi.info; // amicizia con QueueItem!
                 // NB: richiede operator<< sul tipo T
  return os;
}
```

Dobbiamo quindi dichiarare la funzione esterna `operator<<` di `QueueItem` come amica associata della classe `QueueItem`.

```
template <class T>
class QueueItem {
  friend ostream& operator<< <T>(ostream&,const QueueItem<T>&);
  ...
};
```

Campi dati statici in template di classe

Un campo dati statico intero che funzioni da contatore globale degli oggetti *QueueItem* presenti nelle liste di tutti gli oggetti di una certa istanza di *Queue*.

```
template <class Tipo>
class Queue {
  private:
  static int contatore;
  ...
};
```

L'inizializzazione esterna ha come forma:

```
template <class T>
int Queue<T>::contatore = 0;
```

Un campo dati statico è istanziato e quindi inizializzato dalla definizione del template **soltanto se viene effettivamente usato**. La mera definizione di un campo dati statico non provoca allocazione di memoria.

```
template<int I> // parametro valore
class C {
static int numero;
public:
  C() {++numero;}
  C(const C& x) {++numero;}
  static void stampa_numero();
};

// inizializzazione parametrica del campo statico
template<int I>
int C<I>::numero = I;

template<int I>
void C<I>::stampa_numero()
{ cout << "Valore statico: " << numero << endl; }

int main() {
  C<1> uno;
  C<2> due_a;
  C<2> due_b(due_a);
  C<1>::stampa_numero(); // stampa: 2
  C<2>::stampa_numero(); // stampa: 4
}
```

```
class A {
public:
  A(int x=0) {cout << x << "A() ";}
};

template<class T>
class C {
public:
  static A s;
};
template<class T>
A C<T>::s=A(); // stampa 0A()

int main() {
  C<double> c;
  C<int> d;
  C<int>::s = A(2);
}
// stampa: 0A() 2A()
// NOTA BENE: non stampa 0A() 0A() 2A() !!
```

In merito alle classi annidate, ora si fa riferimento allo stesso discorso di sopra, ma con i template di mezzo. Per esempio, annidiamo nella parte privata del template di classe *Queue* la definizione del template di classe *QueueItem*. *QueueItem <T>* è un cosiddetto tipo implicito, in quanto non è un tipo completamente definito ma dipende implicitamente dai parametri di tipo *Queue <T>*.

```
template <class T>
class Queue {
private:
  // template di classe annidato associato
  class QueueItem {
  public:
    QueueItem(const T& val);
    T info;
    QueueItem* next;
  };
  ...
};
```

Tipi e template impliciti in template di classe

```
template <class T>
class C {
public:
  class X {};          // template di classe annidata associato
                       // potrebbe usare il parametro di tipo T

  class D {            // template di classe annidata associato
  public:              // usa il parametro di tipo T
    T x;
  };

  template <class U>
  class E {            // template di classe annidata
  public:              // non associato
    T x;               // usa T del template contenitore
    U y;               // usa il suo parametro di tipo U
    void fun1() {return;}
  };

  template <class U>
  void fun2() {        // template di metodo di istanza
    T x; U y; return; // non associato
  }
};
```

```
template <class T>
  // C<T>::D è un uso di un tipo che
  // effettivamente dipende dal parametro T
void templateFun(typename C<T>::D d) {

  // C<T>::X è un uso di un tipo che
  // comunque dipende dal parametro T
  typename C<T>::X x;

  // C<T>::D è un uso di un tipo che
  // effettivamente dipende dal parametro T
  typename C<T>::D d2 = d;

  // (1) E<int> è un uso del template di classe
  //     annidata che dipende dal parametro T
  // (2) C<T>::E<int> è un uso di un tipo
  //     che dipende dal parametro T
  typename C<T>::template E<int> e;
  e.fun1();

  // c.fun2<int> è un uso del template di funzione
  // che dipende dal parametro T
  C<T> c;
  c.template fun2<int>();
}
```

In alcuni casi, predichiarare (forward declarations) dei template non è permesso: specie quando si ha a che fare con l'operatore di stampa.

```
// dichiarazione incompleta (alias forward declaration)
template<class T> class Queue;

// dichiarazione del template di funzione operator<< <T>
template<class T> std::ostream& operator<<(std::ostream& os,const Queue<T>&);

// dichiarazione incompleta non permessa
// template<class T> class Queue<T>::QueueItem;

// dichiarazione non necessaria
// template<class T> ostream& operator<<(ostream& os,const typename Queue<T>::QueueItem&);

template<class T> class Queue {
  // NON agisce da dichiarazione di template di funzione
  friend std::ostream& operator<< <T> (std::ostream& os, const Queue<T>&);
private:
  class QueueItem {
    friend class Queue<T>;
    friend std::ostream& operator<< <T> (std::ostream& os, const Queue<T>&);
    // agisce da dichiarazione di template di funzione
    friend std::ostream& operator<< <T> (std::ostream& os, const typename Queue<T>::QueueItem&);
  private:
    T info;
    QueueItem* next;
  };
  QueueItem *primo, *ultimo;
public:
  // ...
};

// definizione del template di funzione operator<< <T>
template<class T> std::ostream& operator<< (std::ostream& os, const Queue<T>& q) {
  // ...
  return os;
}

// definizione del template di funzione operator<< <T>
template<class T> std::ostream& operator<< (std::ostream& os, const typename Queue<T>::QueueItem& qi) {
  // ...
  return os;
}
```

Scritto da Gabriel

È possibile dichiarare argomenti predefiniti per un modello solo per la prima dichiarazione del modello. Se si vuole consentire agli utenti di inoltrare la dichiarazione di un template di classe, si deve fornire un header di inoltro.

Definire un template di classe `albero<T>` i cui oggetti rappresentano un **albero 3-ario** in cui i nodi memorizzano dei valori di tipo `T` ed hanno 3 figli (invece dei 2 figli di un usuale albero binario). Il template `albero<T>` deve soddisfare i seguenti vincoli:

1. Deve essere disponibile un costruttore di default che costruisce l'albero vuoto.
2. Gestione della memoria senza condivisione.
3. Metodo `void insert(const T&)`: in una chiamata `a.insert(t)`, inserisce nell'albero `a` una nuova radice che memorizza il valore `t`, ed avente come figli 3 copie di `a`.
4. Metodo `bool search(const T&)`: in una chiamata `a.search(t)` ritorna true se il valore `t` occorre nell'albero `a`, altrimenti ritorna false.
5. Overloading dell'operatore di uguaglianza.
6. Overloading dell'operatore di output.

```
/*
Definire un template di classe albero<T> i cui oggetti rappresentano
un albero 3-ario ove i nodi memorizzano dei valori di tipo T ed hanno
3 figli (invece dei 2 figli di un usuale albero binario). Il template
albero<T> deve soddisfare i seguenti vincoli:
1. Deve essere disponibile un costruttore di default che costruisce l'albero vuoto.
2. Gestione della memoria senza condivisione.
3. Metodo void insert(const T&): a.insert(t) inserisce nell'albero a una nuova radice che memorizza il
valore t ed avente come figli 3 copie di a
4. Metodo bool search(const T&): a.search(t) ritorna true se t occorre nell'albero a, altrimenti ritorna
false.
5. Overloading dell'operatore di uguaglianza.
6. Overloading dell'operatore di output.
*/

#include<iostream>

template <class T> class albero; // dichiarazione incompleta

template<class T>
std::ostream& operator<<(std::ostream& os, const albero<T>& a);

template <class T>
class albero {
  friend std::ostream& operator<< <T> (std::ostream&, const albero&);
  private:
  // classe annidata associata
  class nodo {
  public:
    T info;
    nodo *sx, *cx, *dx;
    nodo(const T& x, nodo* s =0, nodo* c =0, nodo* d =0):
      info(x), sx(s), cx(c), dx(d) {}
  };

  nodo* root;

  // copia profonda ricorsiva
  static nodo* copia(nodo* r) {
    if(!r) return nullptr;
    // albero non vuoto
    return new nodo(r->info, copia(r->sx), copia(r->cx), copia(r->dx));
  }

  // distruzione profonda ricorsiva
  static void distruggi(nodo* r) {
    if(r) {
      distruggi(r->sx); distruggi(r->cx); distruggi(r->dx);
      delete r;
    }
  }
```

```
static bool search_rec(nodo* r, const T& t) {
  if(!r) return false;
  // r punta alla radice di un albero non vuoto
  return r->info == t || search_rec(r->sx,t) || search_rec(r->cx,t) || search_rec(r->dx,t);
}

static bool equal_rec(nodo* r1, nodo* r2) {
  if(!r1 && !r2) return true;
  // r1 | r2
  if(!r1 || !r2) return false;
  // r1 & r2, T deve supportare operator==
  return r1->info == r2->info && equal_rec(r1->sx,r2->sx) &&
    equal_rec(r1->cx,r2->cx) && equal_rec(r1->dx,r2->dx);
}

static std::ostream& print_rec(std::ostream& os, nodo* r){
  // caso base: albero vuoto
  if(!r) return os;
  // passo induttivo: albero non vuoto
  os << r->info << " ";  // T deve supportare operator<<
  print_rec(os,r->sx);
  print_rec(os,r->cx);
  return print_rec(os,r->dx);
}


public:
  albero(): root(nullptr) {}

  albero(const albero& a): root(copia(a.root)) {}

  albero& operator=(const albero& a) {
    if(this != &a) {
      if(root) distruggi(root);
      root = copia(a.root);
    }
    return *this;
  }

  ~albero() {if(root) distruggi(root);}

  void insert(const T& x) {
    root = new nodo(x,copia(root), copia(root), root);
  }

  bool search(const T& t) const {
    return search_rec(root,t);
  }

  bool operator==(const albero& a) const {
    return equal_rec(root,a.root);
  }
};

template<class T>
std::ostream& operator<<(std::ostream& os, const albero<T>& a) {
  return albero<T>::print_rec(os,a.root);
}

int main() {
  albero<char> t1, t2, t3;
  t1.insert('b');
  t1.insert('a');
  t2.insert('a');
  t3 = t1;
  t3.insert('c');
  std::cout << (t1 == t2) << std::endl;
  std::cout << t1.search('b') << std::endl;
  std::cout << t1 << std::endl << t2 << std::endl << t3 << std::endl;
}
```

```
/*
Si considerino le seguenti definizioni. Fornire una dichiarazione
(non e` richiesta la definizione) dei membri pubblici della classe Z
nel minor numero possibile in modo tale che la compilazione del
main() non produca errori. Attenzione: ogni dichiarazione in Z
non necessaria per la corretta compilazione del main() e'
penalizzata.
*/

class Z {
public:
  int& operator++();
  int operator++(int);
  bool operator==(const Z&) const;
  Z(const int&); // agisce da convertitore int => Z
};

template <class T1, class T2 =Z>
class C {
public:
  T1 x;
  T2* p;
};

template<class T1,class T2>
void fun(C<T1,T2>* q) {
  ++(q->p); // nessun requirement
  if(true == false) cout << ++(q->x); // q->x di tipo T1, operator++() T1
  else cout << q->p; // nessun requirement
  (q->x)++; // operator++(int) su T1
  if(*(q->p) == q->x) *(q->p) = q->x; // (1) bool operator==(T2,T1), (2) operator=(T2,const T1&)
  T1* ptr = &(q->x); // nessun requirement
  T2 t2 = q->x; // T2(const T1&)
}

main(){
  C<Z> c1; fun(&c1); C<int> c2; fun(&c2);
  // C<Z,Z> c1;
  // fun<Z,Z>, i.e. T1=Z, T2=Z
  // C<int,Z> c2;
  // fun<int,Z>, i.e. T1=int, T2=Z
}
```

C++ Libreria Standard, vector e metodi, contenitori, Ereditarietà, Static Binding, Basi, derivate, costruttori, operatori e loro definizione

Abbiamo introdotto i template non a caso: infatti caratterizziamo una serie di classi container e relativi iteratori. *La Standard Template Library (STL)* è un insieme di classi template del C++ per fornire strutture dati e funzioni comuni nella programmazione, come liste, stack, array e così via. È una libreria di classi contenitore, algoritmi e iteratori. È una libreria generalizzata e quindi i suoi componenti sono parametrizzati. La conoscenza delle classi template è un prerequisito per lavorare con STL.
STL ha 4 componenti:

- Algorithms (template di funzione)
- Containers (template di classe)
- Functions
- Iterators

Una buonissima documentazione su STL è data da:
https://www.boost.org/sgi/stl/stl_introduction.html
https://cplusplus.com/reference/stl/
https://www.cppreference.com/Cpp_STL_ReferenceManual.pdf

Approfondiamo la classe più importante di STL: il *vector*.

I vectors sono uguali agli array dinamici, con la capacità di ridimensionarsi automaticamente quando un elemento viene inserito o cancellato, e la loro memorizzazione è gestita automaticamente dal contenitore. Gli elementi dei vettori sono collocati in una memoria contigua, in modo che sia possibile accedervi e attraversarli utilizzando gli iteratori. Nei vettori, i dati vengono inseriti alla fine. L'inserimento alla fine richiede un tempo differenziale, poiché a volte è necessario estendere l'array. La rimozione dell'ultimo elemento richiede solo un tempo costante, perché non avviene alcun ridimensionamento. L'inserimento e la cancellazione all'inizio o al centro richiedono un tempo lineare.

- begin() - Restituisce un iteratore che punta al primo elemento del vettore.
- end() - Restituisce un iteratore che punta all'elemento teorico successivo all'ultimo elemento del vettore.
- rbegin() - Restituisce un iteratore inverso che punta all'ultimo elemento del vettore (inizio inverso). Si sposta dall'ultimo al primo elemento
- rend() - Restituisce un iteratore inverso che punta all'elemento teorico che precede il primo elemento del vettore (considerato come fine inverso)
- cbegin() - Restituisce un iteratore costante che punta al primo elemento del vettore.
- cend() - Restituisce un iteratore costante che punta all'elemento teorico che segue l'ultimo elemento del vettore.
- crbegin() - Restituisce un iteratore costante inverso che punta all'ultimo elemento del vettore (inizio inverso). Si sposta dall'ultimo al primo elemento
- crend() - Restituisce un iteratore costante inverso che punta all'elemento teorico che precede il primo elemento del vettore (considerato come fine inverso).

vector è il più semplice contenitore di STL e per la maggioranza delle applicazioni è il più efficiente. È un template di classe che generalizza gli array dinamici.

Caratteristiche di vector:

1) è un contenitore che supporta **l'accesso casuale** agli elementi (accesso in posizione arbitraria in **tempo costante**)

2) inserimento e rimozione in **coda** in **tempo ammortizzato costante**

3) inserimento e rimozione **arbitraria** in **tempo lineare ammortizzato**

4) la **capacità** di un vector può variare dinamicamente

5) la gestione della memoria è **automatica**

Ci sono due modi di usare un *vector*: lo stile array ereditato dal C e lo stile STL più costoso del C++.

```
int a[10];          // array
vector<int> v(10); // costruttore ad 1 argomento
                    // vector(size_type)
                    // con costruzione di default per
                    // gli elementi
```

Accediamo agli elementi con l'operatore di subscripting (*operator[]*).

```
int n = 5;
vector <int> v(n);
int a[n] = {2,4,5,2,-2};
for (int i = 0; i < n; i++)
  v[i] = a[i] + 1;
```

```
vector<string> v(10), w;
cout << v.size() << " " << v.capacity(); // 10 10
vector<string> u(v); // costruzione di copia
w = u;               // assegnazione
```

Il metodo `size()` ritorna il numero di elementi contenuti nel vector.

Il metodo `capacity()` ritorna invece la capacità del vector.

invariante: `v.size() <= v.capacity() == true`

```
template <class T>
void stampa(const vector<T>& v) {
  for (int i = 0; i < v.size(); i++)
    cout << v[i] << endl;
}
```

- Il costruttore *vector(size_type*) costruisce un vector in cui gli elementi sono inizializzati con il costruttore di default.
- Il costruttore *vector (size_type n, const T& t)* permette invece di specificare un valore iniziale *t* da cui sono costruiti di copia tutti gli elementi.

```
vector <int> ivec(10,-1);
```

In C++03 **non** è possibile inizializzare un `vector` con una data sequenza di valori, diventa invece possibile in C++11.

```
int ia[6] = {-1,5,-7,0,12,3};              // OK
vector <int> ivec(6) = {-1,5,-7,0,12,3}; // NO C++03
vector <int> ivec(6) = {-1,5,-7,0,12,3}; // OK C++11
```

Vediamo tutti i metodi fondamentali di *vector*:

```
void push_back(const T&)
void pop_back()
T& front()
T& back()
iterator begin()
iterator end()
```

Metodi fondamentali

1) void push_back(const T&): principale metodo di inserimento in coda ad un vector, inserendo l'elemento con il costruttore di copia

```
void push_back (const value_type& val);
```

Add element at the end

Adds a new element at the end of the vector, after its current last element. The content of *val* is copied (or moved) to the new element.

This effectively increases the container size by one, which causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

```
int main() {
 vector<string> sv; string x;
 while (cin >> x) sv.push_back(x);
 // legge stringhe da cin, separate da spazi, tab o enter
 // fino a end_of_file e le inserisce in coda a sv.
 // Da tastiera end_of_file si invia con una
 // combinazione di tasti particolare:
 // normalmente <Ctrl>+<d>
 cout << endl << "Abbiamo letto:" << endl;
 for (int i = 0; i < sv.size(); i++)
   cout << sv[i] << endl;
}
```

La *size* non può differire tra più compilatori. La dimensione di un vettore è il numero di elementi che contiene, controllato direttamente dal numero di elementi inseriti nel vettore.

La *capacity* è la quantità di spazio totale di cui dispone il vettore. Sotto il cofano, un vettore utilizza semplicemente un array. La capacità del vettore è la dimensione dell'array. È sempre uguale o maggiore della dimensione. La differenza tra i due è il numero di elementi che si possono aggiungere al vettore prima che l'array debba essere riallocato.

La capacità non dovrebbe quasi mai interessare. Esiste per consentire alle persone con vincoli di prestazioni e di memoria molto specifici di fare esattamente ciò che vogliono.

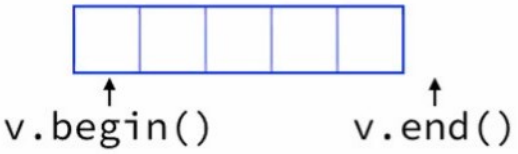
```
#include <iostream>
#include <vector>

int main () {
std::vector<int> myvector;

for (int i=0; i<100; i++) myvector.push_back(i);

std::cout << "size: " << myvector.size() << '\n';
std::cout << "capacity: " << myvector.capacity() << '\n';
}
```

Similmente, possiamo usare i vector per elaborare stringhe di file, anche separati da spazi, tab, enter:

```
int main() {
 vector<string> sv;
 string x;
 ifstream file("dati.txt",ios:in);
 while (file >> x) sv.push_back(x);

 cout << "Abbiamo letto:" << endl;
 vector<string>::iterator it;
 for (it = sv.begin(); it != sv.end(); it++)
   cout << *it << endl;
}
```

Metodi già visti fin qui i successivi:

`sv.begin()` e `sv.end()` sono degli oggetti della classe **iteratore** su vector.



Questi sono gli *iteratori*.
Gli iteratori sono oggetti simili a puntatori che vengono utilizzati per iterare su una sequenza e manipolare gli elementi del contenitore. Il vantaggio dell'uso di un iteratore è che riduce le righe di codice a una sola istruzione, in quanto consente di manipolare gli array incorporati nell'STL usando i puntatori come iteratori. Un iteratore può essere costante o non costante/regolare.

Ad ogni classe contenitore **C** della STL sono associati due tipi iteratore

`C::iterator`

`C::const_iterator`

Si usa **`iterator`** quando si necessita un accesso agli elementi del contenitore come lvalue (in lettura e scrittura), se basta un accesso come rvalue (in sola lettura) si preferisce la protezione di **`const_iterator`**.

Si tratta di cosiddetti ***iteratori bidirezionali***.

Iteratori costanti/const_iterator:
Un iteratore const punta a un elemento di tipo costante, il che significa che l'elemento a cui punta un iteratore const non può essere modificato. È comunque possibile aggiornare l'iteratore (cioè, l'iteratore può essere incrementato o decrementato, ma l'elemento a cui punta non può essere modificato). Può essere usato solo per l'accesso e non può essere usato per la modifica. Se si cerca di modificare il valore dell'elemento utilizzando l'iteratore const, viene generato un errore.

Iteratori regolari/iterator:
Un iteratore regolare o non const punta a un elemento all'interno del contenitore e può essere usato per modificare l'elemento a cui punta. Gli iteratori regolari svolgono un ruolo fondamentale nel collegare l'algoritmo ai contenitori e nella manipolazione dei dati memorizzati al loro interno. La forma più ovvia di iteratore regolare è il puntatore.
Un puntatore può puntare agli elementi di un array e può iterare attraverso di essi usando l'operatore di incremento (++). Ogni tipo di contenitore ha un tipo di iteratore regolare specifico, progettato per iterare i suoi elementi.

L'esempio che segue mostra una chiara differenza (sempre relativa all'accesso di un elemento che deve o non deve, come qui, essere const):

vector<int>::iterator

vector<int>::const_iterator

```
template <class T>
T& vector<T>::iterator::operator*() const;

template <class T>
vector<T>::iterator  vector<T>::begin();
template <class T>
vector<T>::iterator  vector<T>::end();
```

```
template <class T>
const T& vector<T>::const_iterator::operator*() const;

template <class T>
vector<T>::const_iterator  vector<T>::begin() const;
template <class T>
vector<T>::const_iterator  vector<T>::end() const;
```

```
vector<int> v(1); const vector<int> w(2);
iterator it = w.begin();       // ILLEGALE
*(w.begin())=0;                // ILLEGALE
```

Su ogni tipo iteratore (anche const) di qualche istanza di contenitore `Cont<Tipo>::[const_]iterator` sono **sempre disponibili** le seguenti funzionalità:

```
Cont<Tipo> x;
Cont<Tipo>::[const_]iterator i;

x.begin(); // iteratore che punta al primo elemento
x.end();   // particolare iteratore che non punta ad
           // alcun elemento, e' "un puntatore
           // all'ultimo elemento + 1"
*i;        // elemento puntato da i
i++; ++i;  // puntatore all'elemento successivo. Se
           // i punta all'ultimo elemento di x
           // allora ++i == x.end()
i--; --i;  // puntatore all'elemento precedente. Se
           // i punta al primo elemento di x
           // allora i-- è indefinito (x.begin()-1)
           // (v.end())-- punta all'ultimo elemento
```

Piccolo approfondimento per la deque:

(Le code a doppia estremità/deque sono contenitori di sequenze con la caratteristica di espansione e contrazione su entrambe le estremità. Sono simili ai vettori, ma sono più efficienti in caso di inserimento e cancellazione di elementi. A differenza dei vettori, l'allocazione di memoria contigua può non essere garantita.
Le code a doppia estremità sono fondamentalmente un'implementazione della struttura dati coda a doppia estremità. Una struttura di dati a coda consente l'inserimento solo alla fine e la cancellazione dalla parte anteriore. È come una coda nella vita reale, in cui le persone vengono rimosse dalla parte anteriore e

aggiunte dalla parte posteriore. Le code a doppia estremità sono un caso speciale di code in cui le operazioni di inserimento e cancellazione sono possibili a entrambe le estremità.
Le funzioni di deque sono le stesse di quelle dei vettori, con l'aggiunta delle operazioni di push e pop sia per il fronte che per il retro.

Le complessità temporali per l'esecuzione di varie operazioni su deque sono

- Accesso agli elementi - O(1)
- Inserimento o rimozione di elementi - O(N)
- Inserimento o rimozione di elementi all'inizio o alla fine- O(1))

Contenitori ad accesso casuale permettono di avanzare e retrocedere di un numero arbitrario di elementi in tempo costante.

Gli iteratori per i contenitori `vector` e `deque` (**contenitori ad accesso casuale**) permettono di avanzare e di retrocedere di un numero arbitrario di elementi in tempo costante. Sono inoltre disponibili gli operatori di confronto per questi iteratori. Si tratta degli ***iteratori ad accesso casuale***.

```
vector<Tipo> v;              // oppure deque<Tipo>
vector<Tipo>::iterator i,j;
int k;

i += k;
i -= k;
j = i+k;
j = i-k;
i < j;
i <= j;
i > j;
i >= j;
```

Gli iterator normalmente scorrono gli elementi dei container; similmente, nei contenitori usiamo metodi come *empty()* e *size()*.

```
x.empty();   // true se x è vuoto, false altrimenti
x.size();    // numero di elementi contenuti in x
```

È possibile inizializzare un `vector` con un segmento di un array o di un `vector` tramite il **template di costruttore**:

```
template<class InputIterator>
vector(InputIterator,InputIterator)
```

incluso escluso

```
int main() {
  int ia[20];
  vector<int> iv(ia,ia+6); // OK
  cout << iv.size() << endl; // size 6
  vector<int> iv2(iv.begin(),iv.end()-2); // OK
  cout << iv2.size() << endl; // size 4
}
```

Metodi di inserimento in un `vector`:

```
string s;
vector<string> vs, vs1;
vector<string>::iterator i;
...
vs.push_back(s); // aggiunge s in coda al vector
vs.insert(i,s);  // aggiunge s subito prima di *i
vs.insert(i,5,s); // aggiunge 5 s subito prima di *i
i = vs.begin() + vs.size()/2;
vs.insert(i, vs1.begin(), vs1.end());
// inserisce tutti gli elementi di vs1 nella
// posizione mediana di vs
// In generale: v.insert(it1,it2,it3) inserisce
// [it2,it3) in v subito prima di it1
```

```
single element (1)  iterator insert (iterator position, const value_type& val);
fill (2)                 void insert (iterator position, size_type n, const value_type& val);
range (3)           template <class InputIterator>
                         void insert (iterator position, InputIterator first, InputIterator last);
```

Attenzione: Le operazioni di insert() a seconda dell'implementazione della classe vector, possono essere *inefficienti*.

```
vs.insert(i,s);
i = vs.begin() + vs.size()/2;
vs.insert(i, vs1.begin(), vs1.end());
```

Insert elements

The vector is extended by inserting new elements before the element at the specified *position*, effectively increasing the container size by the number of elements inserted.

This causes an automatic reallocation of the allocated storage space if -and only if- the new vector size surpasses the current vector capacity.

Because vectors use an array as their underlying storage, inserting elements in positions other than the vector end causes the container to relocate all the elements that were after *position* to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as list or forward_list).

È anche possibile diminuire la capacity di un vector con il metodo apposito *shrink_to_fit (diminuisci per incastrare).*
Altro metodo ben più utile ed usato di rimozione da un vector (distrugge l'ultimo elemento) è *void pop_back()*.

```
vector<string> v;
vector<string>::iterator i,j;
...
v.pop_back(); // toglie l'ultimo elemento
```

```
iterator erase (iterator position);
iterator erase (iterator first, iterator last);
```

```
vector<string> v;
vector<string>::iterator i,j;
...
v.pop_back(); // toglie l'ultimo elemento
v.erase(i);   // toglie l'elemento puntato da i
v.erase(i,j); // toglie tutti gli elementi
 // compresi tra il puntatore i incluso
 // e il puntatore j escluso,
 // ovvero gli elementi nel segmento [*i,*j)
```

Erase elements

Removes from the vector either a single element (*position*) or a range of elements ([first,last)).

This effectively reduces the container size by the number of elements removed, which are destroyed.

Because vectors use an array as their underlying storage, erasing elements in positions other than the vector end causes the container to relocate all the elements after the segment erased to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as list or forward_list).

Su tutti i contenitori sono definiti tutti gli operatori di confronto (==, !=, <, <=, >, >=)
Il metodo di inserimento *insert()* è disponibile nei suoi overloading per ogni contenitore.
Anche per tutti i contenitori sequenziali, è definita l'operazione di inserimento in coda *push_back*.

```
Cont<Tipo> x;       // x vector, deque, list
Tipo t;
x.push_back(a);     // aggiunge t alla fine di x
```

Similmente, il metodo *void pop_back()* per rimuovere l'ultimo elemento.
Qualsiasi elemento può essere rimosso con il metodo *erase()*.

```
Cont<Tipo> x;
Cont<Tipo>::iterator p,q;

x.erase(p);   // toglie l'elemento puntato da p
x.erase(p,q); // toglie tutti gli elementi
              // nell'intervallo [p,q), cioè
              // da p incluso fino a q escluso
```

In ogni contenitore sequenziale possiamo accedere al primo ed all ultimo elemento con
const T& front() const;
const T& back() const;

Scritto da Gabriel

L'overloading dell'operatore di subscripting [] è disponibile solo per vector e deque; per gli indici è preferibili usare il tipo *size_type* per prendere l'elemento i-esimo (comunque, non usato da noi).

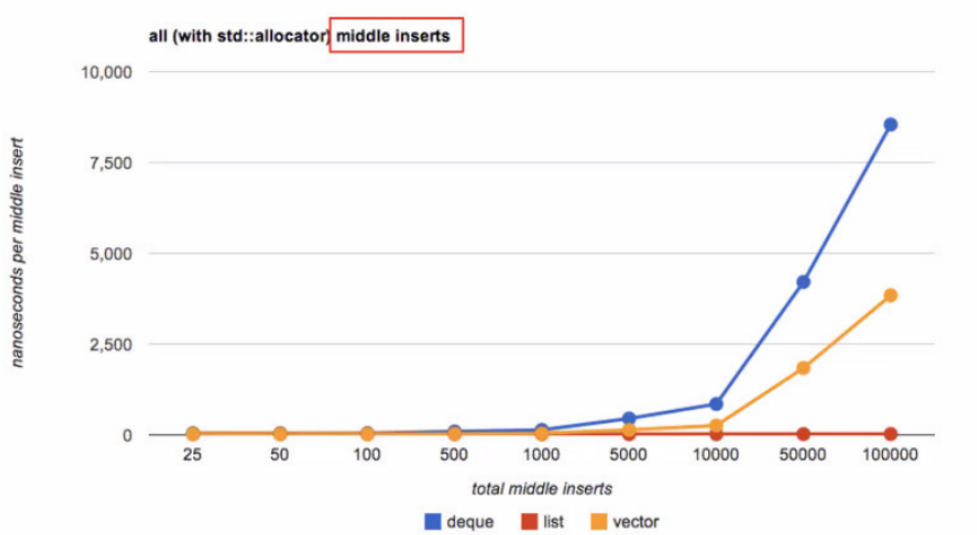
Ora parliamo del tipo *list*.
Le list sono contenitori di sequenze che consentono l'allocazione di memoria non contigua. Rispetto ai vettori, l'elenco ha una traversata lenta, ma una volta trovata una posizione, l'inserimento e la cancellazione sono rapidi. Normalmente, quando si parla di un elenco, si parla di un elenco doppiamente collegato. Per implementare un elenco collegato singolarmente, utilizziamo una lista forward.

Contenitore lista: `list<Tipo>`

E' implementata come una "**doubly-linked list**". A differenza di `vector` permette quindi di eseguire in tempo costante le operazioni di inserimento e rimozione di elementi in posizione arbitraria.

È meno efficiente di `vector` nell'accesso agli elementi: infatti l'implementazione necessariamente deve percorrere la lista dall'inizio (o dalla fine): **non è disponibile** `operator[]`

Una parentesi computazionale tra i vari tipi di container (inserimenti nel mezzo, middle inserts):




```
vector<int> v;
for(int i=0;i<500000;++i) v.insert(v.begin(),0);
// time: 9.31s              FRONT

vector<int> v;
for(int i=0;i<500000;++i) v.insert(v.begin()+v.size()/2,0);
// time: 4.62s              MIDDLE

deque<int> d;
for(int i=0;i<500000; ++i) d.insert(d.begin()+d.size()/2,0);
// time: 11.61s             MIDDLE

list<int> l;
for(int i=0;i<200000; ++i) l.insert(l.begin(),0);
// time: 0.15s              FRONT
```

```
vector<int> v;
for(int i=0; i<9000000; ++i) v.push_back(0);
// time: 0.45s

list<int> l;
for(int i=0; i<9000000; ++i) l.insert(l.begin(),0);
// time: 2.52s
```

In generale:

vector

- Memoria contigua.
- Prealloca lo spazio per gli elementi futuri, quindi richiede spazio extra oltre a quello necessario per gli elementi stessi.
- Ogni elemento richiede solo lo spazio per il tipo di elemento stesso (nessun puntatore extra).
- È possibile riallocare la memoria per l'intero vettore ogni volta che si aggiunge un elemento.
- Gli inserimenti alla fine sono costanti, in tempo ammortizzato, ma gli inserimenti altrove sono costosi O(n).
- Le cancellazioni alla fine del vettore sono a tempo costante, ma per il resto è O(n).
- È possibile accedere ai suoi elementi in modo casuale.
- Gli iteratori vengono invalidati se si aggiungono o rimuovono elementi dal vettore.
- È possibile accedere facilmente all'array sottostante se si ha bisogno di un array di elementi.

list

- Memoria non contigua.
- Nessuna memoria preallocata. L'overhead di memoria per l'elenco stesso è costante.
- Ogni elemento richiede spazio extra per il nodo che lo contiene, compresi i puntatori agli elementi successivi e precedenti dell'elenco.
- Non è mai necessario riallocare la memoria per l'intero elenco solo perché si aggiunge un elemento.
- Gli inserimenti e le cancellazioni sono economici, indipendentemente dalla posizione nell'elenco.
- È economico combinare gli elenchi con lo splicing.
- Non è possibile accedere agli elementi in modo casuale, quindi raggiungere un elemento particolare dell'elenco può essere costoso.
- Gli iteratori rimangono validi anche quando si aggiungono o rimuovono elementi dall'elenco.
- Se avete bisogno di un array di elementi, dovrete crearne uno nuovo e aggiungervi tutti gli elementi, poiché non esiste un array sottostante.

In generale, si usa il vector quando non interessa il tipo di contenitore sequenziale che si sta usando, ma se si devono fare molti inserimenti o cancellazioni da e verso qualsiasi punto del contenitore che non sia la fine, è meglio usare list. Oppure, se si ha bisogno di un accesso casuale, è meglio usare vettori, non liste. A parte questo, ci sono naturalmente casi in cui si ha bisogno dell'uno o dell'altro in base all'applicazione, ma in generale queste sono buone linee guida.

Ora, la deque, tipicamente implementato come una strutta di dimensione fissa e come sorta di buffer circolare, data la sua natura. Le riallocazioni sono efficienti rispetto al vector.

Contenitore bidirezionale: `deque<Tipo>`

Si tratta del contenitore "*Double Ended Queue*", coda a due estremi (si legge "deck").

Il contenitore deque unisce alcuni dei vantaggi principali del vector e della list.

1) Accesso indicizzato efficiente per lettura e scrittura come in un vector.
2) Inserimento ed eliminazione agli estremi del contenitore sono efficienti come in una list.

I contenitori associativi permettono di accedere ad un elemento mediante il valore dell'elemento stesso o di una parte di tale valore, cosiddetta *chiave* di accesso all'elemento. Le chiavi sono mantenute ordinate attraverso l'operatore di ordinamento *operator <*.

Altri esempi di contenitori (citati, ma da noi mai usati):

Contenitore insieme: `set<Tipo>`.

Modella il concetto matematico di insieme e quindi accetta solamente una occorrenza per ogni valore nell'insieme. La **chiave è costituita dal valore dell'elemento**. Quindi gli elementi contenuti nel set devono avere valori tutti distinti.

Contenitore multiinsieme: `multiset<Tipo>`.

La chiave è anche in questo caso data dal valore dell'elemento. Un multiset può contenere **occorrenze multiple dello stesso elemento**, ovvero più elementi con lo stesso valore.

Contenitore mappa: `map<T1,T2>`.

Un elemento di un map è costituito da una coppia (chiave, valore), ovvero da una chiave di accesso e da un valore associato a tale chiave. Il tipo delle chiavi `T1` deve quindi supportare l'operatore relazionale d'ordinamento `operator<`.

Gli elementi contenuti nel map devono avere **chiavi tutte distinte**. Tipicamente, in un map `m` si accede al valore associato ad una chiave `k` tramite l'operatore di indicizzazione: `T2 x = m[k];`

Contenitore multimappa: `multimap<T1,T2>`

Come in un map, il valore di un elemento di un multimap è costituito da una chiave e da un valore associato a tale chiave. La differenza è che un multimap può contenere più elementi con la stessa chiave.

Nella stessa libreria STL sono definiti molti algoritmi generici efficienti, quali count(), equal(), search(), find(), reverse(), copy(), ecc.

L'header <algorithm> definisce un insieme di funzioni appositamente studiate per essere utilizzate su intervalli di elementi.

Un intervallo è una qualsiasi sequenza di oggetti a cui si può accedere tramite iteratori o puntatori, come un array o un'istanza di alcuni contenitori STL. Si noti però che gli algoritmi operano tramite iteratori direttamente sui valori, senza influenzare in alcun modo la struttura di un eventuale contenitore (non influenzano mai la dimensione o l'allocazione di memoria del contenitore).

Complessità media → O (nlog(n))

L'esempio che a noi serve è il *for_each*, iterazione efficiente da C++ 11.

```
for_each(InputIterator first, InputIterator last, UnaryFunction f)
```

```
std::for_each (v.begin(), v.end(), print);
```

Anche in Qt abbiamo molti tipi di iteratori, ma non sono standard, si usano solo in Qt.

```
QVector<T>, QList<T>, QString,
QSet<T>, QMap<Key,T>,
QMultimap<Key,T>, QHash<Key,T>, etc.
```

```
/*
ESERCIZIO.

Definire un template di classe dList<T> i cui oggetti rappresentano una struttura dati lista doppiamente
concatenata (doubly linked list) per elementi di uno stesso tipo T. Il template dList<T> deve soddisfare i
seguenti vincoli:

1. Gestione della memoria senza condivisione.

2. dList<T> rende disponibile un costruttore dList(unsigned int k, const T& t) che costruisce una
lista contenente k nodi ed ognuno di questi nodi memorizza una copia di t.

3. dList<T> permette l'inserimento in testa ed in coda ad una lista in tempo O(1) (cioe` costante):
-- Deve essere disponibile un metodo void insertFront(const T&) con il seguente comportamento:
dl.insertFront(t) inserisce l'elemento t in testa a dl in tempo O(1).
-- Deve essere disponibile un metodo void insertBack(const T&) con il seguente comportamento:
dl.insertBack(t) inserisce l'elemento t in coda a dl in tempo O(1).

4. dList<T> rende disponibile un opportuno overloading di operator< che implementa l'ordinamento
lessicografico (ad esempio, si ricorda che per l'ordinamento lessicografico tra stringhe abbiamo che
"campana" < "cavolo" e che "buono" < "ottimo").

5. dList<T> rende disponibile un tipo iteratore costante dList<T>::const_iterator i cui oggetti permettono
di iterare sugli elementi di una lista.
*/

template<class T>
class dList {
private:

  class nodo {
  public:
    T info;
    nodo *prev, *next;
    nodo(const T& t, nodo* p = 0, nodo* n=0): info(t), prev(p), next(n) {}
  };

  nodo *first, *last; // puntatori al primo e ultimo nodo della lista
  // lista vuota IFF first == nullptr == last

  static void destroy(nodo* n) {
    if (n != nullptr) {
      destroy(n->next);
      delete n;
    }
  }

  static void deep_copy(nodo *src, nodo*& fst, nodo*& last) {
    if (src) {
      fst = last = new nodo(src->info);
      nodo* src_sc = src->next;
        while (src_sc) {
          last = new nodo(src_sc->info, last);
          last->prev->next = last;
          src_sc = src_sc->next;
        }
    }
    else {
      fst = last = nullptr;
    }
  }

 static bool isLess(const nodo* l1, const nodo* l2) {
    if(!l1 && !l2) return false;
    // l1 | l2
    if(!l1) return true; // vuota < non vuota
    if(!l2) return false; // non vuota < vuota
    // l1 & l2
    if(l1->info < l2->info) return true;
    else if(l1->info > l2->info) return false;
    else // l1->info == l2->info
      return isLess(l1->next, l2->next);
 }
```

```
public:

  dList(const dList& l) {
    deep_copy(l.first,first,last);
  }

  dList& operator=(const dList& l) {
    if(this != &l) {
      destroy(first);
      deep_copy(l.first,first,last);
    }
    return *this;
  }

  ~dList() { destroy(first); }

  void insertFront(const T& t) {
    first = new nodo(t,nullptr,first);
    if(first->next==nullptr) { // lista di invocazione era vuota
      last=first;
    }
    else { // lista di invocazione NON era vuota
      (first->next)->prev=first;
    }
  }

  void insertBack(const T& t) {
    if(last){ // lista non vuota
      last = new nodo(t,last,nullptr);
      (last->prev)->next=last;
    }
    else // lista vuota
      first=last=new nodo(t);
  }

  dList(unsigned int k, const T& t): first(nullptr), last(nullptr) {
    for(unsigned int j=0; j<k; ++j) insertFront(t);
  }

  bool operator<(const dList& l) const {
    if(this == &l) return false; // optimization: l < l e' sempre false
    return isLess(first, l.first);
  }

  class const_iterator {
    friend class dList <T>;
  private: // const_iterator indefinito IFF ptr==nullptr & past_the_end==false
    const nodo* ptr;
    bool past_the_end;

    // convertitore "privato" nodo* => const_iterator
    const_iterator(nodo* p, bool pte = false): ptr(p), past_the_end(pte) {}
  public:

    const_iterator(): ptr(nullptr), past_the_end(false) {}

    const_iterator& operator++() {
      if(ptr!= nullptr) {
        if(!past_the_end) {
          if(ptr->next != nullptr) {ptr = ptr->next;}
          else { ptr = ptr+1; past_the_end = true; }
        }
      }
      return *this;
    }

    const_iterator operator++(int){
      const_iterator aux(*this);
      if(ptr!= nullptr) {
        if(!past_the_end) {
          if(ptr->next != nullptr) ptr = ptr->next;
          else {ptr = ptr+1; past_the_end = true;}
        }
      }
```

```
      return aux;
    }

    const_iterator& operator--() {
      if(ptr != nullptr) {
        if(ptr->prev == nullptr) ptr=nullptr;
        else if(!past_the_end) ptr = ptr->prev;
        else {ptr = ptr-1; past_the_end = false;}
      }
      return *this;
    }

    const_iterator operator--(int){
      const_iterator aux(*this);
      if(ptr != nullptr) {
        if(ptr->prev == nullptr) ptr=nullptr;
        else if(!past_the_end) ptr = ptr->prev;
        else {ptr = ptr-1; past_the_end = false;}
      }
      return aux;
    }

    bool operator==(const const_iterator& cit) const {
      return ptr == cit.ptr;
    }

    bool operator!=(const const_iterator& cit) const {
      return ptr != cit.ptr;
    }

    const T& operator*() const {
      return ptr->info;
    }

    const T* operator->() const {
      return &(ptr->info);
    }
  };

  const_iterator begin() const {
    return const_iterator(first);
  }

  const_iterator end() const {
    if(!last) return const_iterator();
    return const_iterator(last+1,true); // attenzione: NON e' past the end
  }

};

// esempio d'uso
#include<iostream>

int main() {
  dList<int> x(4,2), y(0,0), z(6,8);
  y=x;
  x.insertFront(-2); z.insertFront(3); y.insertFront(0);
  if(x<y) std::cout << "x < y" << std::endl;
  if(z<x) std::cout << "z < x" << std::endl;
  if(y<z) std::cout << "y < z" << std::endl;
  if(z<y) std::cout << "z < y" << std::endl;

  std::cout << "x= ";
  dList<int>::const_iterator j = --(x.end());
  for(; j != x.begin(); --j) std::cout << *j << ' ';
  std::cout << *j << std::endl << "y= ";
  for(dList<int>::const_iterator k = y.begin(); k != y.end(); ++k) std::cout << *k << ' ';
  std::cout << std::endl << "z= ";
  dList<int>::const_iterator i = z.begin();
  for( ; i != z.end(); ++i) std::cout << *i << ' ';
  std::cout << std::endl;
}
```

Ora, il concetto più importante visto fino ad ora: *l'ereditarietà.*

La capacità di una classe di derivare proprietà e caratteristiche da un'altra classe si chiama Ereditarietà. L'ereditarietà è una caratteristica o un processo che prevede la creazione di nuove classi a partire da quelle esistenti. La nuova classe creata è chiamata "classe derivata" o "classe figlio" e la classe esistente è nota come "classe base" o "classe genitore". La classe derivata si dice ora ereditata dalla classe base.

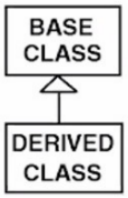
Quando diciamo che la classe derivata eredita la classe base, significa che la classe derivata eredita tutte le proprietà della classe base, senza modificare le proprietà della classe base e può aggiungere nuove caratteristiche alla propria. Queste nuove caratteristiche della classe derivata non influiscono sulla classe base. La classe derivata è la classe specializzata per la classe base.

- Classe secondaria: La classe che eredita proprietà da un'altra classe è chiamata sottoclasse o classe derivata.
- Superclasse: La classe le cui proprietà sono ereditate da una sottoclasse è chiamata classe base o superclasse.

Classe base **B** e classe derivata **D**

Sottoclasse **D** e superclasse **B**

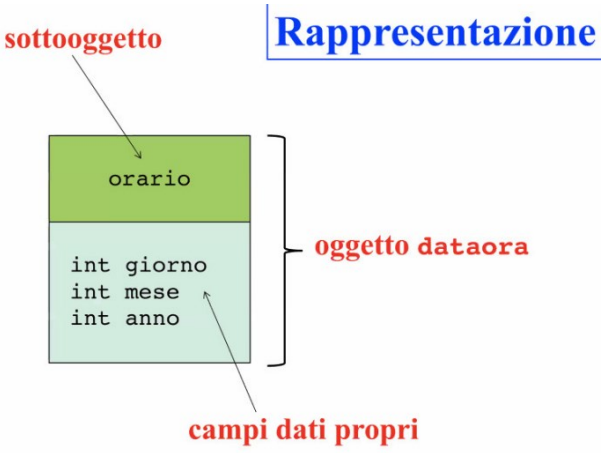
Sottotipo **D** e supertipo **B**



Avviciniamoci agli esempi usati fino ad ora: la classe *orario*. Il tipo *dataora* concettualmente **è** un orario.

```
class dataora : public orario {
private:
  int giorno;
  int mese;
  int anno;
public:
  int Giorno() const;
  int Mese() const;
  int Anno() const;
};
```

e aggiungo membri



Nella teoria dei linguaggi di programmazione, la *subtyping* è una forma di polimorfismo dei tipi in cui un sottotipo è un tipo di dato che è legato a un altro tipo di dato (il supertipo) da una qualche nozione di sostituibilità, il che significa che gli elementi del programma, tipicamente subroutine o funzioni, scritti per operare su elementi del supertipo possono operare anche su elementi del sottotipo.
È la caratteristica fondamentale dell'ereditarietà. Infatti, ogni oggetto della classe derivata è utilizzabile anche come oggetto della classe base.

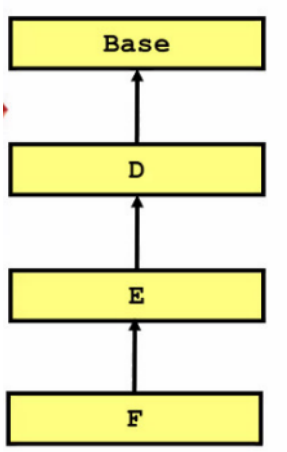
- **Subtyping**: Sottotipo **D** ⇒ Supertipo **B**
- Per oggetti: **D** ⇒ **B** **estrae il sottooggetto**

Concetto decisamente fondamentale: a seconda del contesto di invocazione, il puntatore si riferisce al supertipo o al sottotipo, quindi "cambiando forma". Intuitivamente, ciò significa polimorfo.

Subtyping per puntatori e riferimenti

D* ⇒ B* **D& ⇒ B&**

Puntatori e riferimenti polimorfi



Il sottotipo può essere:

- Diretto, se deriva da una classe direttamente (es. D)
- Indiretto, se deriva transitivamente da una classe (es. E)

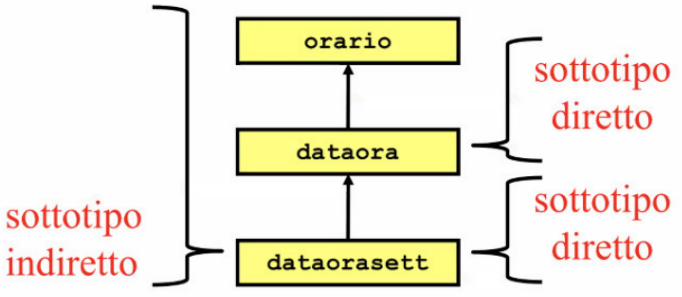
Casi d'uso dell'ereditarietà: (con <: che indica "eredita da"):

1) Per estensione → dataora <: orario
2) Specializzazione → (caso Qt) QButton <: QComponent
3) Ridefinizione → Queue <: List
4) Riutilizzo di codice (non è subtyping)

Nel caso di *dataora* si memorizza il giorno della settimana con *enum*, tipo enumerazione anche detto. Un'enumerazione è un tipo di dati definito dall'utente e composto da costanti integrali. Per definire un'enumerazione si usa la parola chiave enum.
Nell'esempio che segue, ogni valore (lun, mar, ecc.), corrisponde ad un numero (1, 2, ecc.)

```
// tipo enumerazione giorno
enum giorno {lun,mar,mer,gio,ven,sab,dom};

class dataorasett : public dataora {
public:
  giorno GiornoSettimana() const;
private:
  giorno giornosettimana;
};
```

Una semplice gerarchia di tre classi.



Data una classe B, per ogni sottotipo D (in generale indiretto) di B, valgono come conversioni implicite:

D ⇒ B (oggetti)

D& ⇒ B& (riferimenti)

D* ⇒ B* (puntatori)

Grazie alla conversione implicita

dataora ⇒ orario

possiamo scrivere:

```
int F(orario o) {...}
dataora d;
int i = F(d);
```

Il viceversa non vale!

```
int G(dataora d) {...}
orario o;
int i = G(o); // ILLEGALE
```

Un **dataora** è (in particolare) un **orario**, mentre un **orario** non è un **dataora**!

Data D una sottoclasse di B (**fondamentale**):

Sia **D** una sottoclasse di **B**.

conversione **D*** ⇒ **B***

```
D d; B b;
D* pd=&d;
B* pb=&b;
pb=pd;
```

tipo statico del puntatore **pb** versus *tipo dinamico* di **pb**

Quindi: il **tipo statico** di un puntatore **p** è il tipo **T*** di dichiarazione di **p**, mentre se in un certo istante dell'esecuzione il tipo dell'oggetto a cui effettivamente punta **p** è **U** allora in quell'istante **U*** è il **tipo dinamico** di **p**.

Mentre il tipo statico è fissato al momento della dichiarazione, il tipo dinamico in generale può variare a run-time.

Un esempio diretto:

Concetti e terminologia analoghi valgono per i riferimenti:

conversione D& ⟹ B&

```
D d; B b;
D& rd=d;
B& rb=d;

// D& è il tipo dinamico di rb
```

Il controllo statico dei tipi significa che il controllo dei tipi avviene in fase di compilazione. In questo caso, non vengono utilizzate informazioni sui tipi in fase di esecuzione.
Il controllo dinamico dei tipi avviene quando le informazioni sui tipi vengono utilizzate in fase di esecuzione. Vedremo successivamente queste implicazioni (oggetto di tutti gli esercizi ed esami).

La parte privata della classe base è inaccessibile alla parte privata.

Aggiungendo alla classe **dataora** un metodo **Set2K()** che assegna all'oggetto di invocazione le ore 00:00:00 del 1 gennaio 2000 **non possiamo scrivere**

```
dataora::Set2K() {
  sec = 0; // NO, illegale!
  giorno = 1;
  mese = 1;
  anno = 2000;
}
```

Ereditarietà di tipo **Ereditarietà di implementazione**

Classe base	*Classe derivata con derivazione*		
membro	**pubblica**	**protetta**	**privata**
privato	~~inaccess.~~	inaccess.	inaccess.
protetto	~~protetto~~	protetto	privato
pubblico	~~pubblico~~	protetto	privato

Attenzione: Derivazioni protette e private **non supportano** l'ereditarietà di tipo

Esistono vari tipi di ereditarietà, che seguono i modificatori di accesso, quindi:

- privata (non visibili da nessuno)
- pubblica (visibili da tutti)
- protetta (I membri della classe dichiarati come protetti sono inaccessibili al di fuori della classe, ma possono essere accessibili a qualsiasi sottoclasse (classe derivata) di quella classe).

I membri protected rappresentano una violazione all'information hiding, come si vede da qui:

```
class C {
private:
  int priv;
protected:
  int prot;
public:
  int publ;
};

class D: private C {
  // prot e publ divengono privati
};

class E: protected C {
  // prot e publ divengono protetti
};

class F: public D {
  // prot e publ sono qui inaccessibili
public:
  void fF(int i, int j){
    // prot=i; // Illegale
    // publ=j; // Illegale
  }
};
class G: public E {
  // prot e publ rimangono qui protetti
  void fG(int i, int j){
    prot=i; // OK
    publ=j; // OK
  }
};
```

Il concetto di ereditarietà privata è un po' particolare, quando non si è interessati ad una relazione di subtyping e quindi non usare le funzionalità della sottoclasse.

L'ereditarietà privata è uno dei modi per implementare la relazione has-a.

Con l'ereditarietà privata, i membri pubblici e protetti della classe base diventano membri privati della classe derivata. Ciò significa che i metodi della classe base non diventano l'interfaccia pubblica dell'oggetto derivato. Tuttavia, possono essere utilizzati all'interno delle funzioni membro della classe derivata.

Il confronto tra ereditarietà privata e relazione has-a segue:

- has-a usa i metodi direttamente
- Ereditarietà privata significa "usare alla bisogna"

Relazione has-a

```
class Motore {
private:
  int numCilindri;
public:
  Motore(int nc): numCilindri(nc) {}
  int getCilindri() const {return numCilindri;}
  void accendi() const {
    cout << "Motore a " << getCilindri() << " cilindri acceso" << endl;}
};

Class Auto {
private:
  Motore mot; // Auto has-a Motore come campo dati
public:
  Auto(int nc = 4): mot(4) {}
  void accendi() const {
    mot.accendi();
    cout << "Auto con motore a " << mot.getCilindri() << " cilindri accesa" << endl;
  }
};
```

Ereditarietà privata

```
class Motore {
private:
  int numCilindri;
public:
  Motore(int nc): numCilindri(nc) {}
  int getCilindri() const {return numCilindri;}
  void accendi() const {
    cout << "Motore a " << getCilindri() << " cilindri acceso" << endl;}
};

Class Auto: private Motore { // Auto has-a Motore come sottoogetto
public:
  Auto(int nc = 4): Motore(nc) {}
  void accendi() const {
    Motore::accendi();
    cout << "Auto con motore a " << getCilindri() << " cilindri accesa" << endl;
  }
};
```

Ereditarietà privata vs relazione di composizione has-a

Similarità

1) In entrambi i casi un oggetto Motore "contenuto" in ogni oggetto Auto

2) In entrambi i casi, per gli utenti esterni, Auto* non è convertibile a Motore*

Differenze

1) La composizione è necessaria se servono **più motori** in un auto (a meno di usi limite di ereditarietà multipla)

2) Ered.privata può introdurre **ereditarietà multipla** (problematica) **non necessaria**

3) Ered.privata **permette ad Auto** di convertire Auto* a Motore*

4) Ered.privata permette **l'accesso alla parte protetta** della base

Le conversioni implicite indotte dalla derivazione valgono solamente per la derivazione pubblica che è l'unica tipologia che supporta la relazione "is-a". La derivazione protetta e privata *non inducono* alcuna conversione implicita.

Le conversioni implicite indotte dalla derivazione **valgono solamente per la derivazione pubblica** che è quindi l'unica tipologia di derivazione che supporta la relazione "is-a". La derivazione protetta e la derivazione privata **non inducono** alcuna conversione implicita.

```
class C {
private:
  int i;
protected:
  char c;
public:
  float f;
};

class D: private C { };   // derivazione privata
class E: protected C { }; // derivazione pubblica
// nessuna conversione implicita D => C e E => C
int main() {
  C c, *pc; D d, *pd; E e, *pe;
//c=d;      // Illegale: "C is an inaccessible base of D"
//c=e;      // Illegale
//pc=&d;    // Illegale
//pc=&e;    // Illegale
//C& rc=d; // Illegale
}
```

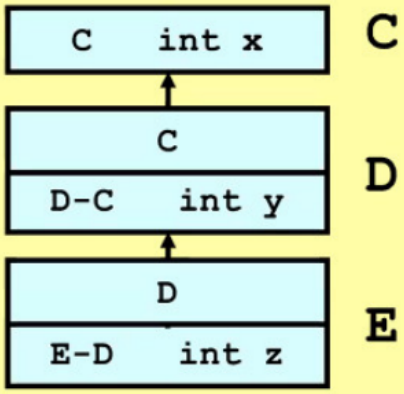
Il ragionamento possibile si articola in due frammenti di codice.
In questo primo, si vede l'oggetto *p* che da oggetto classe Base chiama un metodo della derivata ridefinito; fa un po' un misto del pattern has- a e l'ereditarietà privata e non ha molto senso.

```
class Base {
  int x;
public:
  void f() {x=2;}
};

class Derivata: public Base {
  int y;
public:
  void g() {y=3;}
};

int main() {
  Base b; Derivata d;
  Base* p = &b; Derivata* q = &d;
  p->f(); // OK
  p=&d;   // Derivata* è ora il tipo dinamico di p
  p->f(); // OK
  p->g(); // HA SENSO?
  q->g(); // OK
  q->f(); // OK
}
```

Ben più utile analizzare:

Conversioni Base*⇒Derivata*, Base&⇒Derivata&

```
// VI significa Valore (intero) Imprevedibile
class C {
public:
  int x;
  void f() {x=4;}
};

class D: public C {
public:
  int y;
  void g() {x=5; y=6;}
};

class E: public D {
public:
  int z;
  void h() {x=7; y=8; z=9;}
};

int main() {
  C c; D d; E e;
  c.f(); d.g(); e.h();
  D* pd = static_cast<D*> (&c); // PERICOLOSO!
  cout << pd->x <<" "<< pd->y << endl; // errore run-time o stampa: 4 VI
  E& re = static_cast<E&> (d);  // PERICOLOSO!
  cout << re.x <<" "<< re.y <<" "<< re.z << endl; // err. run-time o stampa: 5 6 VI
  C* pc = &d; pd = static_cast<D*> (pc); // OK
  cout << pd->x <<" "<< pd->y << endl; // stampa: 5 6
  D& rd = e; E& re1 = static_cast<E&> (rd); // OK
  cout << re1.x <<" "<< re1.y <<" "<< re1.z << endl; // stampa: 7 8 9
}
```



Attenzione: le amicizie non vengono ereditate. Quindi, non valgono le relazioni friend sui sottotipi:

```
class C {
private:
  int i;
public:
  C(): i(1) {}
  friend void print(C);
};

class D: public C {
private:
  double z;
public:
  D(): z(3.14) {}
};

void print(C x) {
  cout << x.i << endl;
  D d;
//cout << d.z << endl; // Illegale:
//"D::z is private within this context"
}

int main() {
  C c; D d;
  print(c); // stampa: 1
  print(d); // OK, stampa: 1
}
```

```
class C {
  friend class Z;
private:
  int i;
public:
  C(): i(1) {}
};

class D: public C {
private:
  double z;
public:
  D(): z(3.14) {}
};

class Z {
public:
  void m() { C c; D d; cout << c.i; // OK
  cout << d.z; // Illegale: "D::z is private within this context"
  }
};

int main() {
  Z z;
  z.m(); // stampa: 1
}
```

Alcuni oggetti possono essere inaccessibili a seconda del contesto;
l'esempio successivo lo riporta. Usiamo *this* per accedere ad un membro privato, tra le altre cose attraverso un sottooggetto (quindi con D); l'oggetto *i* rimane comunque inaccessibile, per le considerazioni di sopra-

```
class C {
private:
  int i;
public:
  C(): i(1) {}
  void print() {cout << ' ' << i;}
};

class D: public C {
private:
  double z;
public:
  D(): z(3.14) {}
  void print() {
    C::print(); // l'oggetto di invocazione di C::print() è il
                // sottooggetto di tipo C dell'oggetto di invocazione
  //cout << ' ' << this->i; // membro i INACCESSIBILE
    cout << ' ' << z;}
};

int main() {
  C c; D d;
  c.print(); cout << endl;    // stampa: 1
  d.print(); cout << endl;    // stampa: 1 3.14
}
```

Torniamo sul significato di protected, tale che noi consideriamo un metodo oppure una variabile che viene vista solo dalle sottoclassi. È ciò che accade con *i* e con un metodo protetto di stampa.

```
class B {
protected:
  int i;
  void protected_printB() const {cout << ' ' << i;}
public:
  void printB() const {cout << ' ' << i;}
};

class D: public B {
private:
  double z;
public:
  void stampa() {
    cout << i << ' ' << z; // OK
  }

  static void stampa(const B& b,const D& d) {
    cout << ' ' << b.i; // Illegale: "B::i is protected within this context"
    b.printB(); // OK
    b.protected_printB(); // Illegale: "B::protected_printB() is
                          //            protected within this context"
    cout << ' ' << d.i; // OK
    d.printB(); // OK
    d.protected_printB(); // OK
  }
};
```

Il principale scopo con cui usiamo l'ereditarietà è la possibile *ridefinizione di metodi*, i quali possono richiedere variazioni o adattamenti nella classe derivata.
Ad esempio, la ridefinizione dell'operatore di somma per gli orari:

Ridefinizione di orario::operator+

```
dataora dataora::operator+(const orario& o) const {
  dataora aux = *this;
// ATTENZIONE:
// aux.sec = sec + o.sec; darebbe un errore di compilazione!
// perchè anche se sec è dichiarato protected in orario,
// o.sec è comunque inaccessibile in dataora
  aux.sec = sec + 3600*o.Ore() + 60*o.Minuti() + o.Secondi();
  if (aux.sec >= 86400) {
    aux.sec = aux.sec - 86400;
    aux.AvanzaUnGiorno();
  }
  return aux;
}
```

```
void dataora::AvanzaUnGiorno() { // metodo proprio
  if (giorno < GiorniDelMese()) giorno++;
  else if (mese < 12) { giorno = 1; mese++; }
  else { giorno = 1; mese = 1; anno++;}
}
```

Invochiamo l'operatore di somma di *orario* oppure *dataora* come segue:

```
orario o1, o2;
dataora d1, d2;
o1 + o2;     // invoca orario::operator+
d1 + d2;     // invoca dataora::operator+
o1 + d2;     // invoca orario::operator+
d1 + o2;     // invoca dataora::operator+
orario y = d1 + d2;  // OK
dataora x = o1 + o2; // Illegale
d1.orario::operator+(d2); // invoca orario::operator+
```

Attenzione: chiamare la somma con *dataora* è illegale per effetto della cosiddetta name hiding rule; quindi, una ridefinizione del metodo *m()* nasconde sempre tutte le versioni sovraccaricate di *m()* disponibili in *B*, che non sono quindi direttamente accessibili in *D* ma solamente tramite l'operatore di scoping *B::*

Se ridefiniamo il metodo `Ore` in **`dataora`** con segnatura:

```
int dataora::Ore(int) const {
  ...
}
```

non possiamo più scrivere:

```
dataora d;
cout << d.Ore(); // Illegale
```

perchè il "vecchio" metodo `Ore` della classe **`orario`** è mascherato in **`dataora`** dalla ridefinizione. Per l'accesso possiamo però usare l'operatore di scoping:

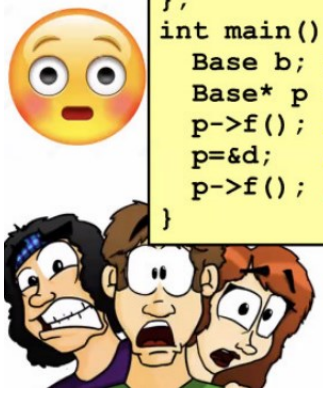
```
dataora d;
cout << d.orario::Ore();
```

Con la stessa logica, si assiste alla *ridefinizione dei campi dati*, come nell'esempio che segue, in cui per ereditarietà recupero un campo ridefinendolo nella derivata ed operando modifiche su di esso:

```
class B {
protected:
  int x;
public:
  B() : x(2) {}
  void print() {cout << x << endl;}
};

class D: public B {
private:
  double x; // ridefinizione del campo dati x
public:
  D() : x(3.14) {}
  // ridefinizione di print()
  void print() {cout << x << endl;} // è D::x
  void printAll() {cout << B::x << ' ' << x << endl;}
};

main () {
  B b; D d;
  b.print();    // stampa: 2
  d.print();    // stampa: 3.14
  d.printAll(); // stampa: 2 3.14
}
```

Un altro concetto fondamentale della programmazione ad oggetti: lo *static binding*.
Il binding statico avviene quando tutte le informazioni necessarie per chiamare una funzione sono disponibili al momento della compilazione e può essere ottenuto utilizzando le normali chiamate di funzione, l'overloading delle funzioni e l'overloading degli operatori.
Cosa si intende con questo?
Si intende che il puntatore, qualora il suo comportamento non venga ridefinito da qualche sottotipo, utilizza ciò "che già conosce" a runtime ed utilizza i metodi della propria classe piuttosto che usare quelli *dinamici*, cioè quelli delle sottoclassi.

```
class Base {
  int x;
public:
  void f() {x=2;}
};
class Derivata: public Base {
  int y;
public:
  void f() { Base::f(); y=3; } // ridefinizione
};
int main() {
  Base b; Derivata d;
  Base* p = &b;
  p->f(); // invoca Base::f()
  p=&d;   // Derivata* è il tipo dinamico di p
  p->f(); // cosa invoca??
}
```



Base::f()

Segue qualche esempio.
Nel primo che vediamo, si hanno tre chiamate illegali, in quanto semplicemente ogni classe ridefinisce il metodo cambiando tipo di ritorno e/o argomenti e per D dovrebbe tornare un intero (viene visto come void nella chiamata) e negli altri due casi è l'opposto (viene visto come una funzione in grado di ritornare un valore ma non ritorna nulla oppure non passa il giusto parametro, caso di *h*).

```
class B {
public:
  int f() const { cout << "B::f()\n"; return 1; }
  int f(string) const { cout << "B::f(string)\n"; return 2; }
};

class D : public B {
public:
  // ridefinizione con la stessa segnatura
  int f() const { cout << "D::f()\n"; return 3; }
};

class E : public B {
public:
  // ridefinizione con cambio del tipo di ritorno
  void f() const { cout << "E::f()\n"; }
};

class H : public B {
public:
  // ridefinizione con cambio lista argomenti
  int f(int) const { cout << "H::f()\n"; return 4; }
};

int main() {
  string s; B b; D d; E e; H h;
  int x = d.f(); // stampa: D::f()
//d.f(s);        // Illegale
//x = e.f();     // Illegale
//x = h.f();     // Illegale
  x = h.f(1);    // stampa: H::f()
}
```

Ulteriore serie di esempi:

```
class C {
public:
  void f(int x) { }
  void f() { }
};
class D: public C {
  int y;
public:
 void f(int x) {f(); y=3+x;}
 // Illegale:
 // "no matching function for D::f()"
};
```

```
class C {
public:
  int x;
  void f() {x=1;}
};

class D: public C {
public:
  int y;
  void f() {C::f(); y=2;} // OK
};

int main() {
  C c; D d; c.f(); d.f();
  cout << c.x << endl; // stampa: 1
  cout << d.x << " " << d.y;
  // stampa: 1 2
}
```

```
class C {
public:
  int x;
  void f() {x=1;}
};

class D: public C {
public:
  int y;
  void f() {std::cout <<"*";
            y=3; f();}
            // errore logico:
            // ricorsione infinita!
};

int main() {
  D d; d.f(); // compila ma...
} // errore run-time:
  // è uno stack overflow!
```

Scritto da Gabriel

```
class B {
public:
void f() {}
};

class D1: public B {
public:
void f() {}
};

class D2: public D1 {
public:
void g() {
D1::f(); // D1::f()
B::f(); // B::f()
}

void f() {
f(); // chiamata ricorsiva D2::f(
D1::f();
B::f();
}
};
```

Domanda: " se io ho più livelli di ereditarietà (B, D1, D2) e voglio usare un metodo f() di B, ridefinto sia in D1 che in D2, da D2 la chiamata sarà d2.B::f() ? "

```
main.cpp: In function ‘void f()’:
main.cpp:25:11: error: cannot call member function ‘void D1::f()’ without object
   25 |     D1::f();
      |           ^
main.cpp:26:10: error: cannot call member function ‘void B::f()’ without object
   26 |     B::f();
      |          ^
```

Un tipico esercizio in cui si verifica se si ha una giusta ridefinizione di puntatori e metodi, commentato dal prof:

```
class C {
public:
  void f() {cout << "C::f\n";}
};

class D: public C {
public:
  void f() {cout << "D::f\n";}; // ridefinizione
};

class E: public D {
public:
  void f() {cout << "E::f\n";}; // ridefinizione
};

int main() {
  C c; D d; E e;
  C* pc = &c; E* pe = &e;
  c = d;     // OK: conversione D => C
  c = e;     // OK: conversione E => C
  d = e;     // OK: conversione E => D
  C& rc=d;   // OK: conversione D => C
  D& rd=e;   // OK: conversione E => D
  pc->f();   // OK
  pc = pe;   // OK: conversione E* => C*
  rd.f();    // OK
  c.f();     // OK
  pc->f();   // OK
}
```

Altra cosa fondamentale: *costruttori, distruttori ed assegnazioni nelle classi derivate*.
Per i primi:

Costruttori nelle classi derivate

La lista di inizializzazione di un costruttore di una classe D derivata direttamente da B in generale può contenere invocazioni di costruttori per i campi dati (propri) di D e l'invocazione di un costruttore della classe base B.

L'esecuzione di un tale costruttore di D avviene nel seguente modo:

[1] viene sempre e comunque invocato per primo un costruttore della classe base B, o esplicitamente o implicitamente il costruttore di default di B quando la lista di inizializzazione non include una invocazione esplicita;

[2] successivamente, secondo il comportamento già noto, viene eseguito il costruttore "proprio" di D, ossia vengono costruiti i campi dati propri di D;

[3] infine viene eseguito il corpo del costruttore.

In particolare, se nella classe derivata D si omette qualsiasi costruttore allora, come al solito, è disponibile il **costruttore di default standard** di D. Il suo comportamento è quindi il seguente:

[1] richiama il costruttore di default di B;

[2] successivamente si comporta come il costruttore di default standard "proprio" di D, ossia richiama i costruttori di default per tutti i campi dati di D

È dunque naturale usare il costruttori a più parametri di *orario* di *dataora*:

```
dataora::dataora(int a, int me, int g, int o, int m, int s)
            : orario(o,m,s), giorno(g), mese(me), anno(a) {}
```

```
dataora d(2020,11,17,11,55,13);
cout << d.Ore();     // stampa: 11
cout << d.Giorno(); // stampa: 17
```

Altri due esempi utili di cosa stampa:

```
class Z {
public:
  Z() {cout << "Z0 ";}
};

class C {
private:
  int x;
public:
  C(int z=1): x(z) {cout << "C01 ";}
};

class D: public C {
private:
  int y;
  Z z;
};

int main() {
  D d; // costruttore standard
}
// stampa: C01 Z0
```

```
class Z {
public:
  Z() {cout << "Z0 ";}
  Z(double d) {cout << "Z1 ";}
};

class C {
private:
  int x;
  Z w;
public:
  C(): w(6.28), x(8) {cout << x << " C0 ";}
  C(int z): x(z) {cout << x << " C1 ";}
};

class D: public C {
private:
  int y;
  Z z;
public:
  D(): y(0) {cout << "D0 ";}
  D(int a): y(a), z(3.14), C(a) {cout << "D1 ";}
};

int main() {
  D d; cout << "UNO\n";
  D e(4); cout << "DUE";
}
// stampa:
// Z1 8 C0 Z0 D0 UNO
// Z0 4 C1 Z1 D1 DUE
```

```
class Z {
public:
  Z() {cout << "Z0 ";}
  Z(const Z& x) {cout << "Zc ";}
};

class C {
private:
  Z w;
public:
  C() {cout << "C0 ";}
  C(const C& x): w(x.w) {cout << "Cc ";}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";}
  // costruttore di copia standard
};

int main() {
  D d; cout << "UNO\n";
  D e = d; cout << "DUE"; // costruttore di copia standard
}
// stampa:
// Z0 C0 Z0 D0 UNO
// Zc Cc Zc DUE
```

```
class Z {
public:
  Z() {cout << "Z0 ";};
  Z(const Z& x) {cout << "Zc ";}
};

class C {
private:
  Z w;
public:
  C() {cout << "C0 ";}
  C(const C& x): w(x.w) {cout << "Cc ";}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";};
  D(const D& x) {cout << "Dc ";} // ridefinizione costr.copia
};

int main() {
  D d; cout << "UNO\n";
  D e = d; cout << "DUE"; // costruttore di copia ridefinito
}
// stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 Dc DUE       // ATTENZIONE!
```

Vediamo l'*assegnazione standard nelle classi derivate*. Funziona esattamente come ogni altro caso; chiaramente, il tipo deve corrispondere e ritorna sempre *this.


```
class Z {
public:
  Z() {cout << "Z0 ";}
  Z(const Z& x) {cout << "Zc ";}
  Z& operator=(const Z& x) {cout << "Z= "; return *this;}
};

class C {
protected:
  Z w;
public:
  C() {cout << "C0 ";}
  C(const C& x): w(x.w) {cout << "Cc ";}
  C& operator=(const C& x) {w=x.w; cout << "C= "; return *this;}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";}
  D(const D& x) {cout << "Dc ";}
};
int main() {
  D d; cout << "UNO\n";
  D e; cout << "DUE\n";
  e=d; cout << "TRE"; // assegnazione standard
}
// stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 D0 DUE
// Z= C= Z= TRE
```

EXAMPLE

```
class Z {
public:
  int x;
  Z(): x(0) {cout << "Z0 ";}
  Z(const Z& x) {cout << "Zc ";}
  Z& operator=(const Z& x) {cout << "Z= "; return *this;}
};
class C {
public:
  Z w;
  C() {cout << "C0 ";}
  C(const C& x): w(x.w) {cout << "Cc ";}
  C& operator=(const C& x) {w=x.w; cout << "C= "; return *this;}
};
class D: public C {
public:
  Z z;
  D() {cout << "D0 ";}
  D(const D& x) {cout << "Dc ";}
  D& operator=(const D& x) {z=x.z; cout << "D= "; return *this;}
  // assegnazione definita male: chi ci pensa ad assegnare il
  // campo dati w di C? e se w fosse private?
};
int main() {
  D d; d.w.x = 3; cout << "UNO\n";
  D e; e.w.x = 5; cout << "DUE\n";
  e=d; cout << "TRE\n";
  cout << e.w.x << ' ' << d.w.x << " QUATTRO";
}
// stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 D0 DUE
// Z= D= TRE
// 5 3 QUATTRO
```

B& b= *this;
b=X;
proposta contorta ma funziona, dal tracciamento evidenzia che chiama prima il costruttore della base

assegnazione ben definita
x=w.x; z=x.z;

EXAMPLE

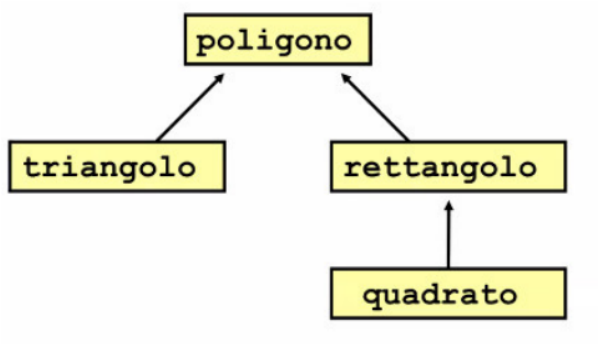
esempio importante per i compiti

```
class B {
private:
  int x;
public:
  B(int k=1): x(k) {}
  B& operator=(const B& a) {x=a.x;}
  void print() const {cout << "x="<<x;}
};
class D: public B {
private:
  int z;
public:
  D(int k=2): B(k), z(k) {}
  // assegnazione con comportamento standard
  D& operator=(const D& x) {
    this->B::operator=(x); // assegnazione per sottooggetto
    z = x.z;
  }
  void print() const {B::print(); cout <<" z="<< z;}
};

int main() {
  D d1(4), d2(5);
  d1.print(); cout << endl; // x=4 z=4
  d2.print(); cout << endl; // x=5 z=5
  d1=d2;
  d1.print(); // x=5 z=5
}
```

```
class Z {
public:
  Z() {cout << "Z0 ";}
  ~Z() {cout <<"~Z ";}
};

class C {
private:
  Z w;
public:
  C() {cout << "C0 ";}
  ~C() {cout << "~C ";}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";}
};

int main() {
  D* p = new D; cout << "UNO\n";
  delete p; cout << "DUE"; // distruttore standard
}
// Z0 C0 Z0 D0 UNO
// ~Z ~C ~Z DUE
```

```
class Z {
public:
  Z() {cout << "Z0 ";}
  ~Z() {cout <<"~Z ";}
};

class C {
private:
  Z w;
public:
  C() {cout << "C0 ";}
  ~C() {cout << "~C ";}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";}
  ~D() {cout <<"~D ";}
};

int main() {
  D* p = new D; cout << "UNO\n";
  delete p; cout << "DUE"; // distruttore ridefinito
}
// stampa:
// Z0 C0 Z0 D0 UNO
// ~D ~Z ~C ~Z DUE
```

Un esempio di derivazione: la classe *Poligono* (completato con alcune osservazione e riporto l'intero codice).




```
//main.cpp

#include "Quadrato.h"
#include "Pol.h"
#include<iostream>
int main(){
Punto v[2]={Punto(5,1),Punto(0,1)};
Quadrato p(v);
Rettangolo l(p);
std::cerr<<p.area()<<std::endl;
std::cerr<<p.perimetro()<<std::endl;


return 0;
}

//pol.h

/*
 * Pol.h
 *
 *  Created on: Nov 16, 2021
 *      Author: marco
 */

#ifndef POL_H_
#define POL_H_
class Punto{
public:
        double x,y;
public:
        Punto(double=0,double=0);
        static double Lung(const Punto&,const Punto&);
};
class Poligono{
public:
        unsigned int nvertici;
        Punto* p;
        static Punto* copia( const Poligono&);
public:
```

```
        Poligono(unsigned int, const Punto* );
        virtual  ~Poligono();
        Poligono(const Poligono&);
        Poligono& operator=(const Poligono&);
        virtual double perimetro() const;
        virtual double area() const =0 ;
};

#endif /* POL_H_ */

//pol.cpp

/*
 * Pol.cpp
 *
 *  Created on: Nov 16, 2021
 *      Author: marco
 */
#include "Pol.h"
#include <math.h>
Punto::Punto(double a,double b): x(a),y(b){}
double Punto::Lung(const Punto& a,const Punto& b){
        return sqrt((b.x-a.x)*(b.x-a.x)+(b.y-a.y)*(b.y-a.y));
}

Poligono::Poligono(unsigned int v, const Punto* b): nvertici(v), p(v>0? new Punto[v]: nullptr){
for(unsigned int i=0;i<v;i++)
        p[i]=b[i];
}
Poligono::~Poligono(){
        delete [] p;
}

Poligono::Poligono(const Poligono& l): nvertici(l.nvertici), p(l.nvertici>0 ? new Punto[l.nvertici]: nullptr){
for (unsigned int i=0;i<l.nvertici;i++){
        p[i]=l.p[i];
}
}
Poligono& Poligono::operator=(const Poligono& q){
if(this!=&q){
        delete p;
        nvertici=q.nvertici;
        p= (nvertici==0 ? nullptr: new Punto[nvertici]);
        for(unsigned int i=0;i<nvertici;i++)
                p[i]=q.p[i];

}
return *this;
}
double Poligono::perimetro() const{
        double s;
        for (unsigned int i=0;i<nvertici; i++){
                s=Punto::Lung(p[i],p[(i+1)%nvertici]);
```

```
        }
        return s;
}

//quadrato.h

/*
 * Quadrato.h
 *
 *  Created on: Nov 16, 2021
 *      Author: marco
 */

#ifndef QUADRATO_H_
#define QUADRATO_H_
#include "Ret.h"
class Quadrato: public Rettangolo{
public:
        Quadrato(const Punto*);
         double perimetro() const;
          double area() const;
};


#endif /* QUADRATO_H_ */

//quadrato.cpp

/*
 * Quadrato.cpp
 *
 *  Created on: Nov 16, 2021
 *      Author: marco
 */
#include"Quadrato.h"
double Quadrato::perimetro()const{
        return Punto::Lung(p[0],p[1])*4;
}
double Quadrato::area() const{
        return Punto::Lung(p[0],p[1])*Punto::Lung(p[0],p[1]);
}
Quadrato::Quadrato(const Punto v[]):Rettangolo(v){}

//ret.h

/*
 * Ret.h
 *
 *  Created on: Nov 16, 2021
 *      Author: marco
 */

#ifndef RET_H_
```

```
#define RET_H_
#include"Pol.h"
class Rettangolo: public Poligono{
public:
        Rettangolo(const Punto*);
         double perimetro() const;
         double area() const;
};


#endif /* RET_H_ */

//Rettangolo (Ret).cpp

/*
 * Rettangolo.cpp
 *
 *  Created on: Nov 16, 2021
 *      Author: marco
 */
#include "Ret.h"

Rettangolo::Rettangolo(const Punto* v): Poligono(4,v){}

double Rettangolo::perimetro() const{
        if(!p) return 0;
        return (Punto::Lung(p[0],p[1])+Punto::Lung(p[1],p[2]))*2;
}
double Rettangolo::area() const{
        return (Punto::Lung(p[0], p[1])*Punto::Lung(p[1], p[2]));
}

//Triangolo.h

/*
 * Triangolo.h
 *
 *  Created on: Nov 16, 2021
 *      Author: marco
 */
#include "Pol.h"
#ifndef TRIANGOLO_H_
#define TRIANGOLO_H_
class Triangolo: public Poligono{
public:
 Triangolo(const Punto*);
 double area() const;
};
class Tri_Rettangolo: public Triangolo {
public:
        Tri_Rettangolo(const Punto*);
        double area() const;
};
```

```
class Tri_Equi: public Triangolo {
public:
	Tri_Equi(const Punto*);
	double area() const;
	double perimetro()const;
};

#endif /* TRIANGOLO_H_ */

//Triangolo.cpp

/*
 * Triangolo.cpp
 *
 *  Created on: Nov 16, 2021
 *      Author: marco
 */
#include "Triangolo.h"
#include <math.h>
Triangolo::Triangolo(const Punto* p): Poligono(3,p){}

double Triangolo::area() const{
	double per=perimetro()/2;
	double a=Punto::Lung(p[0],p[1]), b=Punto::Lung(p[1],p[2]),  c=Punto::Lung(p[2],p[0]);
	return sqrt(per*(per-a)*(per-b)*(per-c));
}
Tri_Rettangolo::Tri_Rettangolo(const Punto* p): Triangolo(p){}
double Tri_Rettangolo::area() const{
return (Punto::Lung(p[0],p[1]) *Punto::Lung(p[1],p[2]))/2;
}
Tri_Equi::Tri_Equi(const Punto* v): Triangolo(v){}
double Tri_Equi::perimetro()const{
	return (Punto::Lung(p[0],p[1]))*3;
}
double Tri_Equi::area() const{
	return ((Punto::Lung(p[0],p[1])/2)*sqrt(3) *Punto::Lung(p[0],p[1]))/2;
}
```

Altro esempio evergreen: ContoBancario.

ESERCIZIO. Definire una superclasse ContoBancario e due sue sottoclassi ContoCorrente e ContoDiRisparmio che soddisfano le seguenti specifiche:

1. Ogni ContoBancario è caratterizzato da un saldo e rende disponibili due funzionalità di deposito e prelievo: double deposita(double) e double preleva(double) che ritornano il saldo aggiornato dopo l'operazione di deposito/prelievo.
2. Ogni ContoCorrente è caratterizzato anche da una spesa fissa uguale per ogni ContoCorrente che deve essere detratta dal saldo ad ogni operazione di deposito e prelievo.
3. Ogni ContoDiRisparmio deve avere un saldo non negativo e pertanto non tutti i prelievi sono permessi; d'altra parte, le operazioni di deposito e prelievo non comportano costi aggiuntivi e restituiscono il saldo aggiornato.
4. Si definisca inoltre una classe ContoArancio derivata da ContoDiRisparmio. La classe ContoArancio deve avere un ContoCorrente di appoggio: quando si deposita una somma S su un ContoArancio, S viene prelevata dal ContoCorrente di appoggio; d'altra parte, i prelievi di una somma S da un ContoArancio vengono depositati nel ContoCorrente di appoggio.

CONTO RANCIO

```
class ContoBancario {
private:
  double saldo;
public:
  double deposita(double x) {
    return x>=0 ? saldo += x : saldo;
  }

  double preleva(double x){
    return x>=0 ? saldo -= x : saldo;
  }

  double getSaldo() const {return saldo;}

  ContoBancario(double s=0.0): saldo(s>=0? s : 0) {}
};

class ContoCorrente: public ContoBancario {
private:
  static double spesaFissa;
public:
  // se x<spesaFissa, non avviene il deposito
   double deposita(double x) {
     return ContoBancario::deposita(x-spesaFissa);
   }

  double preleva(double x){
    return ContoBancario::preleva(x+spesaFissa);
  }

  ContoCorrente(double s=0.0): ContoBancario(s) {}
};
double ContoCorrente::spesaFissa = 1.0;

class ContoDiRisparmio: public ContoBancario {
public:
  // Invariante: saldo >= 0
  double preleva(double x){
    return x<=getSaldo() ? ContoBancario::preleva(x) : getSaldo();
  }
  // ContoBancario::deposita() non necessita di ridefinizione

  ContoDiRisparmio(double s=0.0): ContoBancario(s) {}
};

class ContoArancio: public ContoDiRisparmio {
private:
  // conto di appoggio deve essere modificabile
  ContoCorrente& appoggio;
  // ContoArancio e' un ContoDiRisparmio
```

```
  // Invariante: saldo >= 0
public:
  double preleva(double x) {
    if(x<=getSaldo() && 0<=x ) { appoggio.deposita(x); return ContoDiRisparmio::preleva(x); }
    return getSaldo();
  }

  double deposita(double x) {
    if(x>=0) { appoggio.preleva(x); return ContoDiRisparmio::deposita(x); }
    return getSaldo();
  }

  ContoArancio(ContoCorrente& cc,double s=0.0): ContoDiRisparmio(s), appoggio(cc) {}
};

#include<iostream>

int main() {
  ContoCorrente cc(2000);
  ContoArancio ca(cc,1500);
  ca.deposita(350); ca.preleva(400); cc.preleva(150);
  std::cout << cc.getSaldo() << std::endl; // stampa: 1897
  std::cout << ca.getSaldo() << std::endl; // stampa: 1450
}
```

Polimorfismo, Dynamic binding/lookup, Overriding, Distruttori virtuali, RTTI

La differenze tra i due pezzi di F e di G è il *polimorfismo* del parametro interessato:

```
void F(orario o);
...
dataora d;
F(d);
```

```
void G(const orario& o);
...
dataora d;
G(d);
```

La parola polimorfismo significa avere molte forme. In genere, il polimorfismo si verifica quando c'è una gerarchia di classi e queste sono legate dall'ereditarietà.
Il polimorfismo del C++ significa che una chiamata a una funzione membro causerà l'esecuzione di una funzione diversa a seconda del tipo di oggetto che invoca la funzione. Questo dipende a seconda del contesto di invocazione; infatti, a seconda dei casi, piuttosto che definire altre classi ed appesantire il programma, il polimorfismo consente di chiamare subito con gli strumenti che si possiedono la funzione desiderata.
Per esempio, vorremmo chiamare un metodo *stampa()* su Orario e Dataora in modo polimorfo, quindi in modo che il compilatore capisca chi deve chiamare:

```
orario::Stampa() {...}

dataora::Stampa() {...}
```

Codice esterno:

```
void printInfo(const orario& r) { r.Stampa(); }

void printInfo(const orario* p) { p->Stampa(); }
```

È qui che entra in gioco il polimorfismo. Si usa la parola chiave *virtual*, che permette di far capire al compilatore a quale classe ci stiamo riferendo, chiamando dinamicamente la funzione giusta.
Una funzione virtuale è una funzione membro dichiarata in una classe base e ridefinita (sovrascritta) da una classe derivata. Quando si fa riferimento a un oggetto di una classe derivata utilizzando un puntatore o un riferimento alla classe base, è possibile chiamare una funzione virtuale per quell'oggetto ed eseguire la versione della funzione della classe derivata.

- Le funzioni virtuali assicurano che venga richiamata la funzione corretta per un oggetto, indipendentemente dal tipo di riferimento (o puntatore) utilizzato per la chiamata alla funzione.
- Vengono utilizzate principalmente per ottenere il polimorfismo di runtime.
- Le funzioni sono dichiarate con una parola chiave virtuale nella classe base.
- La risoluzione della chiamata di funzione avviene in fase di esecuzione.

Regole per le funzioni virtuali

- Le funzioni virtuali non possono essere statiche.
- Una funzione virtuale può essere una funzione amica di un'altra classe.
- Le funzioni virtuali devono essere accessibili utilizzando un puntatore o un riferimento al tipo di classe base per ottenere il polimorfismo in fase di esecuzione.
- Il prototipo delle funzioni virtuali deve essere lo stesso sia nella classe base che in quella derivata.
- Sono sempre definite nella classe base e sovrascritte in una classe derivata. Non è obbligatorio che la classe derivata sovrascriva (o ridefinisca) la funzione virtuale; in tal caso, viene utilizzata la versione della classe base della funzione.
- Una classe può avere un distruttore virtuale, ma non può avere un costruttore virtuale.

```
class orario { ...
  virtual void Stampa(); // metodo virtuale
  ...
};

void G(const orario& o) {
  o.Stampa(); // chiamata polimorfa
}
```

```
dataora d;
orario* p = &d;
p->Stampa(); // chiamata polimorfa
```

orario diventa una classe **polimorfa**
Stampa() è un contratto **polimorfo**

Un altro concetto principale è il cosiddetto *overriding*, che significa che è un concetto che consente di definire una funzione con lo stesso nome e la stessa firma di funzione (parametri e relativi tipi di dati) sia nella classe base che nella classe derivata con una definizione di funzione diversa. Ridefinisce una funzione della classe base all'interno della classe derivata, che sovrascrive la funzione della classe base. L'override di funzione è un'implementazione del polimorfismo in tempo reale. Pertanto, sovrascrive la funzione in fase di esecuzione del programma.

Si fa riferimento ad un cosiddetto tipo *covariante*, cioè un tipo che può essere rimpiazzato quando chiamato da un altro tipo sullo stesso livello più stretto (significa semplicemente che, avendo più classi virtuali, a seconda del contesto di invocazione chiameremo una piuttosto che un'altra classe).

Overriding di metodi virtuali

- **identica segnatura**, tipo di ritorno e const incluso
- se la lista degli argomenti è identica ma cambia il tipo di ritorno il compilatore **segnala un errore**

- `virtual T1* m(...)`
  `T2` $\leq$ `T1`

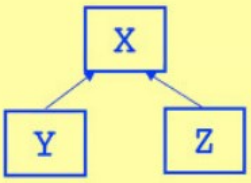
  overriding con tipo di ritorno ***covariante***:

  `T2* m(...)`

Grazie all'uso di *virtual*, siamo in grado di risolvere quello che non si poteva fare prima sulla chiamata degli stessi metodi. Segue quell'esempio specifico ma anche altri ancora:

```
class B {
public:
  virtual int f() { cout << "B::f()\n"; return 1; }
  virtual void f(string s) {cout << "B::f(string)\n";}
  virtual void g() {cout << "B::g()\n";}
};
class D1 : public B {
public:
  // Overriding di un metodo virtuale non sovraccaricato
  void g() {cout << "D1::g()\n";}
};
class D2 : public B {
public:
  // Overriding di un metodo virtuale sovraccaricato
  int f() { cout << "D2::f()\n"; return 2; }
};
class D3 : public B {
public:
  // NON è possibile modificare il tipo di ritorno
  //! void f() { cout << "D3::f()\n";} // NON COMPILA
};
class D4 : public B {
public:
  // Lista degli argomenti modificata: ridefinizione e non overriding
  int f(int) { cout << "D4::f()\n"; return 4; }
};
```

```
class X {};
class Y: public X {};
class Z: public X {};

class B {
  X x;
public:
  virtual X* m() { cout << "B::m() "; return &x; }
};

class C: public B {
  Y y;
public:
  virtual X* m() { return &y; } // overriding
};

class D: public B {
  Z z;
public:
  virtual Z* m() { return &z; } // OK, overriding covariante legale
};
int main(){
  B b; C c; D d; B* pb = &b; X x;
  Y* py = c.m(); // illegale
  X* px = c.m();
  Z* pz = d.m();
  x = *(pb->m());
  pb = &d; x = *(pb->m()); // OK
}
```

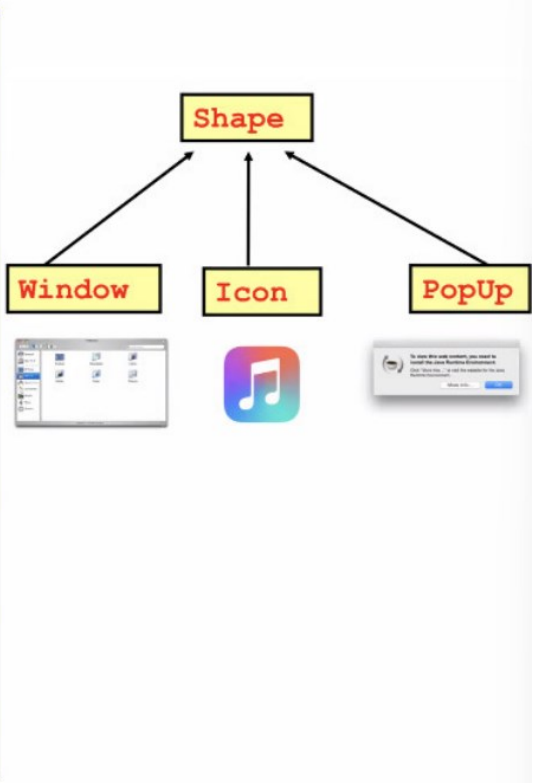


Attenzione all'overriding di metodi virtuali con parametri che prevedono valori di default

```
class B {
public:
  virtual void m(int x=0) { cout << "B::m01 ";}
};

class D: public B {
public:
  // è un overriding di B::m
  virtual void m(int x) { cout << "D::m01 ";}

  // è un nuovo metodo in D e non un overriding di B::m
  virtual void m() {cout << "D::m() ";}
};

int main(){
  B* p = new D;
  p->m(2);     // stampa D::m01 e non B::m01
  p->m();      // stampa D::m01 e non D::m()
  p->B::m();   // stampa B::m01 e non D::m01
}
```

Il polimorfismo in C++ ci permette di riutilizzare il codice creando una funzione utilizzabile per più scopi (semplicemente grazie anche al fatto che il compilatore capisce dinamicamente cosa deve fare). Possiamo anche rendere polimorfi gli operatori e usarli per aggiungere non solo numeri, ma anche per combinare stringhe. In questo modo si risparmia tempo e si ottiene un programma più snello.
Esempio discorsivo ma utile che il prof propone è quello di un Windows Manager:

```
class Shape {
  ...
  virtual void draw(Position){...}
};
class Window: public Shape {
  ...
  void draw(Position) {...}
};
class Icon: public Shape {
  ...
  void draw(Position) {...}
};
class PopUp: public Shape {
  ...
  void draw(Position) {...}
};

class DesktopManager {
  ...
  void show(const Shape& s) {
    ...
    Position p = computePosition();
    s.draw(p);
    ...
  }
};
```



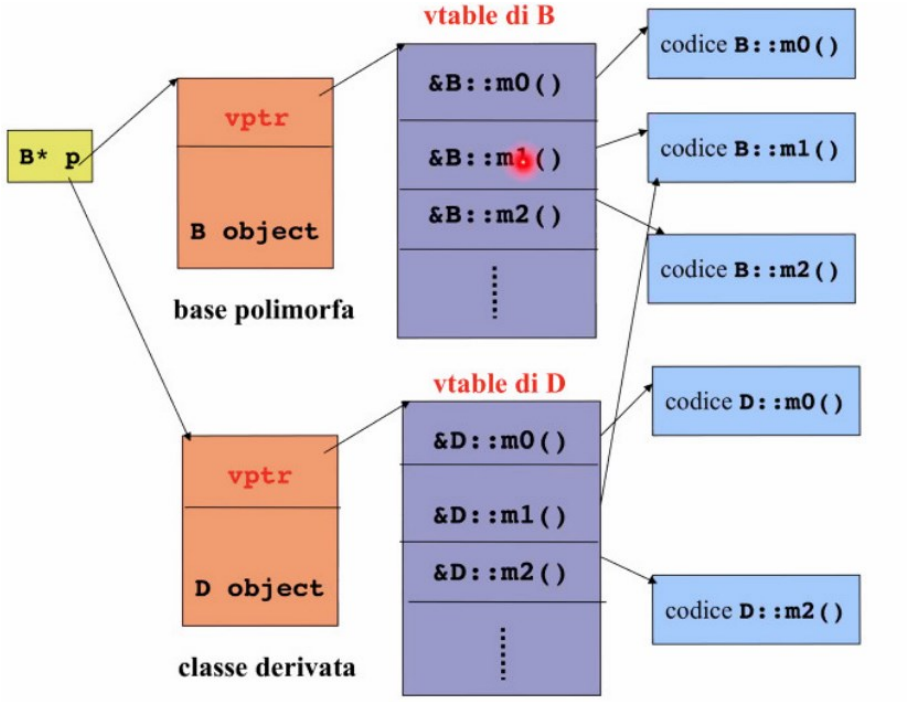
Quanto visto con virtual viene definito come *dynamic/late binding/lookup*. Il binding dinamico si verifica quando un puntatore o un riferimento viene associato a una funzione membro in base al tipo dinamico dell'oggetto (comprendendo l'istanza della variabile in fase di esecuzione).
Il late binding comporta un overhead importante (approssimativamente 6-13% del tempo di esecuzione con overhead anche fino al 50%). Il prof riporta un esempio del 4% oppure, qualche anno prima, del 15% con *clang*.

Overhead
"Sono le risorse necessarie per impostare un'operazione. Potrebbe sembrare non correlato, ma necessario. È come quando si deve andare da qualche parte e si ha bisogno di un'auto. Ma sarebbe molto oneroso prendere un'auto per percorrere la strada, quindi è meglio andare a piedi. Tuttavia, il dispendio di tempo ne varrebbe la pena se si dovesse andare dall'altra parte del Paese.
In informatica, a volte usiamo le auto per andare in strada perché non abbiamo un modo migliore o perché non vale la pena di "imparare a camminare"."

Quando si usano i metodi virtuali, si ha a che fare con la *virtual method table*. Nella programmazione informatica, una tabella dei metodi virtuali (VMT), una tabella delle funzioni virtuali, una tabella delle chiamate virtuali, una tabella di dispacciamento, una vtable o una vftable è un meccanismo utilizzato in un linguaggio di programmazione per supportare il dynamic binding/dispatch (o il binding dei metodi a tempo di esecuzione).

Ogni volta che una classe definisce una funzione (o un metodo) virtuale, la maggior parte dei compilatori aggiunge alla classe una variabile membro nascosta che punta a un array di puntatori a funzioni (virtuali) chiamato tabella dei metodi virtuali. Questi puntatori vengono utilizzati in fase di esecuzione per invocare le implementazioni delle funzioni appropriate, perché in fase di compilazione potrebbe non essere ancora

noto se deve essere chiamata la funzione di base o una funzione derivata implementata da una classe che eredita dalla classe di base.
Concretamente, se non ben definite le classi virtuali (diverse volte è un errore pesantino in Qt, significa che mancano dei virtual oppure non è ben stata scritta tutta la struttura), si hanno errori proprio di questo tipo.
Strutturalmente e con un esempio:




```
class B {
public:
  FunctionPointer* vptr; // vpointer aggiunto dal compilatore
  virtual void m0() {}
  virtual void m1() {}
  virtual void m2() {}
};

class D: public B {
public:
  virtual void m0() {} // overriding
  virtual void m2() {} // overriding
};
```

```
B* p;

p=&b;
p->m2();
// chiama la funzione all'indirizzo *((p->vptr)+2)
// cioè chiama B::m2()

p=&d;
p->m2();
// chiama la funzione all'indirizzo *((p->vptr)+2)
// cioè chiama D::m2()
```

Nell'esempio della classe *Poligono*, aggiungiamo il virtual:

```
// file polv.h
#ifndef POLV_H
#define POLV_H

class punto {
 private:
  double x, y;
 public:
  punto(double a=0, double b=0): x(a), y(b) {}
  static double lung(const punto& p1, const punto& p2);
};

class poligono { // classe polimorfa
 protected:
  int nvertici;
  punto* pp;
 public:
  poligono(int n, const punto v[]);
  ~poligono();
  poligono(const poligono& pol);
  poligono& operator=(const poligono& pol);
  // contratto: ritorno il perimetro del poligono di invocazione
  virtual double perimetro() const; // metodo virtuale
};
#endif
```

```
// file triv.h
#ifndef TRIV_H
#define TRIV_H
#include "polv.h"

class triangolo: public poligono {
 public:
  triangolo(const punto v[]);
  // eredita perimetro()
  // contratto: ritorno l'area del triangolo di invocazione
  virtual double area() const; // nuovo metodo virtuale
};
#endif

// file triv.cpp
#include <iostream>
#include <math.h>
#include "triv.h"

triangolo::triangolo(const punto v[]) : poligono(3, v) {}

// formula di Erone
double triangolo::area() const {
  double a = punto::lung(pp[1], pp[0]);
  double b = punto::lung(pp[2], pp[1]);
  double c = punto::lung(pp[0], pp[2]); double p = (a + b + c)/2;
  return sqrt(p*(p-a)*(p-b)*(p-c));
}
```

```
// Contratto: stampa il perimetro del poligono *p
void stampa_perimetro(poligono* p) {
  cout << "Il perimetro è " << p->perimetro() << endl; // chiamata polimorfa
}

// Contratto: stampa l'area del triangolo t
void stampa_area_triangolo(const triangolo& t) {
  cout << "L'area è " << t.area() << endl; // chiamata polimorfa
}

int main() {
  int i; punto vv[4]; double x,y;
  cout << "Scrivi le coordinate di 4 vertici:\n";
  for (i=0; i<4; i++) { cin >> x >> y; v[i]=punto(x,y); }
  poligono po(4, vv);
  cout << "Scrivi le coordinate di 3 vertici:\n";
  for (i=0; i<3; i++) { cin >> x >> y; v[i]=punto(x,y); }
  triangolo tr(vv) ;
  cout << "Scrivi le coordinate di 3 vertici, con angolo retto sul primo:\n";
  for (i=0; i<3; i++) { cin >> x >> y; v[i]=punto(x,y); }
  tri_rettangolo rr(vv) ;

  cout << "Poligono:\n";
  stampa_perimetro(&po);
  cout << "\nTriangolo:\n";
  stampa_perimetro(&tr); stampa_area_triangolo(tr);
  cout << "\nTriangolo rettangolo:\n";
  stampa_perimetro(&rr); stampa_area_triangolo(rr);
}
```

Sia *D* una classe derivata da *B*. Consideriamo come situazione:

```
D* pd = new D;
B* pb = pd; // pb ha tipo dinamico D*
delete pb;
```

Concretamente, quando si ha di mezzo il polimorfismo, nel caso di deallocazione/distruzione di oggetti usiamo i *distruttori virtuali* come segue nell'esempio.
Un distruttore virtuale viene utilizzato per liberare lo spazio di memoria allocato dall'oggetto o dall'istanza della classe derivata, mentre elimina le istanze della classe derivata utilizzando un oggetto puntatore alla classe base. Un distruttore di classe base o genitore utilizza la parola chiave virtual che assicura che sia il distruttore della classe base che quello della classe derivata vengano chiamati in fase di esecuzione, ma chiama prima la classe derivata e poi la classe base per liberare lo spazio occupato da entrambi i distruttori.

- Perché si usa il distruttore virtuale in C++?

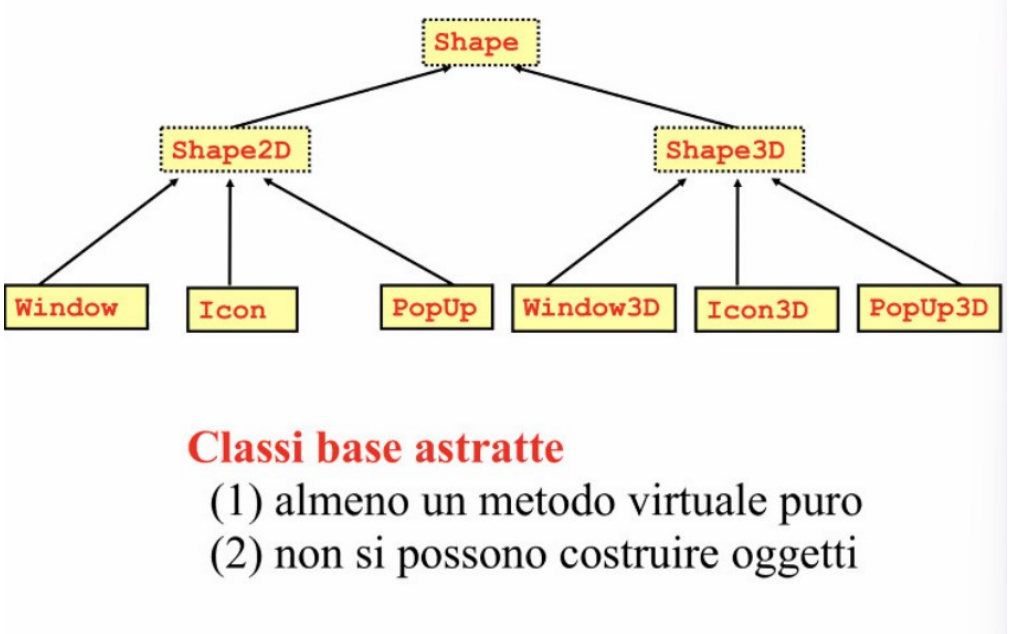
Quando un oggetto della classe esce dallo scope o l'esecuzione della funzione main() sta per terminare, un distruttore viene automaticamente richiamato nel programma per liberare lo spazio occupato dalla funzione distruttore della classe. Quando viene eliminato un oggetto puntatore della classe base che punta alla classe derivata, viene richiamato solo il distruttore della classe madre, grazie al bind anticipato del compilatore. In questo modo, si evita di chiamare il distruttore della classe derivata, con conseguenti perdite di memoria nel programma. Quando si usa la parola chiave virtual preceduta dal segno tilde (~) di distruttore all'interno della classe base, si garantisce che prima venga chiamato il distruttore della classe derivata. Poi viene richiamato il distruttore della classe base per liberare lo spazio occupato da entrambi i distruttori nella classe ereditaria.

```
class B {
private:
  int* p;
public:
  B(int n, int v) : p(new int[n]) {
    for(int i=0; i<n; i++) p[i]=v;
  };
  virtual ~B() { delete[] p; cout <<"~B() "; }; // distruttore virtuale
};

class C : public B {
private:
  int* q;
public:
  C(int sizeB, int sizeC, int v) : B(sizeB, v), q(new int[sizeC]) {
    for(int i=0; i<sizeC; i++) q[i]=v;
  };
  virtual ~C() {delete[] q; cout <<"~C() ";};
};

int main() {
  C* q = new C(4,2,18);
  B* p=q; // puntatore polimorfo
  delete p; // distruzione virtuale: invoca ~C()
}
// stampa: ~C() ~B()
// se ~B() non fosse virtuale verrebbe invocato solamente ~B() !
```

Torniamo agli esempi visti ed espandiamo il quadro ulteriormente (per esempio, tornando al discorso del window manager).
Introduciamo il concetto di *astratto* in C++, il che significa che esiste un metodo di qualsiasi tipo dichiarato come *virtual (metodo) =0*. Viene anche detto virtuale puro.
Una classe astratta in C++ ha almeno una funzione virtuale pura per definizione. In altre parole, una funzione che non ha definizione. I discendenti della classe astratta devono definire la funzione virtuale pura, altrimenti la sottoclasse diventerebbe una classe astratta a tutti gli effetti.

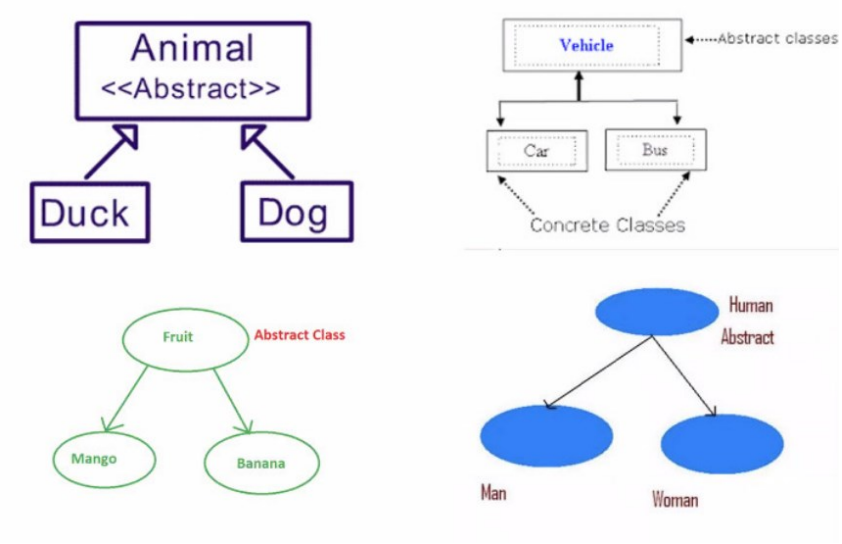
Le classi astratte sono utilizzate per esprimere concetti ampi da cui possono essere derivate classi più concrete. Un oggetto di tipo classe astratta non può essere creato. Per i tipi di classe astratta, tuttavia, è possibile utilizzare puntatori e riferimenti. Quando si crea una classe astratta, si deve dichiarare almeno una funzione membro virtuale pura. La sintassi dello specificatore puro (= 0) viene utilizzata per dichiarare una funzione virtuale.

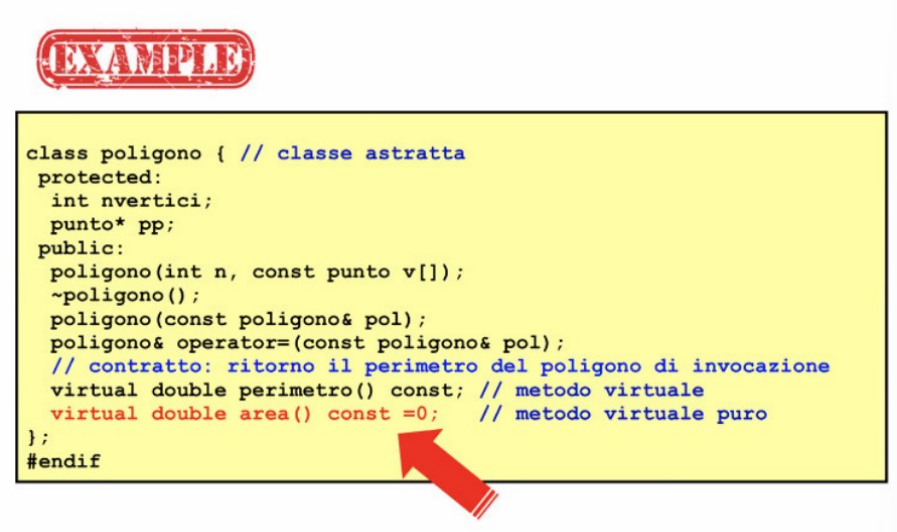
Una classe che ridefinisce un metodo astratto viene definita *concreta* e, a differenza di quella astratta, può essere istanziata. I metodi astratti garantiscono l'idea delle interfacce, quindi dei metodi che possono sempre essere riutilizzati con quella struttura con il vincolo di ridefinizione quando necessario.



```
class B { // classe base astratta
public:
  virtual void f() = 0;
};

class C: public B { // sottoclasse astratta
public:
  void g() {cout << "C::g() ";}
};

class D: public B { // sottoclasse concreta
public:
  virtual void f() {cout << "D::f() ";}
};

int main() {
  // C c; // Illegale: "cannot declare c of type C ..."
  D d;    // OK, D è concreta
  B* p;   // OK, puntatore a classe astratta
  p = &d; // puntatore (super)polimorfo
  p->f(); // stampa: D::f()
}
```

Vediamo una serie di esempi e anche l'applicazione su Poligono:





Attenzione, come segue, a non dichiarare/allocare oggetti astratti:

Distruttore virtuale puro

```
class Base {
// classe astratta
 public:
  virtual ~Base() =0;
  // distruttore virtuale puro
};
Base::~Base() {} // definizione comunque obbligatoria!

int main() {
Base b;              // "cannot declare b of abstract type Base"
Base* p = new Base; // "cannot allocate an object of abstract type Base"
}
```

Segue l'esempio della classe Lavoratore:

```
class Lavoratore { // classe base astratta
public:
  virtual ~Lavoratore() {}
  Lavoratore(string s): nome(s) {};
  string getNome() const {return nome;};
  virtual double stipendio() const = 0;            // virtuale pura
  virtual void printInfo() const {cout << nome;}; // virtuale
private:
  string nome;
};
```

```
class Dirigente : public Lavoratore {
private:
  double fissoMensile; // stipendio fisso
public:
  Dirigente(string s, double d = 0):
                              Lavoratore(s), fissoMensile(d) {}

  virtual double stipendio() const { // implementazione
    return fissoMensile;
  }

  virtual void printInfo() const { // overriding
    cout << "Dirigente ";
    Lavoratore::printInfo(); // invocazione statica
  }
};
```

```
class Rappresentante : public Lavoratore {
private:
  double baseMensile; // stipendio base fisso
  double commissione; // commissione per pezzo venduto
  int tot;            // pezzi venduti in un mese
public:
  Rappresentante(string s, double d=0, double e=0, int x=0):
    Lavoratore(s), baseMensile(d), commissione(e), tot(x) {}

  virtual double stipendio() const { // implementazione
    return baseMensile + commissione*tot;
  }
  virtual void printInfo() const { // overriding
    cout << "Rappresentante "; Lavoratore::printInfo();
  };
};
```

```
class LavoratoreOre : public Lavoratore {
private:
  double pagaOraria;
  double oreLavorate;  // ore lavorate nel mese
public:
  LavoratoreOre(string s, double d=0, double e=0):
    Lavoratore(s), pagaOraria(d), oreLavorate(e) {}

  virtual double stipendio() const { // implementazione
    if ( oreLavorate <= 160 ) // nessuno straordinario
      return pagaOraria*oreLavorate;
    else  // le ore straordinarie sono pagate il doppio
      return 160*pagaOraria+(oreLavorate-160)*2*pagaOraria
  };
  virtual void printInfo() const { // overriding
    cout << "Lavoratore a ore "; Lavoratore::printInfo();
  };
};
```

Si specifica che la chiamata superpolimorfa significa semplicemente che, a livello di polimorfismo, può chiamare diverse/più classi virtuali.

```
// funzione esterna
void stampaStipendio(Const Lavoratore& x) {
   x.printInfo(); // chiamata polimorfa
   cout << " in questo mese ha guadagnato "
 << x.stipendio() <<" Euro.\n"; // chiamata (super)polimorfa
}

int main() {
   Dirigente d("Pippo", 4000);
   Rappresentante r("Topolino",1000,3,250);
   LavoratoreOre l("Pluto",15,170);
   stampaStipendio(d);
   stampaStipendio(r);
   stampaStipendio(l);
}
```

```
// STAMPA:
Dirigente Pippo in questo mese ha guadagnato 4000 Euro.
Rappresentante Topolino in questo mese ha guadagnato 1750 Euro.
Lavoratore a ore Pluto in questo mese ha guadagnato 2700 Euro.
```

Altro concetto molto importante in programmazione ad oggetti: *RTTI: Run-Time Type Information*.

TD(ptr) tipo dinamico di puntatore polimorfo **ptr**

TD(ref) tipo dinamico di riferimento polimorfo **ref**

In C++, RTTI (Run-time type information) è un meccanismo che espone informazioni sul tipo di dati di un oggetto in fase di esecuzione ed è disponibile solo per le classi che hanno almeno una funzione virtuale. Consente di determinare il tipo di un oggetto durante l'esecuzione del programma.

Cast a tempo di esecuzione/Runtime casts

Il cast a runtime, che verifica la validità del cast, è l'approccio più semplice per accertare il tipo a runtime di un oggetto che utilizza un puntatore o un riferimento. È particolarmente utile quando si deve eseguire il cast di un puntatore da una classe base a un tipo derivato. Quando si ha a che fare con la gerarchia ereditaria delle classi, di solito è necessario eseguire il casting di un oggetto. Esistono due tipi di fusione:

- Upcasting: Quando un puntatore o un riferimento di un oggetto di classe derivata viene trattato come un puntatore di classe base.
- Downcasting: Quando un puntatore o un riferimento di classe base viene convertito in un puntatore di classe derivata.

Scritto da Gabriel

Utilizzo di "dynamic_cast": In una gerarchia di ereditarietà, viene utilizzato per il downcasting di un puntatore di classe base a una classe figlia. Se il casting ha successo, restituisce un puntatore del tipo convertito; tuttavia, fallisce se si tenta di eseguire il casting di un tipo non valido, come un puntatore a un oggetto che non è del tipo della sottoclasse desiderata.

Ad esempio, dynamic_cast utilizza RTTI e il seguente programma fallisce con l'errore "cannot dynamic_cast `b' (of type `class B*') to type `class D*' (source type is not polymorphic)" perché non esiste alcuna funzione virtuale nella classe base B.

Un operatore in grado di determinare il tipo di qualsiasi espressione a runtime è *typeid*, di cui si riportano esempi e comportamento.

```
#include <typeinfo> // includere sempre questo header file
#include <iostream> // per usare typeid

int main() {
 int i=5;
 std::cout << typeid(i).name() << endl;     // stampa: i(nt)
 std::cout << typeid(3.14).name() << endl; // stampa: d(ouble)
 if (typeid(i) == typeid(int)) std::cout << "Yes";
}
```

L'operatore `typeid` ha come argomento una espressione o un tipo qualsiasi e ritorna un oggetto della classe `type_info`. La definizione della classe `type_info` è nel file header `typeinfo` e rende disponibili i seguenti metodi comuni a tutte le implementazioni del compilatore.

```
class type_info {
// rappresentazione dipendente dall'implementazione
private:
  type_info();
  type_info(const type_info&);
  type_info& operator=(const type_info&);
public:
  bool operator==(const type_info&) const;
  bool operator!=(const type_info&) const;
  const char* name() const;
};
```

Comportamento di `typeid`

[1] Se l'espressione operando di `typeid` è un riferimento `ref` ad una classe che **contiene almeno un metodo virtuale**, cioè una **classe polimorfa**, allora `typeid` restituisce un oggetto di `type_info` che rappresenta il tipo dinamico di `ref`.

[2] Se l'espressione operando di `typeid` è un puntatore **dereferenziato** `*punt`, dove `punt` è un puntatore ad un **tipo polimorfo**, allora `typeid` restituisce un oggetto di `type_info` che rappresenta il tipo `T` dove `T*` è il tipo dinamico di `punt`.

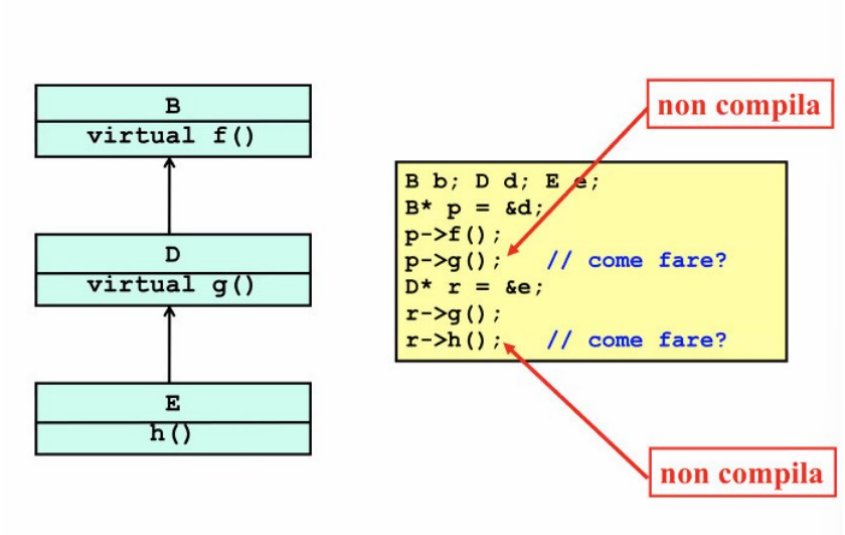
ATTENZIONE

(A) Se la classe non contiene metodi virtuali allora `typeid` restituisce il tipo statico del riferimento o del puntatore dereferenziato.

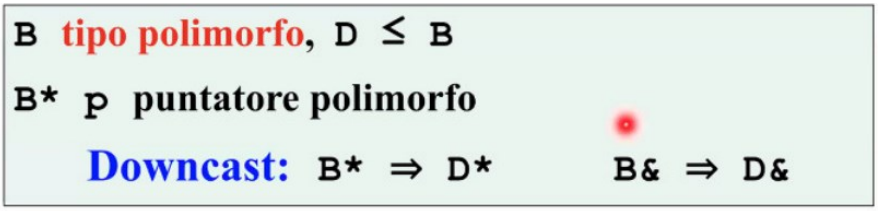
(B) `typeid` su un puntatore (non dereferenziato) restituisce sempre il tipo statico del puntatore.

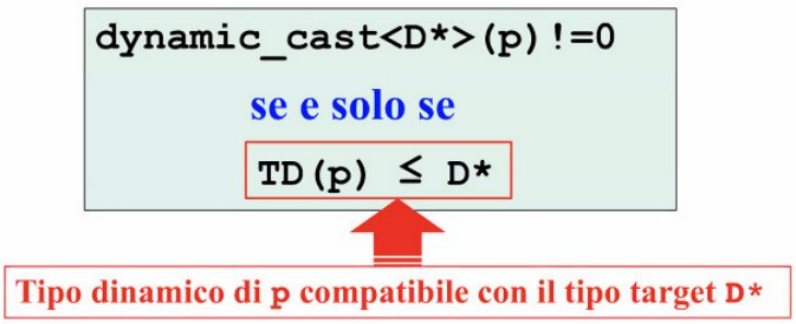
```
class A { public: virtual ~A() {} };
class B : public A {};
class D : public B {};

#include<typeinfo>
#include<iostream>

int main() {
  B b; D d;
  B& rb = d;
  A* pa = &b;
  if(typeid(rb) == typeid(B)) std::cout << '1';
  if(typeid(rb) == typeid(D)) std::cout << '2';
  if(typeid(*pa) == typeid(A)) std::cout << '3';
  if(typeid(*pa) == typeid(B)) std::cout << '4';
  if(typeid(*pa) == typeid(D)) std::cout << '5';
  // stampa: 24
}
```

Nel seguente esempio, si usa quanto introdotto prima: infatti i tipi non corrispondono e va eseguito un *cast a tempo di esecuzione*, *verso l'alto oppure verso il basso*.



Fondamentale in questo è il dynamic_cast. Normalmente questo viene usato per una conversione verso il basso a runtime, naturalmente quando ciò è possibile.

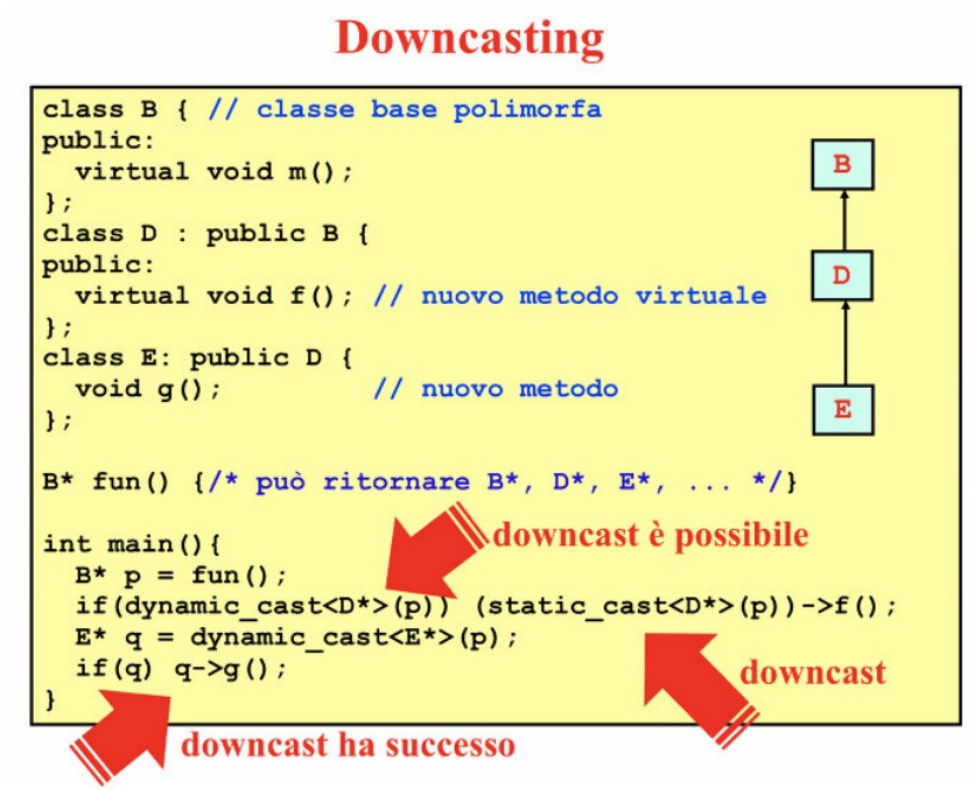




Scritto da Gabriel

Quando i cast falliscono, lo standard prevede una apposita eccezione → *bad_cast*, presente nell'header *typeinfo*. Segue un esempio di bad cast e di downcasting.

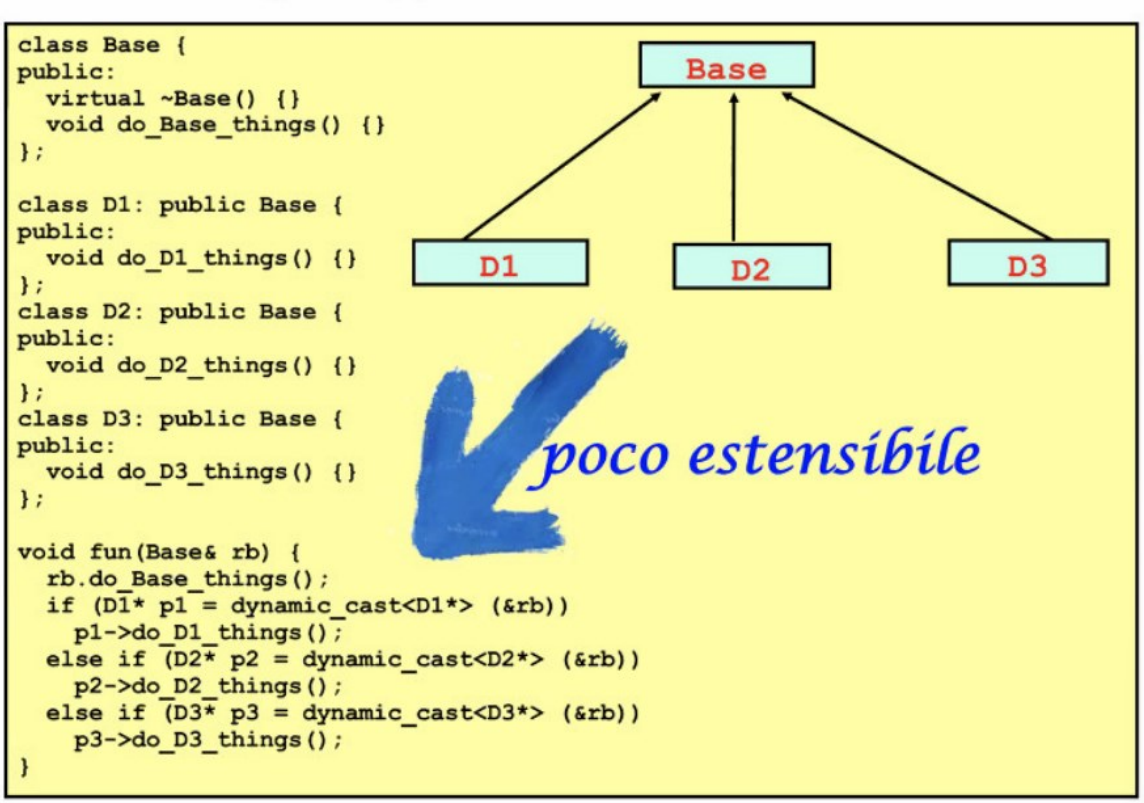
```
class X { public: virtual ~X() {} };
class B { public: virtual ~B() {} };
class D : public B {};

#include<typeinfo>
#include<iostream>

int main() {
  D d;
  B& b = d; // upcast
  try {
    X& xr = dynamic_cast<X&>(b);
  } catch(std::bad_cast e) {
      std::cout << "Cast fallito!" << std::endl;
    }
}
```



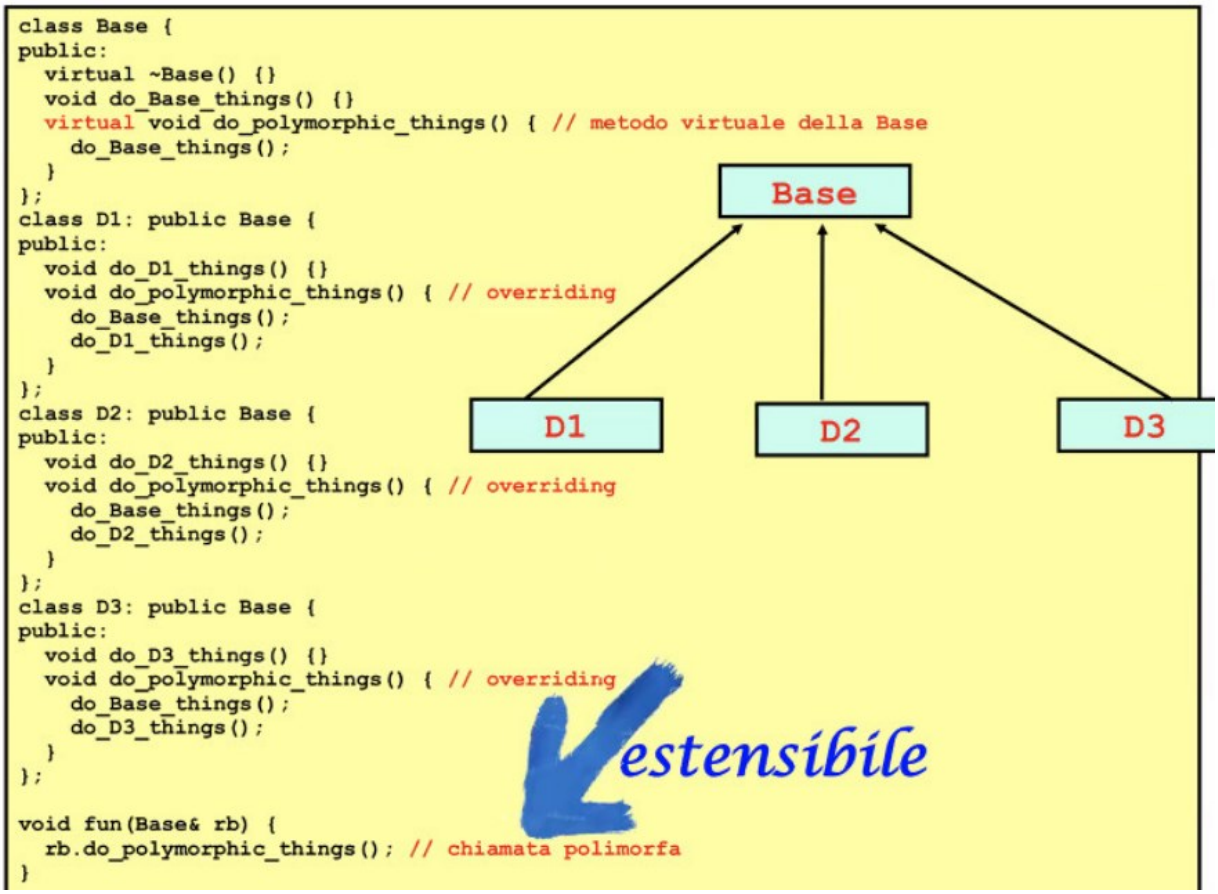
Sorgono però varie critiche: infatti, il downcasting è pesante e comunque non ottimale da un punto di vista di design. Infatti

- Il downcasting si usa solo quando necessario
- È meglio non fare type checking dinamico inutile
- Ove possibile, usare metodi virtuali nelle classi base piuttosto che usare type checking dinamico

Un esempio in cui una serie di classi ereditano da una base e si opera una conversione per caso con il dynamic_cast è poco estensibile, in quanto pesante computazionalmente e una soluzione poco implementabile, specie se si usano molte classi:



Con una situazione di questo tipo, invece, la cosa migliora decisamente:



L'esempio su impiegato che applica il polimorfismo. Molto meglio il cast alla bisogna piuttosto che usare *typeid.*

```
class impiegato { // classe base astratta
protected: static double stipBase;
public:    virtual double stipendioBase() const =0;
};
double impiegato::stipBase = 1500;

class manager : public impiegato {
public:
  virtual double stipendioBase() const {return 2*impiegato::stipBase;}
};

class programmatore : public impiegato {
private: double bonus; // campo dati proprio
public:
  programmatore(double b): bonus(b) {}
  virtual double stipendioBase() const {return impiegato::stipBase;}
  double getBonus() const {return bonus;}; // metodo non virtuale
};

class GoogleAdmin {
public:
  static double stipendio(const impiegato& p) {
    const programmatore* q = dynamic_cast<const programmatore*>(&p);
    // se il cast ha successo,
    // cioè il tipo dinamico di &p è sottotipo di programmatore*
    if(q) return q->stipendioBase() + q->getBonus();
    // altrimenti (q==0),
    // cioè il tipo dinamico di &p non è sottotipo di programmatore*
    else return p.stipendioBase();
  }
};
```

```
class impiegato { // classe astratta
protected: static double stipBase;
public:    virtual double stipendioBase() const =0;
};
double impiegato::stipBase = 1500;

class manager : public impiegato {
public:
  virtual double stipendioBase() const {return 2*impiegato::stipBase;}
};

class programmatore : public impiegato {
private: double bonus;
public:
  programmatore(double b): bonus(b) {}
  virtual double stipendioBase() const {return impiegato::stipBase;}
  double getBonus() const {return bonus;}; // metodo non virtuale
};

class GoogleAdmin {
public:
static double stipendio(impiegato* p) {
    if(typeid(*p)==typeid(programmatore)) { // non è codice estensibile!
      programmatore* q = dynamic_cast<programmatore*>(p);
      return q->stipendioBase() + q->getBonus();
    }
    else return p->stipendioBase();
  }
};
```

I principi di design che si seguono in OOP è *SOLID*.

I principi SOLID sono stati introdotti per la prima volta dal famoso scienziato informatico Robert J. Martin (alias Uncle Bob) in un suo articolo del 2000.
Non è quindi una sorpresa che tutti questi concetti di codifica pulita, architettura orientata agli oggetti e design pattern siano in qualche modo collegati e complementari tra loro.

Servono tutti allo stesso scopo:
"Creare codice comprensibile, leggibile e testabile su cui molti sviluppatori possano lavorare in modo collaborativo".
Analizziamo ogni principio uno per uno. Seguendo l'acronimo SOLID, essi sono:

- Principio della responsabilità unica/The Single Responsibility Principle
- Principio dell'apertura-chiusura/ The Open-Closed Principle
- Il principio di sostituzione di Liskov/The Liskov Substitution Principle
- Principio della segregazione dell'interfaccia/ The Interface Segregation Principle
- Il principio di inversione delle dipendenze/ The Dependency Inversion Principle

1) Il principio della responsabilità unica afferma che una classe deve fare una sola cosa e quindi deve avere un solo motivo per cambiare.
   Per enunciare questo principio in modo più tecnico: Solo un potenziale cambiamento (logica del database, logica di registrazione e così via) nelle specifiche del software dovrebbe essere in grado di influenzare le specifiche della classe.

2) Il principio aperto-chiuso prevede che le classi siano aperte all'estensione e chiuse alla modifica. Modificare significa cambiare il codice di una classe esistente, mentre estendere significa aggiungere nuove funzionalità.
   Questo principio vuole dire che: Dovremmo essere in grado di aggiungere nuove funzionalità senza toccare il codice esistente della classe. Questo perché ogni volta che modifichiamo il codice esistente, corriamo il rischio di creare potenziali bug. Quindi, se possibile, dovremmo evitare di toccare il codice di produzione testato e affidabile (per lo più).
   Ma come si fa ad aggiungere nuove funzionalità senza toccare la classe? Di solito lo si fa con l'aiuto di interfacce e classi astratte.

3) Il principio di sostituzione di Liskov afferma che le sottoclassi dovrebbero essere sostituibili alle loro classi base.
   Ciò significa che, dato che la classe B è una sottoclasse della classe A, dovremmo essere in grado di passare un oggetto della classe B a qualsiasi metodo che si aspetta un oggetto della classe A e il metodo non dovrebbe dare alcun risultato strano in questo caso.
   Questo è il comportamento atteso, perché quando si usa l'ereditarietà si assume che la classe figlia erediti tutto ciò che ha la superclasse. La classe figlia estende il comportamento, ma non lo restringe mai.
   Pertanto, quando una classe non obbedisce a questo principio, si verificano alcuni bug difficili da rilevare.

4) Segregazione significa mantenere le cose separate e il principio di segregazione delle interfacce riguarda la separazione delle interfacce.
   Il principio afferma che molte interfacce specifiche per il client sono meglio di un'unica interfaccia generale. I client non devono essere costretti a implementare una funzione di cui non hanno bisogno.

5) Il principio dell'inversione delle dipendenze afferma che le nostre classi dovrebbero dipendere da interfacce o classi astratte invece che da classi e funzioni concrete.

Qualche esercizio:

```
class A {
private:
  void h() {cout<<" A::h ";}
public:
  virtual void g() {cout <<" A::g ";}
  virtual void f() {cout <<" A::f "; g(); h();}
  void m() {cout <<" A::m "; g(); h();}
  virtual void k() {cout <<" A::k "; g(); h(); m(); }
  A* n() {cout <<" A::n "; return this;}
};

class B: public A {
private:
  void h() {cout <<" B::h ";}
public:
  virtual void g() {cout <<" B::g ";}
  void m() {cout <<" B::m "; g(); h();}
  void k() {cout <<" B::k "; g(); h(); m();}
  B* n() {cout <<" B::n "; return this;}
};

B* b = new B(); A* a = new B();
```

```
// COMPILA?
// ERRORE RUN-TIME?
// COSA STAMPA?

b->f();
b->m();
b->k();
a->f();
a->m();
a->k();
(b->n())->g();
(b->n())->n()->g();
(a->n())->g();
(a->n())->m();
```

Stampe:
A::f B::g A::h
B::m B::g B::h
B::k B::g B::h B::m B::g B::h
A::f B::g A::h
B::k B::g B::h B::m B::g B::h
B::n B::g
(Le ultime 3 danno errore)

Esercizio 11.18 (libro)

Si consideri il seguente modello di realtà concernente i file audio memorizzati in un riproduttore audio digitale iZod©.

(A) Definire la seguente gerarchia di classi.

1. Definire una classe base polimorfa astratta `FileAudio` i cui oggetti rappresentano un file audio memorizzabile in un iZod. Ogni `FileAudio` è caratterizzato dal titolo (una stringa) e dalla propria dimensione in MB. La classe è astratta in quanto prevede i seguenti **metodi virtuali puri**:
   - un metodo di "clonazione": `FileAudio* clone()`.
   - un metodo `bool qualita()` con il seguente contratto: `f->qualita()` ritorna true se il file audio `*f` è considerato di qualità, altrimenti ritorna false.
2. Definire una classe concreta `Mp3` derivata da `FileAudio` i cui oggetti rappresentano un file audio in formato mp3. Ogni oggetto `Mp3` è caratterizzato dal proprio bitrate espresso in Kbit/s. La classe `Mp3` implementa i metodi virtuali puri di `FileAudio` come segue:
   - per ogni puntatore `p` a `Mp3`, `p->clone()` ritorna un puntatore ad un oggetto `Mp3` che è una copia di `*p`.
   - per ogni puntatore `p` a `Mp3`, `p->qualita()` ritorna true se il bitrate di `*p` è $\geq$ 192 Kbit/s, altrimenti ritorna false.
3. Definire una classe concreta `WAV` derivata da `FileAudio` i cui oggetti rappresentano un file audio in formato WAV. Ogni oggetto `WAV` è caratterizzato dalla propria frequenza di campionamento espressa in kHz e dall'essere lossless oppure no (cioè con compressione senza perdita oppure con perdita). La classe `WAV` implementa i metodi virtuali puri di `FileAudio` come segue:
   - per ogni puntatore `p` a `WAV`, `p->clone()` ritorna un puntatore ad un oggetto `WAV` che è una copia di `*p`.
   - per ogni puntatore `p` a `WAV`, `p->qualita()` ritorna true se la frequenza di campionamento di `*p` è $\geq$ 96 kHz, altrimenti ritorna false.

(B) Definire una classe `iZod` i cui oggetti rappresentano i brani memorizzati in un iZod. La classe `iZod` deve soddisfare le seguenti specifiche:

1. È definita una classe annidata `Brano` i cui oggetti rappresentano un brano memorizzato nell'iZod. Ogni oggetto `Brano` è rappresentato da un puntatore polimorfo ad un `FileAudio`.
   - La classe `Brano` deve essere dotata di un opportuno costruttore `Brano(FileAudio*)` con il seguente comportamento: `Brano(p)` costruisce un oggetto `Brano` il cui puntatore polimorfo punta ad una copia dell'oggetto `*p`.
   - La classe `Brano` ridefinisce costruttore di copia profonda, assegnazione profonda e distruttore profondo.
2. Un oggetto di `iZod` è quindi caratterizzato da un vector di oggetti di tipo `Brano` che contiene tutti i brani memorizzati nell'iZod.
3. La classe `iZod` rende disponibili i seguenti metodi:
   - Un metodo `vector<Mp3> mp3(double, int)` con il seguente comportamento: una invocazione `iz.mp3(dim,br)` ritorna un vector di oggetti `Mp3` contenente tutti e soli i file audio in formato mp3 memorizzati nell'iZod `iz` che: (i) hanno una dimensione $\geq$ `dim` e (ii) hanno un bitrate $\geq$ `br`.
   - Un metodo `vector<FileAudio*> braniQual()` con il seguente comportamento: una invocazione `iz.braniQual()` ritorna il vector dei puntatori ai `FileAudio` memorizzati nell'iZod `iz` che: (i) sono considerati di qualità e (ii) se sono dei file audio `WAV` allora devono essere lossless.
   - Un metodo `void insert(Mp3*)` con il seguente comportamento: una invocazione `iz.insert(p)` inserisce il nuovo oggetto `Brano(p)` nel vector dei brani memorizzati nell'iZod `iz` se il file audio mp3 `*p` non è già memorizzato in `iz`, mentre se il file audio `*p` risulta già memorizzato non provoca alcun effetto.

```
#include<string>
#include<typeinfo>

class FileAudio {
private:
  std::string titolo;
  double size;
public:
  virtual FileAudio* clone() const = 0;
  virtual bool qualita() const =0;
  virtual ~FileAudio() {}
  double getSize() const {return size;}
  virtual bool operator==(const FileAudio& f) const {
    return typeid(*this) == typeid(f) && titolo == f.titolo && size == f.size;
  }
};

class Mp3: public FileAudio {
private:
  unsigned int Kbits;
public:
  static const unsigned int sogliaQualita;
  Mp3* clone() const override {
    return new Mp3(*this);
  }
  bool qualita() const override {return Kbits>= sogliaQualita;}
  unsigned int getBitrate() const {return Kbits;}
  bool operator==(const FileAudio& f) const override {
    return FileAudio::operator==(f)
      && Kbits == static_cast<const Mp3&>(f).Kbits;
  }
};
const unsigned int Mp3::sogliaQualita = 192;

class WAV: public FileAudio {
private:
  unsigned int frequenza;
  bool lossLess;
public:
  static const unsigned int sogliaQualita;
  WAV* clone() const override {
    return new WAV(*this);
```

```
  }
  bool qualita() const override {return frequenza >= sogliaQualita;}
  bool getLossLess() const {return lossLess;}
  bool operator==(const FileAudio& f) const override {
    return FileAudio::operator==(f)
      && frequenza == static_cast<const WAV&>(f).frequenza &&
      lossLess == static_cast<const WAV&>(f).lossLess;
  }
};
const unsigned int WAV::sogliaQualita = 96;

#include<vector>

class iZod {
private:
  class Brano {
  public:
    FileAudio* ptr; // puntatore (super)polimorfo
    // conversione FileAudio* => Brano
    Brano(FileAudio* p): ptr(p->clone()) {}
    Brano(const Brano& b): ptr(b.ptr->clone()) {}
    Brano& operator=(const Brano& b) {
      if(this != &b) {
        delete ptr;
        ptr = b.ptr->clone();
      }
      return *this;
    }
    ~Brano() {delete ptr;}
  };

  std::vector<Brano> brani;

public:
  std::vector<Mp3> mp3(double dim, int br) const {
    std::vector<Mp3> v;
    for(std::vector<Brano>::const_iterator cit = brani.begin(); cit != brani.end(); ++cit) {
      Mp3* p = dynamic_cast<Mp3*> (cit->ptr);
      if(p != nullptr && p->getSize() >= dim && p->getBitrate() >= br)
        v.push_back(*p);
    }
    return v;
  }

  std::vector<FileAudio*> braniQual() const {
    std::vector<FileAudio*> v;
    for(auto cit = brani.begin(); cit != brani.end(); cit++)
      if((cit->ptr)->qualita() &&
         (dynamic_cast<WAV*>(cit->ptr) == nullptr ||
          static_cast<WAV*>(cit->ptr)->getLossLess()))
      v.push_back(cit->ptr);
    return v;
  }

  void insert(Mp3* p) {
    bool found = false;
    for(auto it = brani.begin(); !found && it != brani.end(); ++it)
      if(*(it->ptr) == *p) found = true;
    if(!found) brani.push_back(p);
  }
};

int main() {

}
```

Gang of Four, MVC (Model View Controller), Qt e classi principali (Widget, QObject), Signals and slots, Ereditarietà Multipla, Stream di file e stringhe, Eccezioni

Un accenno più che altro ideale: la GoF (Gang Of Four), così chiamata in quanto è un libro scritto col titolo omonimo da 4 autori diversi, una serie di design pattern suddivisi in 3 principali categorie:

Creational Design Patterns

- Abstract Factory. Consente la creazione di oggetti senza specificarne il tipo concreto.
- Builder. Si usa per creare oggetti complessi.
- Factory Method. Crea oggetti senza specificare la classe esatta da creare.
- Prototype. Crea un nuovo oggetto da un oggetto esistente.
- Singleton. Assicura la creazione di una sola istanza di un oggetto.

Structural Design Patterns

- Adapter. Permette a due classi incompatibili di lavorare insieme avvolgendo un'interfaccia attorno a una delle classi esistenti.
- Bridge. Disaccoppia un'astrazione in modo che due classi possano variare indipendentemente.
- Composite. Trasforma un gruppo di oggetti in un singolo oggetto.
- Decorator. Permette di estendere dinamicamente il comportamento di un oggetto in fase di esecuzione.
- Facade. Fornisce una semplice interfaccia a un oggetto sottostante più complesso.
- Flyweight. Riduce il costo dei modelli di oggetti complessi.
- Proxy. Fornisce un'interfaccia segnaposto a un oggetto sottostante per controllare l'accesso, ridurre i costi o ridurre la complessità.

Behavior Design Patterns

- Chain of Responsibility. Delega i comandi a una catena di oggetti di elaborazione.
- Command. Crea oggetti che incapsulano azioni e parametri.
- Interpreter. Implementa un linguaggio specializzato.
- Iterator. Accede agli elementi di un oggetto in modo sequenziale senza esporre la sua rappresentazione sottostante.
- Mediator. Permette l'accoppiamento libero tra le classi, essendo l'unica classe ad avere una conoscenza dettagliata dei loro metodi.
- Memento. Fornisce la possibilità di ripristinare un oggetto al suo stato precedente.
- Observer. È un modello publish/subscribe che consente a un certo numero di oggetti osservatori di vedere un evento.
- State. Permette a un oggetto di modificare il suo comportamento quando cambia il suo stato interno.
- Strategy. Permette di selezionare al volo, in fase di esecuzione, uno di una famiglia di algoritmi.
- Template Method. Definisce lo scheletro di un algoritmo come classe astratta, consentendo alle sue sottoclassi di fornire un comportamento concreto.
- Visitor. Separa un algoritmo da una struttura di oggetti, spostando la gerarchia dei metodi in un unico oggetto.

Il pattern di progettazione *Model View Controller (MVC)* specifica che un'applicazione consiste in un modello di dati, informazioni di presentazione e informazioni di controllo. Il pattern richiede che ognuno di questi elementi sia separato in oggetti diversi.
MVC è più che altro un modello architetturale, ma non per un'applicazione completa. MVC si riferisce principalmente all'interfaccia utente/livello di interazione di un'applicazione. Avrete comunque bisogno di un livello di logica aziendale, forse di un livello di servizi e di un livello di accesso ai dati.

- Il Modello contiene solo i dati dell'applicazione pura, non contiene la logica che descrive come presentare i dati all'utente.
- La View presenta i dati del modello all'utente. La vista sa come accedere ai dati del modello, ma non sa cosa significhino questi dati o cosa l'utente possa fare per manipolarli.
- Il Controller esiste tra la vista e il modello. Ascolta gli eventi scatenati dalla vista (o da un'altra fonte esterna) ed esegue la reazione appropriata a tali eventi. Nella maggior parte dei casi, la reazione consiste nel chiamare un metodo sul modello. Poiché la vista e il modello sono collegati attraverso un meccanismo di notifica, il risultato di questa azione si riflette automaticamente nella vista.

Vediamo un esempio semplice in codice:

```
class Student
{
   private String rollNo;
   private String name;

   public String getRollNo()
   {
      return rollNo;
   }

   public void setRollNo(String rollNo)
   {
      this.rollNo = rollNo;
   }

   public String getName()
   {
      return name;
   }

   public void setName(String name)
   {
      this.name = name;
   }
}

class StudentView
{
   public void printStudentDetails(String studentName, String studentRollNo)
   {
      System.out.println("Student: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " + studentRollNo);
   }
}
```

Scritto da Gabriel

```
class StudentController
{
   private Student model;
   private StudentView view;

   public StudentController(Student model, StudentView view)
   {
      this.model = model;
      this.view = view;
   }

   public void setStudentName(String name)
   {
      model.setName(name);
   }

   public String getStudentName()
   {
      return model.getName();
   }

   public void setStudentRollNo(String rollNo)
   {
      model.setRollNo(rollNo);
   }

   public String getStudentRollNo()
   {
      return model.getRollNo();
   }

   public void updateView()
   {
      view.printStudentDetails(model.getName(), model.getRollNo());
   }
}

class MVCPattern
{
   public static void main(String[] args)
   {
      Student model  = retriveStudentFromDatabase();

      StudentView view = new StudentView();

      StudentController controller = new StudentController(model, view);

      controller.updateView();

      controller.setStudentName("Vikram Sharma");

      controller.updateView();
```

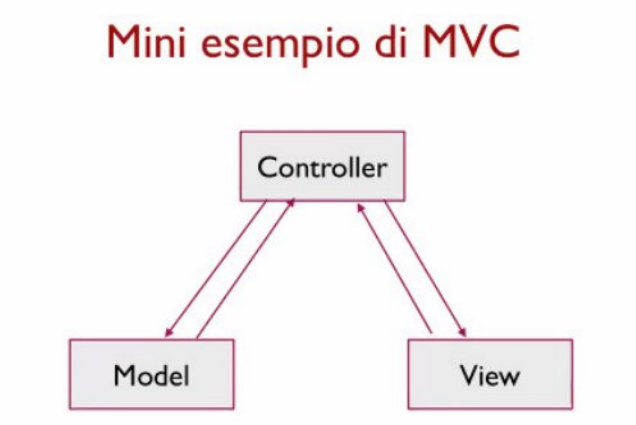
```
    }

    private static Student retriveStudentFromDatabase()
    {
        Student student = new Student();
        student.setName("Lokesh Sharma");
        student.setRollNo("15UCS157");
        return student;
    }

}
```

Vantaggi di MVC: vantaggi principali

- Facilità di manutenzione del codice, facile da estendere e da far crescere.
- Il componente MVC Model può essere testato separatamente dall'utente
- Supporto più semplice per nuovi tipi di client
- Lo sviluppo dei vari componenti può essere eseguito in parallelo.
- Aiuta a evitare la complessità dividendo un'applicazione nelle tre unità. Modello, vista e controllore
- Utilizza solo il pattern Front Controller, che elabora le richieste dell'applicazione web attraverso un singolo controller.
- Offre il miglior supporto per lo sviluppo guidato dai test
- Funziona bene per le applicazioni Web che sono supportate da grandi team di progettisti e sviluppatori Web.
- Fornisce una separazione pulita delle preoccupazioni (SoC).
- È compatibile con l'ottimizzazione dei motori di ricerca (SEO).
- Tutte le classi e gli oggetti sono indipendenti l'uno dall'altro, in modo da poterli testare separatamente.
- Il design pattern MVC consente di raggruppare logicamente le azioni correlate su un controllore.

Svantaggi dell'uso di MVC

- Difficile leggere, modificare, testare unitariamente e riutilizzare questo modello.
- La navigazione del framework può essere a volte complessa, in quanto introduce nuovi livelli di astrazione che richiedono agli utenti di adattarsi ai criteri di decomposizione di MVC.
- Nessun supporto formale per la validazione
- Aumento della complessità e dell'inefficienza dei dati
- La difficoltà di utilizzare MVC con la moderna interfaccia utente
- È necessario l'intervento di più programmatori per condurre una programmazione parallela.
- È richiesta la conoscenza di più tecnologie.
- Manutenzione di molti codici nel controllore



Model

```
class GraphModel {
private:
    int number; // dato logico

public:
    GraphModel(): number(1) {} // costruttore

    void increaseNumber() { number += 10; }  // scrittura/lettura

    int getNumber() const { return number; } // lettura
};
```

View

```
class GraphView {
private:
    Button*              button;      // view components
    GraphController*     controller;  // control

public:
    // costruisce view e control
    GraphView():
         button(new Button("Click Me")),
         controller(new GraphController(this)) {}

    ~GraphView() {
        delete button;
        delete controller;
    }

    // definisce il gestore del button click
    void setClickHandler(ButtonHandler* bh){
        button->setHandler(bh);
    }

    void drawGraph() {
        // ottiene dati logici dal model via control
        int dati = controller->getDataDrawing();
        // Disegna il grafo sui dati
        // ...
    }
};
```

Controller

```
class GraphController {
private:
    GraphModel*     model;  // model
    GraphView*      view;   // view

public:
   GraphController(GraphView* v): model(new GraphModel()), view(v) {
        // installazione del gestore sulla view
        view->setClickHandler(&onButtonClicked);
    }

    // trasmette l'input dalla view al model e modifica il model
    void onButtonClicked() {
        model->increaseNumber();
    }

    // ottiene dati dal model
    int getDataDrawing() const {
        return model->getNumber();
    }
};
```

Qt: Segnali, slot, esempi vari

Il compilatore di Qt è il *Meta-Object Compiler, moc* , è il programma che gestisce le estensioni C++ di Qt. Lo strumento moc legge un file di intestazione C++. Se trova una o più dichiarazioni di classi che contengono la macro Q_OBJECT, produce un file sorgente C++ contenente il codice dei meta-oggetti per quelle classi. Infatti, in Qt tutti gli oggetti ereditano da QObject. Ogni singolo oggetto possiede relazioni di parentela, tale che esiste sempre un oggetto *parent* (genitore), da cui erediti metodi e caratteristiche.

Similmente, QWidget eredita da QObject. Il main di Qt ha questa struttura.

```
// file "main.cpp"

// inclusione di header files
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[]) {
  // una QApplication in ogni Qt GUI
  // argc e argv sono passati al costruttore di QApplication
  QApplication app(argc, argv);
```

Aggiungiamo per esempio un bottone e lo mostriamo:

```
// file "main.cpp"

// inclusione di header files
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[]) {
  // una QApplication in ogni Qt GUI
  // argc e argv sono passati al costruttore di QApplication
  QApplication app(argc, argv);
  // QPushButton è un widget
  // costruzione del QPushButton hello
  // hello non ha parent window, lui stesso è una window con frame e title bar
  QPushButton hello("Hello world!");
  // show() è uno slot, rende visibile un widget, che altrimenti non è visibile
  hello.show();
```

A questo bottone sono associate delle azioni, per esempio al clic o alla pressione:

- i segnali definiscono il comportamento (che metodo viene chiamato) quando viene eseguita una certa azione.
- gli slot, che sono i metodi chiamati dai segnali.

I segnali connettono gli slot. Ogni oggetto definisce che segnali può emettere e ad ogni slot possono essere connessi vari segnali.
Un esempio semplice di invocazione:

```
class MyWidget : public QWidget
{
    Q_OBJECT // moc: macro Q_OBJECT in ogni classe con signal/slot

public:
    MyWidget();

signals:
    void buttonClicked();

private:
    QPushButton *myButton;
};

MyWidget::MyWidget()
{
    myButton = new QPushButton(this);
    connect(myButton, SIGNAL(clicked()),
            this, SIGNAL(buttonClicked()));
}
```

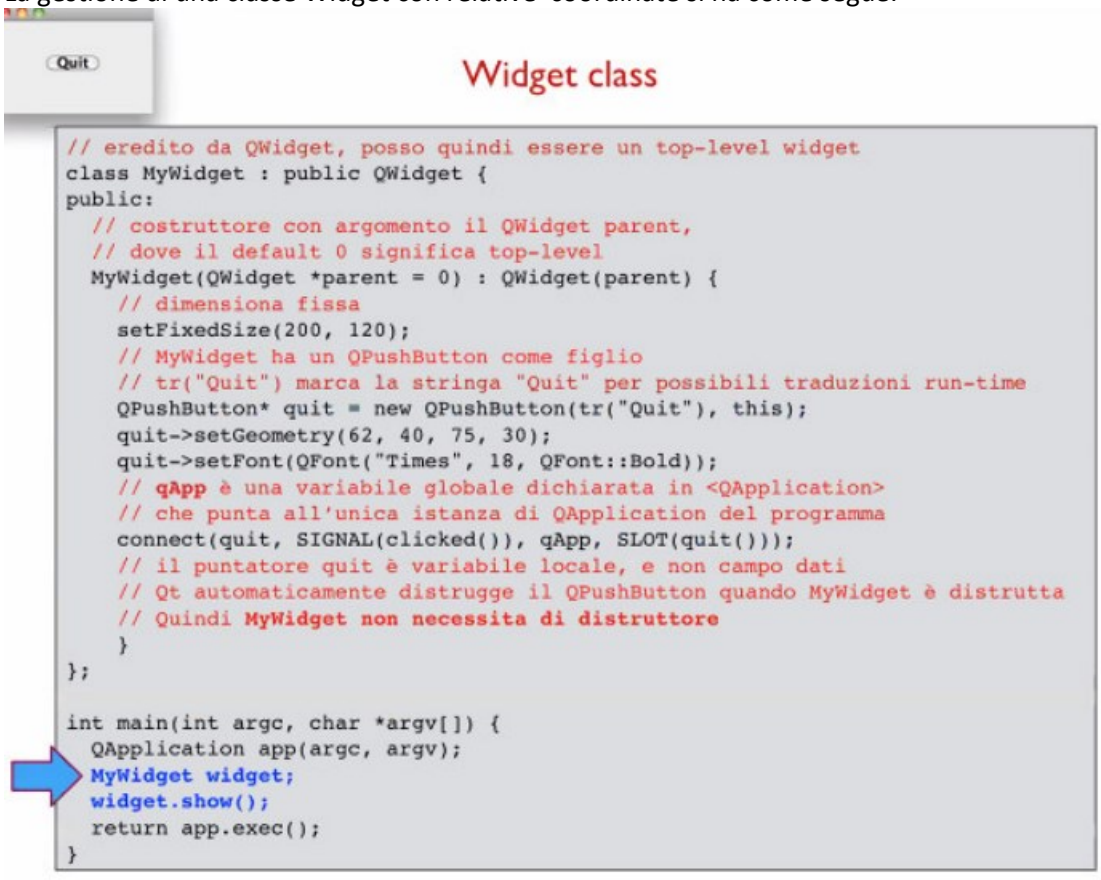
Altro esempio di connessione tra oggetti, con una successiva finestra parent che ridimensione e gestisce bottoni e applicazione:

```
#include <QApplication>
#include <QFont>
#include <QPushButton>

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  // costruisce il pulsante con una label
  QPushButton quit("Quit");
  // ridimensiona il pulsante quit
  quit.resize(75, 30);
  // setta la font del pulsante quit
  quit.setFont(QFont("Times", 18, QFont::Bold));
  // invocazione di connect(), metodo statico di QObject
  // stabilisce una connessione tra due QObject
  // Ogni QObject (e quindi ogni QWidget) può avere signal (mandare messaggi)
  // e slot (ricevere messaggi)
  // Il segnale clicked di quit è connesso allo slot quit() di app:
  // quando si clicka il pulsante quit l'applicazione app termina
  QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
  quit.show();
  return QApplication::exec();
}
```

Parent window

```
#include <QApplication>
#include <QFont>
#include <QPushButton>
#include <QWidget>

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  // QWidget è la classe base di tutti i widget
  // Un QWidget è un atomo di una GUI: riceve eventi dal sistema (mouse,
  // keyboard, etc), e rappresenta sè stessa sullo schermo.
  // Una QWidget è detenuta dal suo parent
  // Una QWidget senza parent è detta una independent window (con frame e
  // taskbar). La posizione iniziale è controllata dal sistema
  QWidget window;
  // setta il titolo
  window.setWindowTitle("I'm a QWidget");
  // ridimensionamento di window
  window.resize(200, 120);
  // quit ha come parent window, ovvero quit è figlio di window
  // Un figlio è sempre mostrato nell'area del suo parent, per default
  // al top-left corner alla posizione (0,0)
  QPushButton quit("Quit", &window);
  quit.setFont(QFont("Times", 18, QFont::Bold));
  // QWidget::setGeometry(x,y,w,h):
  // (x,y) coordinate del top-left corner in pixel
  // (w,h) base ed altezza in pixel
  quit.setGeometry(10, 40, 180, 40);
  QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
  // la chiamata di show() su window chiama show() anche su tutti i figli
  window.show();
  return app.exec();
}
```

La gestione di una classe Widget con relative coordinate si ha come segue:




```
// eredito da QWidget, posso quindi essere un top-level widget
class MyWidget : public QWidget {
public:
  // costruttore con argomento il QWidget parent,
  // dove il default 0 significa top-level
  MyWidget(QWidget *parent = 0) : QWidget(parent) {
    // dimensiona fissa
    setFixedSize(200, 120);
    // MyWidget ha un QPushButton come figlio
    // tr("Quit") marca la stringa "Quit" per possibili traduzioni run-time
    QPushButton* quit = new QPushButton(tr("Quit"), this);
    quit->setGeometry(62, 40, 75, 30);
    quit->setFont(QFont("Times", 18, QFont::Bold));
    // qApp è una variabile globale dichiarata in <QApplication>
    // che punta all'unica istanza di QApplication del programma
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    // il puntatore quit è variabile locale, e non campo dati
    // Qt automaticamente distrugge il QPushButton quando MyWidget è distrutta
    // Quindi MyWidget non necessita di distruttore
    }
};

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  MyWidget widget;
  widget.show();
  return app.exec();
}
```

La gestione del layout, quindi la grafica che è in grado di gestire (verticalmente, orizzontalmente, a griglia) che con tengono widget e possono impostare un'apposita gestione grafica.

```
class MyWidget : public QWidget {
public:
MyWidget(QWidget *parent = 0) : QWidget(parent) {
  QPushButton *quit = new QPushButton(tr("Quit"));
  quit->setFont(QFont("Times", 18, QFont::Bold));
  // QLCDNumber è un widget che mostra numeri come un LCD
  QLCDNumber *lcd = new QLCDNumber(2); // 2 cifre
  // rende lcd più leggibile
  lcd->setSegmentStyle(QLCDNumber::Filled);
  // Il widget QSlider permette di selezionare un valore intero in un range
  // slider è un QSlider orizzontale con valori in [0,99], e valore iniziale 0
  QSlider *slider = new QSlider(Qt::Horizontal);
  slider->setRange(0, 99); slider->setValue(0);
  connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
  // segnale valueChanged(int) di slider connesso allo slot display(int) di lcd
  // slider emette il segnale valueChanged() quando cambia il suo valore
  // lo slot è quindi chiamato quando viene emesso questo segnale
  connect(slider, SIGNAL(valueChanged(int)),lcd, SLOT(display(int)));
  // QVBoxLayout è un layout manager (LM) che gestisce la geometria di piazzamento
  // dei figli di un widget; inoltre il LM di un widget w gestisce il
  // ridimensionamento dei figli di w quando viene ridimensionato w
  // QLayout non è sottotipo di QWidget, quindi un LM non ha mai un parent
  QVBoxLayout *layout = new QVBoxLayout;
  // layout->addWidget() aggiunge dei widget al LM layout
  layout->addWidget(quit); layout->addWidget(lcd); layout->addWidget(slider);
  // QWidget::setLayout(layout) installa layout come LM di this
  // inoltre rende layout un figlio di this, e tutti i widget in layout
  // diventano figli di this
  setLayout(layout);
  }
};
```

Come detto, gli slot sono metodi ordinari connessi agli oggetti tramite la macro privata QObject::Q_OBJECT, gestito automaticamente da *qmake*. Un segnale si *dichiara*, non si definisce e *non va implementato*.

- Aggiungere sempre alla parte privata di una classe con segnali propri la macro `QObject::Q_OBJECT`. Serve al MOC
- Un segnale si **dichiara** (non si definisce) come un metodo ordinario preceduto dalla Qt keyword `signals`

```
signals:
        void valueChanged(int newValue);
```

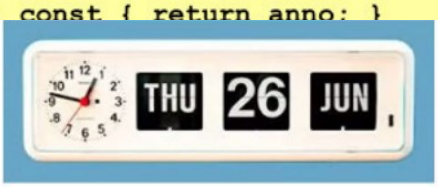
- Il segnale **non va implementato**
- Per emettere il segnale si usa la Qt keyword `emit`:

```
emit valueChanged(12);
```

Ereditarietà multipla, ereditarietà a diamante e gestione virtual

L'ereditarietà multipla è il concetto di ereditarietà in C++ che consente a una classe figlia di ereditare proprietà o comportamenti da più classi base. Pertanto, possiamo dire che è il processo che consente a una classe derivata di acquisire funzioni membro, proprietà e caratteristiche da più di una classe base. Riprendiamo ora l'esempio di *dataora*, che ora erediterà contemporaneamente da *data* (qui riportata) e da *orario*.

```
class data {
private: // metodi di utilità
  int GiorniDelMese() const;
  bool Bisestile() const;
protected:
  int giorno, mese, anno;
  void AvanzaUnGiorno(); // utilità per la gerarchia
public:
  data(int =1, int =1, int =0);
  int Giorno() const { return giorno; }
  int Mese() const { return mese; }
  int Anno() const { return anno; }
```




```
#include "orario.h"
#include "data.h"

class dataora : public data, public orario {
public:
  dataora() {}
  dataora(int a, int me, int g, int o, int m, int s)
          : data(a,me,g), orario(o,m,s) {}
  dataora operator+(const orario&) const;
  bool operator==(const dataora&) const;
  ...
};
```

Supponiamo di avere un metodo *Stampa()* sia in *orario* che in *data*: dataora erediterà due metodi diversi con stesso nome e segnatura! Ciò porta ad ambiguità.

```
void orario::Stampa() const {
  cout << Ore() << ':' << Minuti() << ':' << Secondi();
}
```

```
void data::Stampa() const {
  cout << Giorno() << '/' << Mese() << '/' << Anno();
}
```

```
dataora d;
d.Stampa();   // ILLEGALE
```

Come al solito per gli errori di compilazione dovuti da ambiguità, la generazione dell'errore avviene **soltanto quando si tenta di invocare** tale metodo `Stampa()`

L'ambiguità rimarrebbe anche se le segnature dei metodi nelle classi base fossero diverse

```
class A {
public:
  void f() {cout << "A::f ";}
};

class B {
public:
  void f(int x) {cout << "A::f ";}
};

class D: public A, public B {};

int main(){
  D d;
  d.f();  // Illegale: "request for member f is ambiguous"
  d.f(2); // Illegale: "request for member f is ambiguous"
}
```

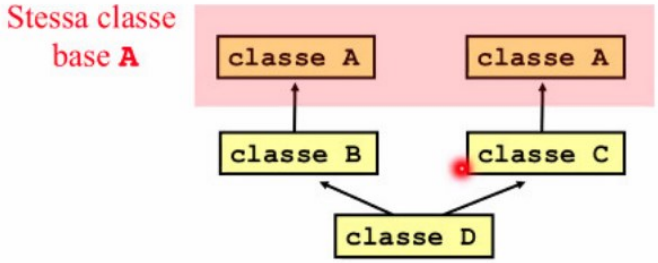
Possiamo risolvere l'ambiguità usando l'operatore di scoping:

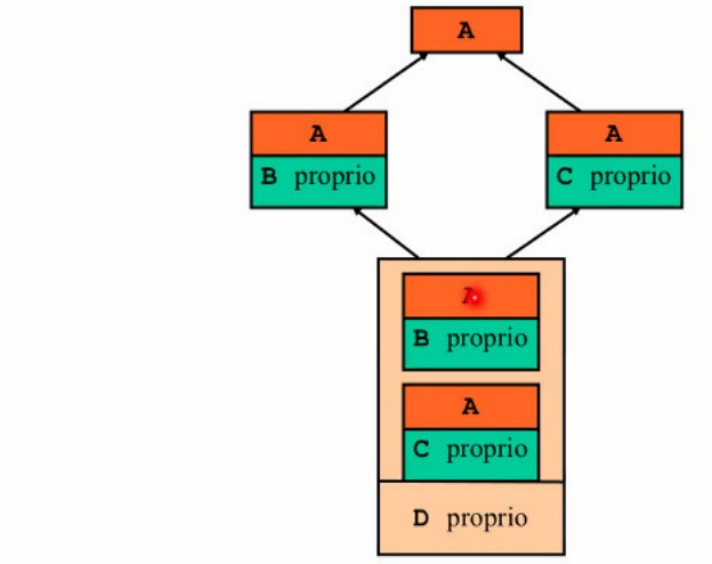
```
dataora d;
d.data::Stampa();
// OPPURE
dataora d;
d.orario::Stampa();
```

Una ridefinizione del metodo `Stampa()` nella classe `dataora` avrebbe nascosto entrambi i metodi `Stampa()` delle classi base e non ci sarebbe quindi stata alcuna ambiguità.

```
void dataora::Stampa() const {
  data::Stampa(); cout << ' '; orario::Stampa();
}
```

Il problema più grande con l'ereditarietà multipla è *l'ereditarietà a diamante*. Il problema del diamante si verifica quando una classe figlio eredita da due classi genitore che condividono una classe nonno comune.





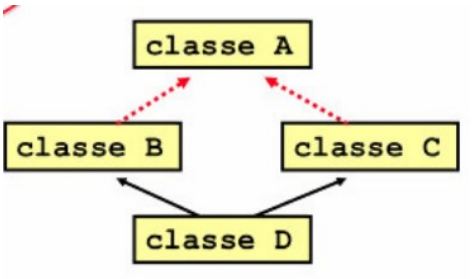
Due sottooggetti della classe base comune **A**:
1) ambiguità
2) o come minimo spreco di memoria

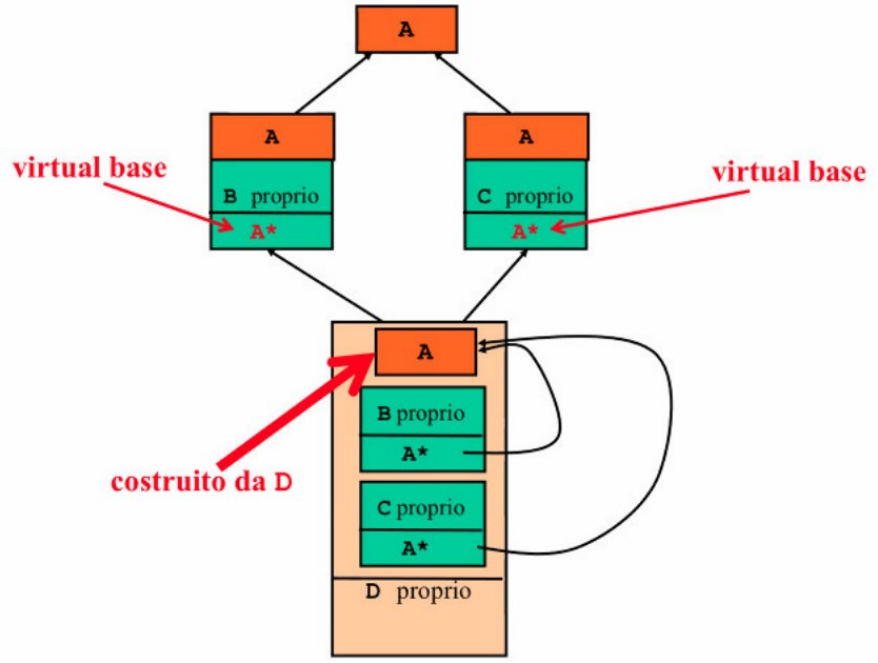
```
class A {
public:
  int a;
  A(int x=1): a(x) {}
};

class B: public A {
public:
  B(): A(2) {}
};

class C: public A {
public:
  C(): A(3) {}
};

class D: public B, public C { };

int main(){
  D d;
  A* p = &d;     // Illegale: A è una classe base ambigua per D
  cout << p->a; // Quale sottooggetto di A si dovrebbe usare?
}
```

```
class A {
protected:
  int x;
public:
  A(int y=0): x(y) {}
  virtual void print() =0; // virtuale puro
};
class B: public A {
public:
  B(): A(1) {}
  virtual void print() {cout << x;} // implementazione in B
};
class C: public A {
public:
  C(): A(2) {}
  virtual void print() {cout << x;} // implementazione in C
};

class D: public B, public C {
public:
  virtual void print() {cout << x;} // overriding: quale x?
};

int main(){
  D d;
  A* p = &d;  // ERRORE: "A is an ambiguous base of D"
  p->print(); // la chiamata polimorfa non è legale
}
```

Per risolvere l'ereditarietà a diamante, usiamo la *derivazione virtuale*. Basterà mettere la parola *virtual* davanti alla keyword modificatrice di accesso.

```
class A { // A è una classe base virtuale
...
};

class B : virtual public A {
...
};

class C : virtual public A {
...
};

class D : public B, public C {
...
};
```





Ulteriori esempi:

```
class A {
  double d; // 8 byte
};

class B: virtual public A {
  // 8 byte + 1 puntatore (8 byte)
};

class C : virtual public A {
  // 8 byte + 1 puntatore (8 byte)
};

class D: public B, public C {
};

int main() {
  cout << "sizeof(A) == " << sizeof(A) << endl; // 8
  cout << "sizeof(B) == " << sizeof(B) << endl; // 16=8+8
  cout << "sizeof(C) == " << sizeof(C) << endl; // 16=8+8
  cout << "sizeof(D) == " << sizeof(D) << endl; // 24=8+8+8
}
```

Attenzione: rappresentazione non standard, i compilatori possono implementare in modo diverso

```
class A {
public:
  int a;
  A(int x=1): a(x) {}
};

class B: virtual public A {
public:
  B(): A(2) {}
};

class C: virtual public A {
public:
  C(): A(3) {}
};

class D: public B, public C {
};

int main(){
  D d;
  A* p = &d;     // compila
  cout << p->a; // stampa: 1 (e non 2 o 3!), perchè?
}
```

La stampa si ha in questo modo dovuto al binding statico di b, in quanto le sottoclassi si riferiscono virtualmente ad A e la variabile a viene inizializzata una volta sola.

Un problema interessante: unique final overrider. Innanzitutto, parliamo della keyword *override*.
La parola chiave override ha due scopi:

1) Mostra al lettore del codice che "questo è un metodo virtuale, che sta sovrascrivendo un metodo virtuale della classe base".
2) **2)** Il compilatore sa anche che si tratta di un override; quindi, può "controllare" che non si stiano modificando/aggiungendo nuovi metodi che si pensa siano override.

In poche parole, il problema che si ha nell'esempio che segue è questo:
tutti i metodi ereditano e ridefiniscono virtualmente. Se l'ultima classe in linea ereditaria non ridefinisce il metodo, non si sa quale classe allo stesso livello di gerarchia chiama quel metodo. Da cui l'errore.

Seguono un insieme di esempi di questi, i tipi di ereditarietà e ulteriori esempi di codice.

```
class A {
public:
  virtual void print()=0;
};

class B: virtual public A {
public:
  void print() override {cout << "B ";}
};

class C: virtual public A {
public:
  void print() override {cout << "C ";}
};

class D: public B, public C {
  // se ometto questo overriding si ottiene un errore di compilazione:
  // "no unique final overrider for A::print()"
  void print() override {cout << "D ";}
};

int main(){
  D d;
  A* p = &d;  // compila
  p->print(); // stampa: D
}
```

MOTIVAZIONE
La vtable di **D** deve avere
un indirizzo per l'entry di **print()**

```
class A {
public:
  virtual void print() {cout << "A ";}
};

class B: virtual public A {
public:
  // void print() override {cout << "B ";}
};

class C: virtual public A {
public:
  // void print() override {cout << "C ";}
};

class D: public B, public C {

  // in questo caso è legale, ereditiamo un "unique final overrider"
  // void print() override {cout << "D ";}
};

int main(){
  D d;
  A* p = &d;  // compila
  p->print(); // stampa: A
}
```

```
class A {
public:
  void print() {cout <<"A ";}
};

class B: virtual public A {
public:
  void print() {cout << "B ";}
};

class C: virtual public A {
public:
  void print() {cout << "C ";}
};

class D: public B, public C {
  // eredito 2 metodi print() non virtuali,
  // compila ma ambiguità in compilazione per una invocazione d.print()
};

int main(){
   D d;
   A* p = &d;     // compila
   p->print();    // stampa: A
   d.print();     // Illegale: chiamata ambigua
   d.B::print(); // OK, stampa B
}
```

```
class A {
public:
  void print() {cout <<"A ";}
};

class B: virtual public A {
public:
  void print() {cout << "B ";}
};

class C: virtual public A {
public:
  void print() {cout << "C ";}
};

class D: public B, public C {
  // eredito 2 metodi print() non virtuali,
  // compila ma ambiguità per una invocazione d.print()
};

// MOTIVAZIONE
// D non ha vtable, oppure non ha una entry per print() nella sua vtable
```

Anche per la derivazione virtuale multipla, possiamo avere derivazione privata, pubblica o protetta.

Vale la seguente **regola**: la derivazione protetta prevale su quella privata, e la derivazione pubblica prevale su quella protetta.

```
class A {
public:
  void f() {cout << "A";}
};
class B: virtual private A {}; // derivazione virtuale privata

class C: virtual public A {};  // derivazione virtuale pubblica

class D: public B, public C {}; // prevale la derivazione pubblica

int main(){
  D d;
  d.f(); // OK, stampa: A (sarebbe C::f())
  d.B::f(); // Illegale, non compila, A::f() inaccessibile
}
```

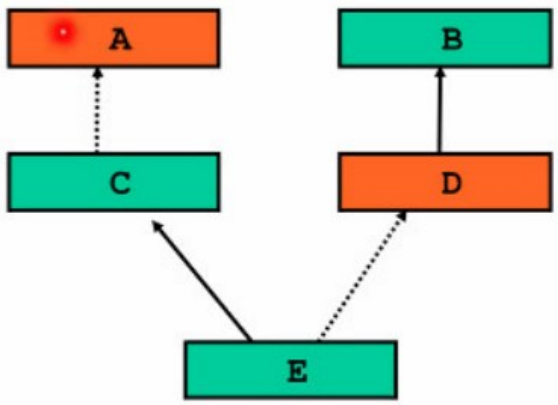
Costruttore di D in presenza di basi virtuali

(1) Per primi vengono richiamati, una sola volta, i costruttori delle **classi base virtuali** che si trovano nella gerarchia di derivazione di D. Vi può essere più di una classe base virtuale nella gerarchia di derivazione di D: la ricerca delle classi base virtuali nella gerarchia procede seguendo l'ordine da sinistra verso destra e dall'alto verso il basso (``left-to-right top-down order'').

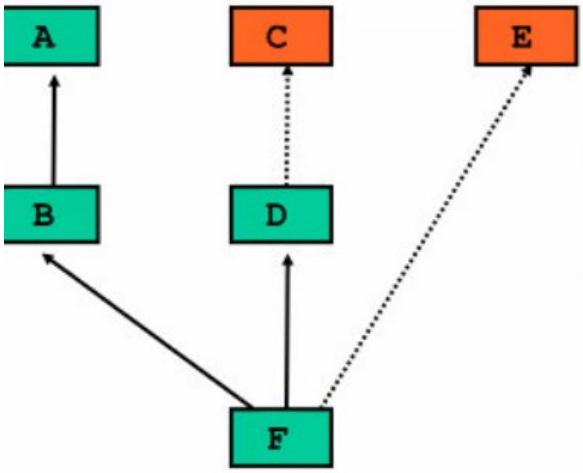
(2) Una volta che sono stati invocati i costruttori delle classi virtuali nella gerarchia di derivazione di D, vengono richiamati i costruttori delle superclassi dirette non virtuali di D: questi costruttori **escludono di richiamare** eventuali costruttori di classi virtuali già richiamati al passo (1);

(3) Infine viene eseguito il costruttore "proprio" di D, ovvero vengono costruiti i campi dati propri di D e quindi viene eseguito il corpo del costruttore di D.

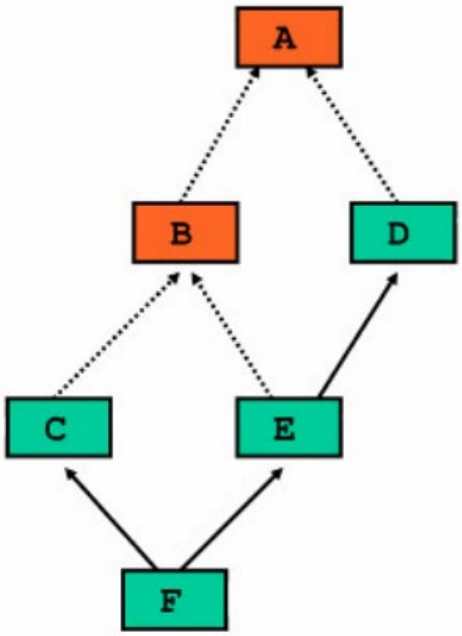
Le chiamate dei costruttori dei punti (1) e (2), **se non sono esplicite, vengono automaticamente inserite** dal compilatore nella lista di inizializzazione del costruttore di D: in questo caso, come al solito, si tratta di chiamate implicite ai costruttori di default.




```
class A {
public:
  A() { cout << "A ";}
};

class B {
public:
  B() { cout << "B ";}
};

class C: virtual public A {
public:
  C(){ cout << "C ";}
};

class D: public B {
public:
  D(){  cout << "D ";}
};

class E : public C, virtual public D {
public:
  E(){  cout << "E ";}
};

int main() {     E e; }
  // stampa: A B D C E
```



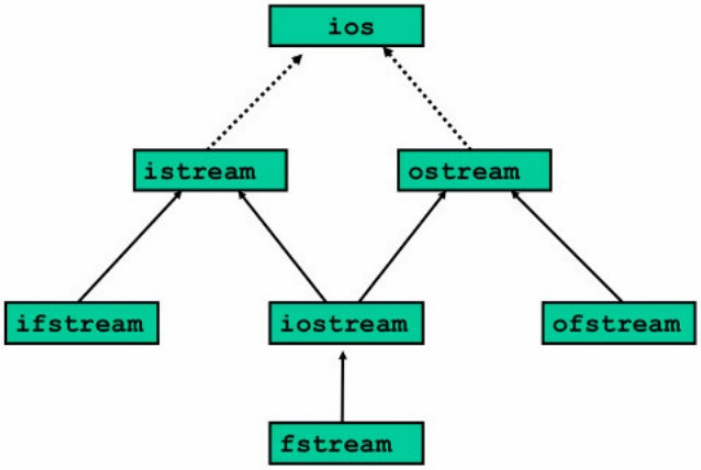

```
class A {
public:
  A() {cout << "A ";}
};
class B: public A {
public:
  B() {cout << "B ";}
};
class C {
public:
  C() {cout << "C ";}
};
class D: virtual public C {
public:
  D() {cout << "D ";}
};
class E {
public:
  E() {cout << "E ";}
};
class F:
    public B, public D, virtual public E
{
public:
  F() {cout << "F ";}
};
int main(){ F f; }
// stampa: C E A B D F
```




```
class A {
public:
  A() {cout << "A ";}
};
class B: virtual public A {
public:
  B() {cout << "B ";}
};
class D: virtual public A {
public:
  D() {cout << "D ";}
};
class C: virtual public B {
public:
  C() {cout << "C ";}
};
class E: virtual public B, public D {
public:
  E() {cout << "E ";}
};
class F: public C, public E {
public:
  F() {cout << "F ";}
};
int main(){ F f; }
// stampa: A B C D E F
```

Gerarchia delle classi input/output, stream, errori ed eccezioni

La classe ios è la classe più alta nella gerarchia delle classi di stream. È la classe base per le classi istream, ostream e streambuf. La classe ios è responsabile di fornire tutte le strutture di input e output a tutte le altre classi di stream. La classe ios è ereditata indirettamente dalla classe iostream tramite istream e ostream.



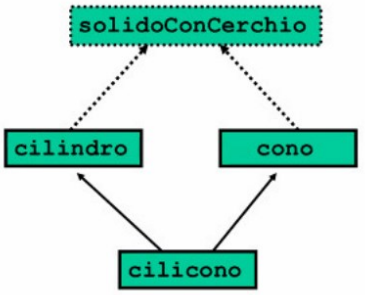
istream e ostream sono le classi base della classe iostream (derivano virtualmente da ios). La classe istream viene utilizzata per l'input e ostream per l'output. Questa classe è responsabile della gestione del flusso di input. Fornisce una serie di funzioni per la gestione di caratteri, stringhe e oggetti, come get, getline, read, ignore, putback ecc.

La classe ostream: Questa classe è responsabile della gestione dei flussi di uscita. Fornisce una serie di funzioni per la gestione di caratteri, stringhe e oggetti, come write, put ecc.

La classe iostream: Questa classe è responsabile della gestione di entrambi i flussi di input e output, in quanto sia la classe istream che la classe ostream sono ereditate in essa. Fornisce le funzioni della classe istream e della classe ostream per gestire caratteri, stringhe e oggetti, come get, getline, read, ignore, putback, put, write ecc.

Classe istream_withassign: Questa classe è una variante di istream che consente l'assegnazione di oggetti. L'oggetto predefinito cin è un oggetto di questa classe e può quindi essere riassegnato in fase di esecuzione a un altro oggetto istream.

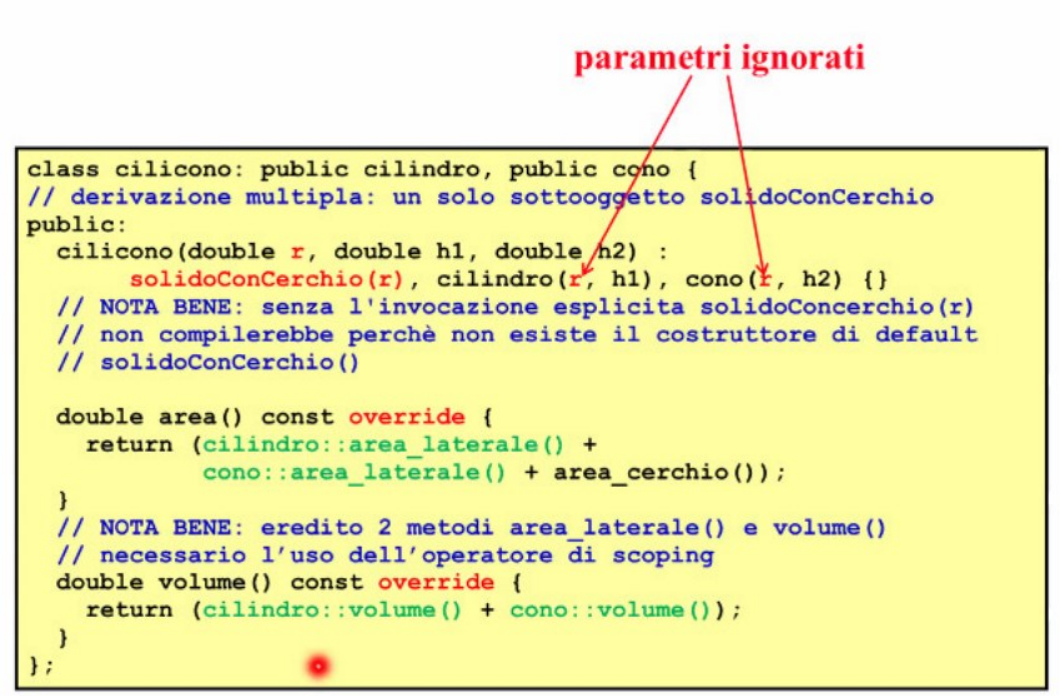
Esempio di una gerarchia → Missile/Cilicono (altro evergreen)




```
#include <cmath>

// gli oggetti sono solidi che hanno un cerchio come faccia

class solidoConCerchio { // classe base astratta virtuale
protected:
 double raggio; // del cerchio
 double circonferenza() const {return (2*M_PI*raggio);}
 double area_cerchio() const {return (M_PI*raggio*raggio);}
public:
 // no costruttore di default
 solidoConCerchio(double r): raggio(r) {}
 // virtuale pura: area del solido che ha un cerchio come faccia
 virtual double area() const = 0;
 // virtuale pura: volume del solido che ha un cerchio come faccia
 virtual double volume() const = 0;
};
```

```
// derivazione virtuale
class cilindro: virtual public solidoConCerchio {
protected:
  double altezza; // altezza del cilindro
  double area_laterale() const {
    return (circonferenza()*altezza);
  }
public:
  cilindro(double r, double h): solidoConCerchio(r), altezza(h) {}
  double area() const override {
    return (2*area_cerchio() + area_laterale());
  }
  double volume() const override {
    return (area_cerchio()*altezza);
  }
};
```

```
// derivazione virtuale
class cono: virtual public solidoConCerchio {
protected:
  double altezza; // altezza del cono
  double area_laterale() const {
    double apotema = sqrt(raggio*raggio + altezza*altezza);
    return (2*circonferenza()*apotema);
  }
public:
  cono(double r, double h): solidoConCerchio(r), altezza(h) {}
  double area() const override {
    return (area_cerchio() + area_laterale());
  }
  double volume() const override {
    return (area_cerchio()*altezza/3);
  }
};
```

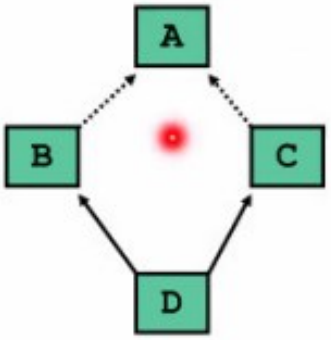


Alcuni esercizi:

```
class Z {
public:
  Z() {cout << "Z() ";}
  Z(const Z& x) {cout << "Zc ";}
};

class A {
private:
  Z w;
public:
  A() {cout << "A() ";}
  A(const A& x) {cout << "Ac ";}
};

class B: virtual public A {
private:
  Z z;
public:
  B() {cout << "B() ";}
  B(const B& x) {cout << "Bc ";}
};

class C: virtual public A {
private:
  Z z;
public:
  C() {cout << "C() ";}
};

class D: public B, public C {
public:
  D() {cout << "D() ";}
  D(const D& x): C(x) {cout << "Dc ";}
};
```

Cosa stampa?

```
D d1;
D d2 = d1;
```




```
class Z {
public:
  Z() {cout << "Z() ";}
  Z(const Z& x) {cout << "Zc ";}
};

class A {
private:
  Z w;
public:
  A() {cout << "A() ";}
  A(const A& x) {cout << "Ac ";}
};

class B: virtual public A {
private:
  Z z;
public:
  B() {cout << "B() ";}
  B(const B& x) {cout << "Bc ";}
};

class C: virtual public A {
private:
  Z z;
public:
  C() {cout << "C() ";}
};

class D: public B, public C {
public:
  D() {cout << "D() ";}
  D(const D& x): C(x) {cout << "Dc ";}
};
```

Cosa stampa?

```
D d1;
Z() A() Z() B() Z() C() D()
```

```
class Z {
public:
  Z() {cout << "Z() ";}
  Z(const Z& x) {cout << "Zc ";}
};

class A {
private:
  Z W;
public:
  A() {cout << "A() ";}
  A(const A& x) {cout << "Ac ";}
};

class B: virtual public A {
private:
  Z z;
public:
  B() {cout << "B() ";}
  B(const B& x) {cout << "Bc ";}
};

class C: virtual public A {
private:
  Z z;
public:
  C() {cout << "C() ";}
};

class D: public B, public C {
public:
  D() {cout << "D() ";}
  D(const D& x): C(x) {cout << "Dc ";}
};
```

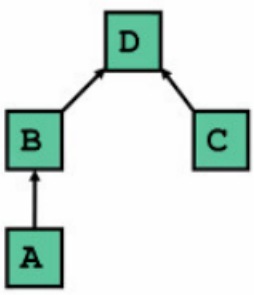
Cosa stampa?

```
D d2=d1;
Z() A() Z() B() Zc Dc
```

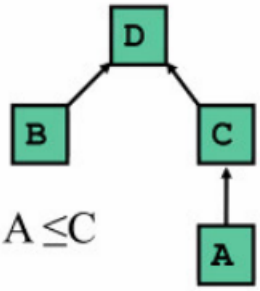
Si assuma che `A`, `B`, `C`, `D` siano quattro classi polimorfe. Si consideri il seguente `main()`.

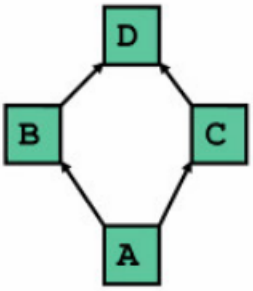
```
main() {
 A a; B b; C c; D d;
 cout << (dynamic_cast<D*>(&c) ? "0 " : "1 ");
 cout << (dynamic_cast<B*>(&c) ? "2 " : "3 ");
 cout << (!(dynamic_cast<C*>(&b)) ? "4 " : "5 ");
 cout << (dynamic_cast<B*>(&a) || dynamic_cast<C*>(&a)  ? "6 " : "7 ");
 cout << (dynamic_cast<D*>(&b) ? "8 " : "9 ");
}
```

Si supponga che tale `main()` compili ed esegua correttamente. Disegnare i diagrammi di **tutte** le possibili gerarchie per le classi `A`, `B`, `C`, `D` tali che l'esecuzione del `main()` provochi la stampa: `0 3 4 6 8`.



$C \leq D$
$C \not\leq B$
$B \not\leq C$
$A \leq B$ or $A \leq C$
$B \leq D$





Abbiamo parlato di classe ios; essa utilizza un'astrazione dei dispositivi di I/O detta *stream*, cosiddetta "sequenza non limitata di celle ciascuna contenente un byte".

- La posizione delle celle di uno stream parte da 0
- I/O effettivo avviene tramite un buffer associato allo stream
- Uno stream può trovarsi nello stato di end-of-file

-

Uno stream può trovarsi in 8 (=2^3) stati di funzionamento diversi. Lo stato è un intero in [0,7] rappresentato dal campo dati **state** della classe base **ios** che corrisponde al numero binario

bad fail eof

dove bad, fail ed eof sono dei bit (0 o 1) di stato:

eof==1 ⇔ lo stream è nella **posizione di end-of-file**.

fail==1 ⇔ la precedente operazione sullo stream è fallita: si tratta di un **errore senza perdita di dati**, normalmente è possibile continuare. Ad esempio, ci si aspettava in input un **int** e si trova invece un **double**.

bad==1 ⇔ la precedente operazione sullo stream è fallita con perdita dei dati: è un **errore fatale/fisico**, normalmente non è possibile continuare. Ad esempio, cerco di accedere ad un file o ad una network connection inesistenti

La classe ios

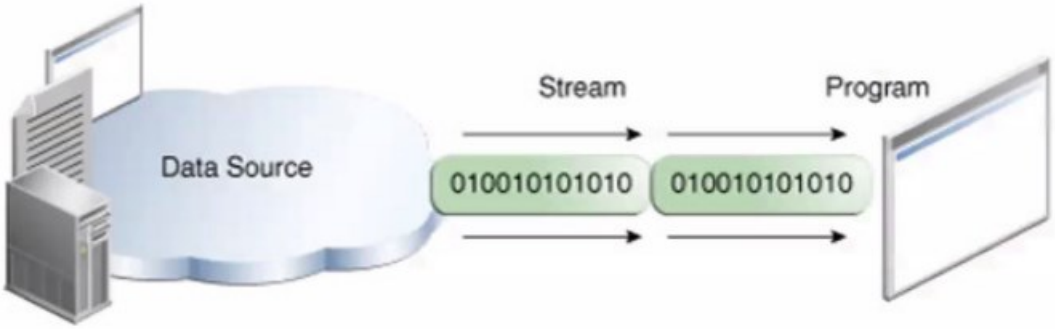
ios (derivata da **ios_base**) è la **classe base astratta "virtuale"** della gerarchia che permette di controllare lo stato di funzionamento di uno stream. Per quanto concerne lo stato di uno stream, la dichiarazione della classe **ios** è la seguente:

```
class ios: public ios_base {
  int state;   // stato dello stream
public:        //      000       001        010       100
  enum io_state {goodbit=0, eofbit=1, failbit=2, badbit=4};
  int good() const;     // ritorna 1 se lo stato è good
  int eof() const;      // ritorna il bit eof
  int fail() const;     // ritorna il bit fail
  int bad() const;      // ritorna il bit bad
  int rdstate() const; // ritorna lo stato come int in [0,7]
  void clear(int i=0); // setta lo stato i
...
};
```

badbit failbit eofbit

La classe istream

Gli oggetti della sottoclasse **istream** rappresentano **stream di input**. **cin** è un oggetto di **istream** che rappresenta lo standard input.



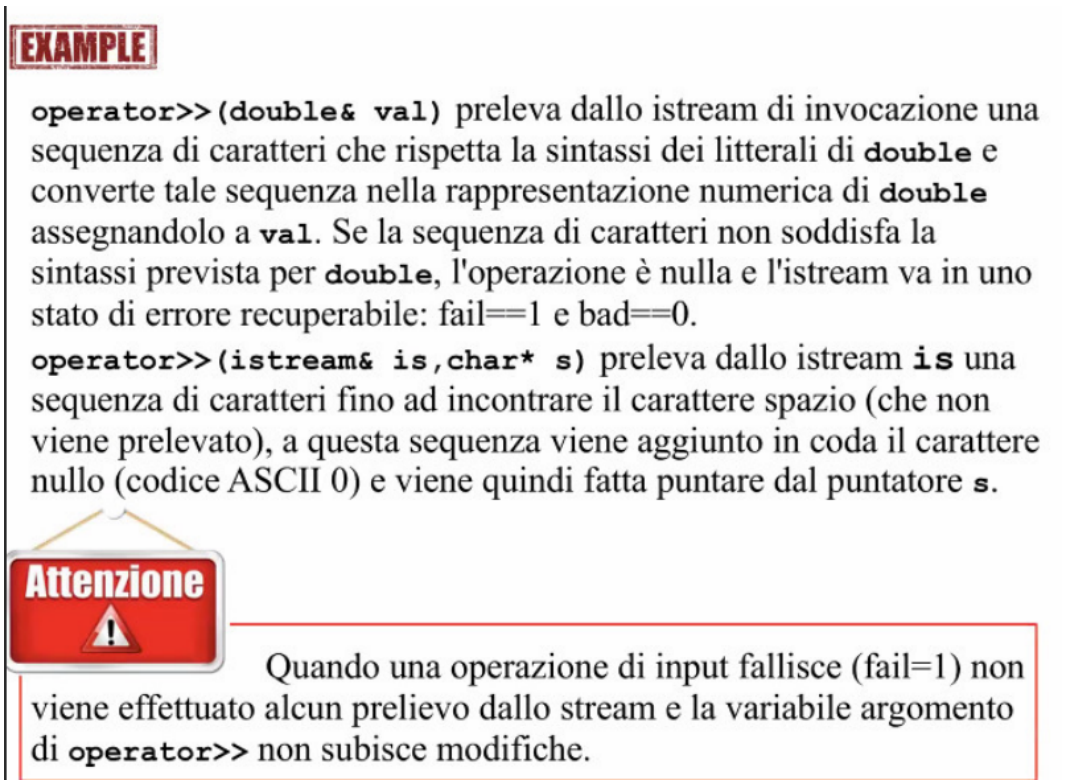
istream include l'overloading dell'operatore di input **operator>>** per i tipi primitivi e per gli array di caratteri.

```
class istream: public virtual ios {
public:
  // metodi interni (con istream di invocazione)
  istream& operator>>(bool&);
  istream& operator>>(int&);
  istream& operator>>(double&);
  ...
};
// funzioni esterne in std::
istream& std::operator>>(istream&, char&); // byte
istream& std::operator>>(istream&, char*); // stringhe
```

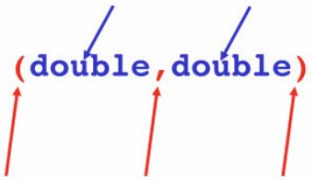
Tutti gli operatori di input **ignorano le spaziature** (cioè spazi, tab, enter) presenti prima del valore da prelevare.

Quando una **operazione di input fallisce (fail==1) non viene effettuato alcun prelievo** dallo stream e la variabile argomento di **operator>>** non subisce modifiche.

Definire un overloading di operator >> per qualche classe C significa dare un significato alla conversione, quindi eseguire il *parsing* di una sequenza di byte di input secondo le regole sintattiche del linguaggio. Vediamo esempi di operatori ed un esempio di parsing:



Parsing di un oggetto punto nel piano reale rappresentato testualmente in forma cartesiana come



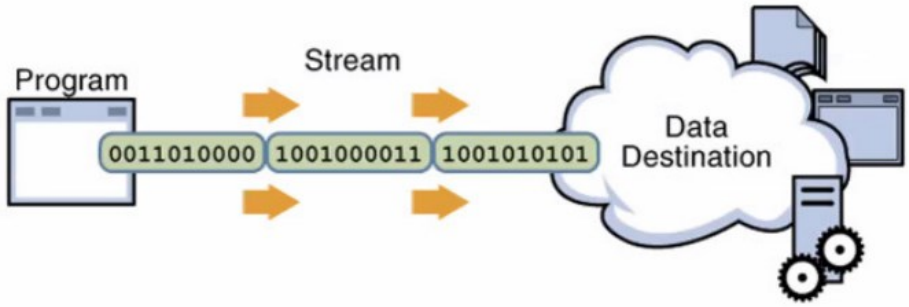

```
class Punto {
  friend istream& operator>>(istream&, Punto&);
  // legge nel formato (x1,x2): rappresentazione testuale di Punto
private:
  double x, y;
};

// ALGORITMO DI PARSING
istream& operator>>(istream& in, Punto& p) {
  char cc; in >> cc; // std::operator>>(istream&,char&)
  if (cc=='q') return in; // carattere q per uscire
  if (cc != '(') { in.clear(ios::failbit); return in;}
  else {
    in >> p.x;        // istream::operator>>(double&)
    if(!in.good()) { in.clear(ios::failbit); return in;}
    in >> cc;
    if (cc != ',') { in.clear(ios::failbit); return in;}
    else {
      in >> p.y;      // istream::operator>>(double&)
      if(!in.good()) { in.clear(ios::failbit); return in;}
      in >> cc;
      if (cc != ')') { in.clear(ios::failbit); return in;}
    }
  }
  return in;
}
```

Scritto da Gabriel

```
#include "Punto.h"
using std::cin; using std::cout;

int main() {
 Punto p;
 cout <<"Inserisci un punto nel formato (x,y) ['q' per uscire]\n";
 while(cin.good()) { // while(stato == 0)
  cin >> p;
  if(cin.fail()) {
    cout << "Input non valido, ripetere!\n";
    cin.clear(ios::goodbit);
    char c=0;
    // 10 è il codice ASCII del carattere newline
    while(c!=10) { cin.get(c); } // svuota cin, get() per input binario
    cin.clear(ios::goodbit);
  }
  else cin.clear(ios::eofbit); // stato 1
 }
}
```

La classe ostream

Gli oggetti della sottoclasse ostream rappresentano **stream di output.** `cout` e `cerr` sono oggetti di ostream (standard output ed error).



La classe ostream include l'overloading dell'operatore di output `operator<<` per i tipi primitivi e per gli array di caratteri costanti.

```
class ostream: public virtual ios {
public:
  // metodi interni (con ostream di invocazione)
  ostream& operator<<(bool);
  ostream& operator<<(int);
  ostream& operator<<(double);
  ...
};
// funzioni esterne
ostream& std::operator<<(ostream&,char);        // byte
ostream& std::operator<<(ostream&,const char*); // stringhe
```

Questi operatori convertono valori di tipo primitivo in sequenze di caratteri che vengono scritti (immessi) nelle celle dell'ostream di invocazione. Per quanto riguarda l'output di stringhe, i caratteri della stringa vengono scritti nell'ostream fino al carattere nullo escluso.

I/O testuale e binario

L'input/output sugli stream tramite gli operatori di input/output >> e << considerano gli stream nel cosiddetto ***formato testo***.

```
T x;
cin >> x;
/* conversione dalla rappresentazione testuale di T
   alla rappresentazione binaria in memoria di T */
cout << x;
/* conversione dalla rappresentazione binaria in
memoria di T alla rappresentazione testuale di T */
```

Quindi, l'informazione da leggere o scrivere su uno stream deve avere una **natura testuale**. Spesso ciò non è vero (almeno in modo naturale). In questo caso, possiamo considerare lo stream in ***formato binario***, cioè tutti i singoli caratteri dello stream vengono trattati allo stesso modo **senza alcuna interpretazione**.

L'**input binario** da uno istream, cioè carattere per carattere (byte per byte senza interpretazione per i byte) può essere fatto tramite alcuni metodi di "get()" di istream.

```
class istream: public virtual ios {
public:
  int get();
  istream& get(char& c);
  istream& read(char* p, int n);
  istream& ignore(int n=1, int e = EOF);
}
```

Il metodo **`int get()`** preleva un singolo carattere (1 byte) dall'istream di invocazione e lo restituisce convertito ad intero in [0,255]. Se si è tentato di leggere EOF ritorna -1.

Il metodo **`get(char& c)`** invece memorizza in **`c`** il carattere prelevato, se questo esiste.

Il metodo **`read(char* p, int n)`** preleva dall'istream di invocazione **`n`** caratteri, a meno che non incontri prima EOF, e li memorizza in una stringa puntata da **`p`**.

Il metodo **`ignore(int n,int e=EOF)`** effettua il prelievo di **`n`** caratteri ma non li memorizza.

L'**output binario** su uno ostream può essere fatto tramite i seguenti metodi di "put()" di ostream.

```
class ostream: public virtual ios {
public:
  ostream& put(char c);
  ostream& write(const char* p, int n);
...
}
```

Il metodo `put(char c)` scrive il carattere `c` nello ostream di invocazione.

Il metodo `write(const char* p, int n)` scrive sullo ostream di invocazione i primi `n` caratteri della stringa puntata da `p`.

Stream di file

Gli stream associati a file sono oggetti delle classi **`ifstream`**, **`ofstream`** e **`fstream`**. Sono disponibili diversi costruttori (vedere documentazione), i più comuni dei quali sono:

```
ifstream(const char* nomefile, int modalita=ios::in);
ofstream(const char* nomefile, int modalita=ios::out);
fstream(const char* nomefile, int modalita=ios::in | ios::out);
```

La stringa **`nomefile`** è il nome del file associato allo stream, mentre le modalità di apertura dello stream sono specificate da un tipo enum nella classe base **`ios`**

```
class ios {
public:
  enum openmode {
  in,              // apertura in lettura
  out,             // apertura in scrittura
  ate,             // spostamento a EOF dopo l'apertura
  app,             // spostamento a EOF prima di ogni write
  trunc,           // erase file all'apertura
  binary,          // apertura in binary mode, default text mode
  };
...
}
```

member constant	opening mode
app	(**app**end) Set the stream's position indicator to the end of the stream before each output operation.
ate	(**at** end) Set the stream's position indicator to the end of the stream on opening.
binary	(**binary**) Consider stream as binary rather than text.
in	(**in**put) Allow input operations on the stream.
out	(**out**put) Allow output operations on the stream.
trunc	(**trunc**ate) Any current content is discarded, assuming a length of zero on opening.

Le modalità di apertura di uno stream su file possono essere combinate tramite l'OR bitwise |

Per default, gli oggetti di ifstream sono aperti in lettura mentre quelli di ofstream sono aperti in scrittura. Un fstream può essere aperto sia in lettura che in scrittura.

```
fstream file("dati.txt", ios::in|ios::out);
if (file.fail()) cout << "Errore in apertura\n";
```

Apre il file `"dati.txt"` in i/o.

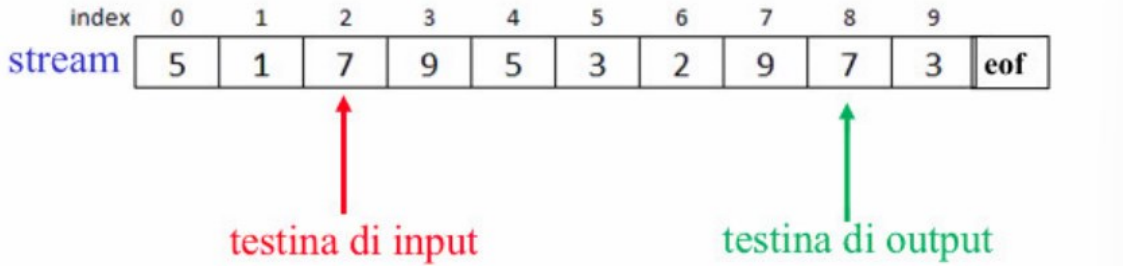
```
ofstream file("dati.txt", ios::app|ios::nocreate|ios::binary);
if (file.bad()) cout << "Il file non esiste\n";
```

Il file `"dati.txt"` viene aperto in modalità binaria di append alla fine.

Chiusura di un file: Il metodo `close()` chiude esplicitamente un file: viene automaticamente invocato dal distruttore dello stream.

Il posizionamento in una cella di uno stream associato ad un file si effettua tramite i seguenti metodi (indici nello stream partono da 0)

```
class istream: public virtual ios {
public:
  long tellg(); // ritorna la posizione di input nello stream
  istream& seekg(long p); // setta la posizione di input nello stream
...
};
class ostream: public virtual ios {
public:
  long tellp(); // ritorna la posizione di output nello stream
  ostream& seekp(long p); // setta la posizione di output nello stream
...
};
```



Le costanti **ios::beg**, **ios::cur** e **ios::end** sono delle posizioni definite in ios:

ios::beg = posizione iniziale dello stream, cioè vale 0.

ios::cur = posizione corrente

ios::end = posizione finale dello stream, cioè la cella di EOF successiva all'ultimo byte dello stream

Esempio

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

int main(){
  fstream f("dati.txt", ios::trunc|ios::in|ios::out);
  if (!f.fail()) cout << "Errore in apertura\n"; ...
  f << "Pippo";
  cout << f.tellp() << endl;
  f.seekp(ios::beg);
  f << "Topolino";
  cout << f.tellp() << endl;
  f << " Pluto";  // append
  cout << f.tellp() << endl; // posizione testina di output: 14
  f.seekg(ios::beg); // testina di input all'inizio
  char c; while (f.get(c)) cout << c; // stampa: Topolino Pluto
}
```

Stream di stringhe

Si possono definire **stream associati a stringhe**, ossia sequenze di caratteri memorizzate in RAM (si parla anche di i/o in memoria). Il carattere nullo di terminazione gioca il ruolo di marcatore di fine stream.

Le classi da utilizzare sono: **`istringstream`**, **`ostringstream`** e **`stringstream`**, il file header che le dichiara è **`<sstream>`.**

I costruttori sono i seguenti.

```
istringstream(const char* initial, int = ios::in);
ostringstream(int = ios::out);
ostringstream(const char* initial, int = ios::out);
stringstream(int = ios::in|ios::out);
stringstream(const char* initial, int = ios::in|ios::out);
```

I metodi di scrittura/lettura sono quelli ereditati da istream, ostream e iostream. Il metodo **`str()`** applicato ad uno stream di stringhe ritorna la stringa associata allo stream. Vediamo un esempio.

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

int main(){
  stringstream ss;
  ss << 236 << ' ' << 3.14 << "  pippo  "; // output su stringstream
  cout << ss.tellp() << ' ' << ss.tellg() << endl;
  // posizioni di testina di output e input: 17 0
  // la stringa in memoria è: "236 3.14  pippo  "
  // la testina di output è avanzata alla fine ios::end
  // la testina di input è ancora a ios::beg
  int i; ss >> i; // input da stringstream
  cout << i << endl; // stampa: 236
  double d; ss >> d; cout << d << endl; // stampa: 3.14
  string s; ss >> s; cout << "*" << s << "*\n"; // stampa: *pippo*
}
```

A cosa può servire?

Ad implementare input/output mediante **`operator>>`** ed **`operator<<`** su stringhe, ad esempio fornite dall'interazione con una GUI.

```
int main() {
  string x("(1a2.23,35)\n(12.23,a35)\n(12.23,35)\n(a14.2,5)\n");
  // ad esempio, stringa x ricevuta in input da GUI
  stringstream ss(x); // stringstream ss inizializzato con x
  Punto p; // posso invocare parsing di un Punto su ss
  while(ss.good()) { // while(stato == 0)
    ss >> p; // parsing di un Punto p
    if(ss.good()) { cout << "Input corretto di un Punto\n"; break; }
    else if(ss.fail()) {
      cout <<"Input non valido!\n";
      ss.clear(ios::goodbit); char c=0;
      // 10 è il codice ASCII del carattere newline
      while(c!=10) { ss.get(c); } // svuota ss sino a newline
      ss.clear(ios::goodbit);
    }
    else ss.clear(ios::failbit);
  }
}
```

Ora abbiamo la gestione delle eccezioni.
Un'eccezione è un problema che si verifica durante l'esecuzione di un programma. Un'eccezione C++ è una risposta a una circostanza eccezionale che si verifica durante l'esecuzione di un programma, come ad esempio il tentativo di dividere per zero.
Le eccezioni forniscono un modo per trasferire il controllo da una parte all'altra del programma. La gestione delle eccezioni in C++ si basa su tre parole chiave: try, catch e throw.

- throw - Un programma lancia un'eccezione quando si presenta un problema. Questo viene fatto utilizzando la parola chiave throw.
- catch - Un programma cattura un'eccezione con un gestore di eccezioni nel punto del programma in cui si desidera gestire il problema. La parola chiave catch indica la cattura di un'eccezione.
- try - Un blocco try identifica un blocco di codice per il quale verranno attivate particolari eccezioni. È seguito da uno o più blocchi catch.

La funzione in cui si verifica la situazione eccezionale *solleva (o lancia) una eccezione* tramite una `throw`

```
telefonata bolletta::Estrai_Una() {
  if (Vuota()) throw Ecc_Vuota();
  telefonata aux = first->info;
  first = first->next;
  return aux;
}
```

```
class Ecc_Vuota {};
```

Nella funzione chiamante:

```
int main() {
  ...
  try { b.Estrai_Una(); }
  catch (Ecc_Vuota e) {
    cerr << "La bolletta è vuota" << endl;
    abort(); // definita in stdlib.h
             // terminazione abnormale di programma
  }
  ...
}
```

function

abort <cstdlib>

C | C++11

```
void abort (void);
```

Abort current process

Aborts the current process, producing an abnormal program termination.

The function raises the SIGABRT signal (as if raise(SIGABRT) was called). This, if uncaught, causes the program to terminate returning a platform-dependent *unsuccessful termination* error code to the host environment.

Abbiamo poi un esempio di parser non robusto, perché ci sono tante situazioni di possibile errore non gestito, poi sistemato con l'aggiunta delle eccezioni apposite.

```
istream& operator>>(istream& is,
                    orario& o) {
  // formato di input: hh:mm:ss
  char c; int ore, minuti, secondi;
  string::size_type pos;
  string cifre("0123456789");
  is >> c; // prima cifra delle ore
  pos = cifre.find(c); ore = pos;
  is >> c;
  if (c != ':') {
    // seconda cifra delle ore
    pos = cifre.find(c);
    ore = ore * 10 + pos;
    is >> c; // input di ':'
  } // ho letto le ore e c = ':'
  is >> c; // prima cifra dei minuti
  pos = cifre.find(c);
  minuti = pos;
  is >> c;
  if (c != ':') {
    // seconda cifra dei minuti
    pos = cifre.find(c);
    minuti = minuti * 10 + pos;
    is >> c; // input di ':'
  } // ho letto i minuti e c = ':'
  is >> c; // prima cifra dei secondi
  pos = cifre.find(c);
  secondi = pos;
  is >> c;
  if(is && cifre.find(c)!= string::npos){
    // per cin, is == 0 con Ctrl-D
    // seconda cifra secondi
    pos = cifre.find(c);
    secondi = secondi * 10 + pos;
  } // ho letto i secondi
  else if (is) // carattere non cifra
  is.putback(c);
  o.sec = ore*3600 + minuti*60 + secondi;
  return is;
}
```

parsing di **orario**
in formato
hh:mm:ss

Si possono verificare varie situazioni di errore in questa funzione. Il parser non è "**robusto**"

Definiamo le seguenti classi di eccezioni

```
class err_sint {};     // errore di sintassi

class fine_file {};    // file finito prematuramente

class err_ore {};      // ora > 23

class err_minuti {};   // minuti > 59

class err_secondi {}; // secondi > 59
```

```
istream& operator>>(istream& is,
                    orario& o) {
  char c; string::size_type pos;
  string cifre("0123456789");
  int ore, minuti, secondi;
  if (!(is >> c)) throw fine_file();
  // prima cifra ore
  pos = cifre.find(c);
  if (pos == string::npos)
    throw err_sint();
  ore = pos;
  if (!(is >> c)) throw fine_file();
  if (c != ':') {
    // seconda cifra ore
    pos = cifre.find(c);
    if (pos == string::npos)
      throw err_sint();
    ore = ore * 10 + pos;
    if (ore > 23) throw err_ore();
    if (!(is >> c))
      throw fine_file();
  }
  // ore lette, c deve essere ':'
  if (c != ':') throw err_sint();
  if (!(is >> c)) throw fine_file();
  // prima cifra minuti
  pos = cifre.find(c);
  if (pos == string::npos)
    throw err_sint();
  minuti = pos;
  if (!(is >> c)) throw fine_file();
  if (!(is >> c)) throw fine_file();
  if (c != ':') {
    // seconda cifra minuti
    pos = cifre.find(c);
    if (pos == string::npos)
      throw err_sint();
    minuti = minuti * 10 + pos;
    if (minuti > 59) throw err_minuti();
    if (!(is >> c)) throw fine_file();
  }
  // minuti letti, c deve essere ':'
  if (c != ':') throw err_sint();
  if (!(is >> c)) throw fine_file();
  // prima cifra secondi
  pos = cifre.find(c);
  if (pos == string::npos) throw err_sint();
  secondi = pos;
  is >> c;
  if (is && cifre.find(c) != string::npos){
    // per cin, is==0 con Ctrl-D
    // seconda cifra secondi
    pos = cifre.find(c);
    secondi = secondi * 10 + pos;
    if (secondi > 59) throw err_secondi();
  } // ho letto i secondi
  else if (is) // carattere non cifra
  is.putback(c);
  o.sec = ore*3600 + minuti*60 + secondi;
  return is;
}
```

parsing **robusto** di **`orario`** in formato `hh:mm:ss`

Una funzione esterna che chiede in input da cin due orari da sommare:

```
orario sommaDueOrari() {
  orario o1,o2;
  try { cin >> o1; } // può sollevare eccezioni
  catch (err_sint)
    {cerr << "Errore di sintassi"; return orario();}
  catch (fine_file)
    {cerr << "Errore fine file"; abort();}
  catch (err_ore)
    {cerr << "Errore nelle ore"; return orario();}
  catch (err_minuti)
    {cerr << "Errore nei minuti"; return orario();}
  catch (err_secondi)
    {cerr << "Errore nei secondi"; return orario();}
  try { cin >> o2; } // può sollevare eccezioni
  catch (err_sint)
    {cerr << "Errore di sintassi"; return orario();}
  catch (fine_file)
    {cerr << "Errore fine file"; abort();}
  catch (err_ore)
    {cerr << "Errore nelle ore"; return orario();}
  catch (err_minuti)
    {cerr << "Errore nei minuti"; return orario();}
  catch (err_secondi)
    {cerr << "Errore nei secondi"; return orario();}
  return o1+o2;
}
```

Utilizziamo un unico blocco `try`:

```
orario sommaDueOrari() {
  try {
    orario o1, o2;
    cin >> o1 >> o2;
    return o1 + o2;
  }
  catch (err_sint)
    {cerr << "Errore di sintassi"; return orario();}
  catch (fine_file)
    {cerr << "Errore fine file"; abort();}
  catch (err_ore)
    {cerr << "Errore nelle ore"; return orario();}
  catch (err_minuti)
    {cerr << "Errore nei minuti"; return orario();}
  catch (err_secondi)
    {cerr << "Errore nei secondi"; return orario();}
}
```

Una **throw** può sollevare una espressione di **qualsiasi tipo**.

```
// enum di eccezioni invece che classi distinte
enum Errori {ErrSintassi, ErrFineFile, ErrOre,
             ErrMinuti, ErrSecondi};
...

   if (secondi > 59) throw ErrSecondi;
// invece di
// if (secondi > 59) throw err_secondi();

...
```

Flusso del controllo provocato da una throw

Quando in una funzione **F** viene sollevata una eccezione di tipo **T** tramite una istruzione **throw** inizia la ricerca della clausola **catch** in grado di catturarla.

① Se l'espressione **throw** è collocata in un blocco **try** nel corpo della stessa funzione **F**, l'esecuzione abbandona il blocco **try** e vengono esaminate in successione tutte le **catch** associate a tale blocco.

② Se si trova un type match per una **catch** l'eccezione viene catturata e viene eseguito il codice della **catch**; eventualmente, al termine dell'esecuzione del corpo della **catch** il controllo dell'esecuzione passa al punto di programma che segue l'ultimo blocco **catch**.

2 - Poco sensato!

③ Se non si trova un type match per una **catch** oppure se l'istruzione **throw** non era collocata all'interno di un blocco **try** della stessa funzione **F** la ricerca continua nella funzione che ha invocato la funzione **F**.

3 - Caso più comune

④ Questa **ricerca top-down sullo stack** delle chiamate di funzioni continua fino a che si trova una **catch** che cattura l'eccezione o si arriva alla funzione **main** nel qual caso viene richiamata la funzione di libreria **terminate()** che per default chiama la funzione **abort()** che fa terminare il programma in errore.

Rilanciare un'eccezione

È possibile che una clausola `catch` si accorga di non poter gestire direttamente una eccezione. In tal caso essa può **rilanciare** l'eccezione alla funzione chiamante con una `throw`.

```
orario somma() try {
  orario t1, t2;
  cin >> t1 >> t2;
  return t1 + t2;
  }
  catch (err_sint)
    {cerr << "Errore di sintassi"; return orario();}
  catch (fine_file)
    {cerr << "Errore fine file"; throw; }
  catch (err_ore)
    {cerr << "Errore nelle ore"; return orario();}
  catch (err_minuti)
    {cerr << "Errore nei minuti"; return orario();}
  catch (err_secondi)
    {cerr << "Errore nei secondi"; return orario();}
```

Un'eccezione può essere lanciata, come segue, bloccando l'ulteriore esecuzione del programma:

```
class A {
public: ~A() {cout << "~A ";}
};

void F() { A* p = new A[3]; throw 1; delete[] p;};

int main() {
  try { F(); }
  catch (int) {cout << "Eccezione int ";}
  cout << "Fine ";
}
// stampa: Eccezione int Fine
// ovviamente non stampa: ~A ~A ~A
```

Utilizzo di risorse

```
gestore () {
  risorsa rs; // alloco la risorsa
  rs.use();
  ...
  ...  // codice che può sollevare eccezioni
  ...
  rs.release(); // non viene eseguita in caso
                // di eccezione
}
```

Se viene sollevata una eccezione e questa non viene catturata all'interno della funzione si esce dalla funzione senza rilasciare la risorsa. Ad esempio, la risorsa è la memoria e quindi si potrebbe provocare **garbage**.

Clausola `catch` *generica*

```
gestore () try {
  risorsa rs;
  rs.use();
  ... // codice che può sollevare eccezioni
  rs.release(); // non viene eseguita in caso
                // di eccezione
}
catch (...) {
  rs.release();
  throw;  // rilancio l'eccezione al chiamante
}
```

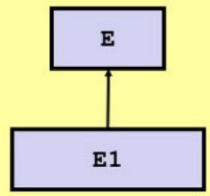
Match del tipo delle eccezioni

La catch che cattura un'eccezione di tipo **E** è la prima catch incontrata durante la ricerca che abbia un *tipo* **T** *compatibile* con **E**.

Le regole che definiscono la compatibilità tra il tipo **T** del parametro di una catch non generica ed il tipo **E** dell'eccezione sono le seguenti:

- Il tipo **T** è uguale al tipo **E**;
- Il tipo **E** è un sottotipo di **T**, ovvero:
  - ✓ **E** è un sottotipo derivato pubblicamente da **T**;
  - ✓ **T** è un tipo puntatore **B*** ed **E** è un tipo puntatore **D*** dove **D** è un sottotipo di **B**
  - ✓ **T** è un tipo riferimento **B&** ed **E** è un tipo riferimento **D&** dove **D** è un sottotipo di **B**
- **T** è il tipo **`void*`** ed **E** è un qualsiasi tipo puntatore
- **Non possono** essere applicate conversioni implicite.

```
class E { public: virtual ~E() {} };
class E1: public E {};

void modify(vector<int>& v) {
   ...
   if(v.size()==0) throw new E();
   if(v.size()==1) throw new E1();
   ...
}

void G(vector<int>& v) {
try{
  modify(v);
}
catch(E* p) {...}
catch(E1* q) {...}
}
```



Comportamenti tipici di una clausola catch sono i seguenti:

- rilanciare un'eccezione
- convertire un tipo di eccezione in un altro, rimediando parzialmente e lanciando un'eccezione diversa
- cercare di ripristinare il funzionamento, in modo che il programma possa continuare dall'istruzione che segue l'ultima catch
- analizzare la situazione che ha causato l'errore, eliminarne eventualmente la causa e riprovare a chiamare la funzione che ha causato originariamente l'eccezione
- esaminare l'errore ed invocare `std::terminate()`

Specifica esplicita delle eccezioni (alla Java)

```
istream& operator>>(istream& is, orario& t)
  throw(err_sint, fine_file, err_ore, err_secondi,
        err_minuti) {
  ...
}
```

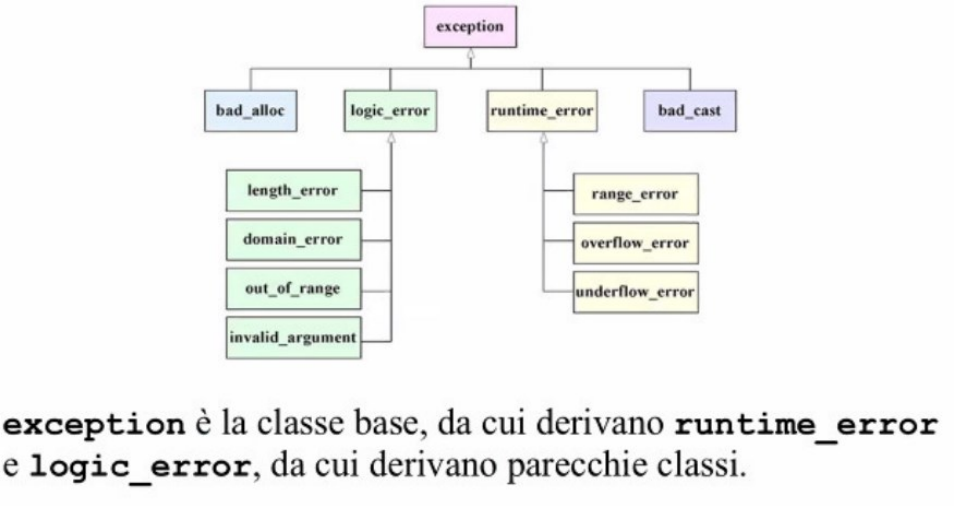
Deprecata da C++11

Problemi nella specifica delle eccezioni

- **Run-time checking**: il test di conformità delle eccezioni avviene a run-time e non a compile-time, quindi non vi era una garanzia statica di conformità.
- **Run-time overhead**: Run-time checking richiede al compilatore del codice addizionale che potrebbe inficiare alcune ottimizzazioni.
- **Inutilizzabile con i template**: in generale i parametri di tipo dei template non permettono di specificare le eccezioni.

Importante: Qt, ad esempio, non possiede le eccezioni di per sé, in quanto già allora presenti per tutti i compilatori. Vanno quindi definite, alla bisogna, dall'utente.

In C++ si ha la gerarchia *exception.*
L'esperienza ha dimostrato che le eccezioni si suddividono in diverse categorie. La libreria standard del C++ include una gerarchia di classi di eccezioni. Questa gerarchia è guidata dalla classe base exception (definita nel file di intestazione), che contiene la funzione virtuale what, che le classi derivate possono sovrascrivere per emettere messaggi di errore appropriati.



exception è la classe base, da cui derivano **runtime_error** e **logic_error**, da cui derivano parecchie classi.

Le classi derivate immediate della classe base exception includono runtime_error e logic_error (entrambe definite nell'intestazione), ognuna delle quali ha diverse classi derivate. Da exception derivano anche le eccezioni lanciate dagli operatori del C++, ad esempio bad_alloc è lanciata da new, bad_cast è lanciata da dynamic_cast e bad_typeid è lanciata da typeid. Includere bad_exception nell'elenco dei lanci di una funzione significa che, se si verifica un'eccezione inattesa, la funzione unexpected può lanciare bad_exception invece di terminare l'esecuzione del programma (per impostazione predefinita) o chiamare un'altra funzione specificata da set_unexpected.

Collocare un gestore di catch che cattura un oggetto di classe base prima di un catch che cattura un oggetto di una classe derivata da quella classe base è un errore logico. Il catch della classe base cattura tutti gli oggetti delle classi derivate da quella classe base; quindi, il catch della classe derivata non verrà mai eseguito.

- La classe logic_error è la classe base di diverse classi di eccezioni standard che indicano errori nella logica del programma. Ad esempio, la classe invalid_argument indica che è stato passato un argomento non valido a una funzione. (Una corretta codifica può, ovviamente, evitare che argomenti non validi raggiungano una funzione). La classe length_error indica che è stata utilizzata una lunghezza superiore alla dimensione massima consentita per l'oggetto da manipolare. La classe out_of_range indica che un valore, come un pedice in un array, ha superato l'intervallo di valori consentito.

- La classe runtime_error è la classe base di diverse altre classi di eccezioni standard che indicano errori in fase di esecuzione. Ad esempio, la classe overflow_error descrive un errore di overflow aritmetico (cioè, il risultato di un'operazione aritmetica è più grande del numero più grande che può essere memorizzato nel computer) e la classe underflow_error descrive un errore di underflow aritmetico (cioè, il risultato di un'operazione aritmetica è più piccolo del numero più piccolo che può essere memorizzato nel computer).

- Le classi di eccezione definite dal programmatore non devono necessariamente derivare dalla classe exception. Pertanto, scrivendo catch (exception anyException) non si garantisce la cattura di tutte le eccezioni che un programma potrebbe incontrare.

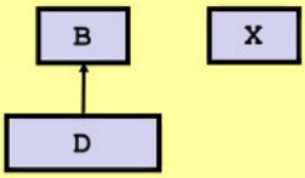
Per catturare tutte le eccezioni potenzialmente lanciate in un blocco try, utilizzare catch(...). Un punto debole della cattura delle eccezioni in questo modo è che il tipo di eccezione catturata è sconosciuto al momento della compilazione. Un altro punto debole è che, senza un parametro con nome, non c'è modo di fare riferimento all'oggetto eccezione all'interno del gestore dell'eccezione.

La gerarchia delle eccezioni standard è un buon punto di partenza per la creazione di eccezioni. I programmatori possono creare programmi in grado di lanciare eccezioni standard, lanciare eccezioni derivate dalle eccezioni standard o lanciare eccezioni proprie non derivate dalle eccezioni standard.

Usare catch(...) per eseguire un recupero che non dipende dal tipo di eccezione (ad esempio, liberare risorse comuni). L'eccezione può essere rigettata per avvisare gestori di catch più specifici.

Se il **dynamic_cast** di un riferimento fallisce allora viene automaticamente lanciata un'eccezione di tipo **bad_cast**

```
class X { public: virtual ~X() {} };
class B { public: virtual ~B() {} };
class D : public B {};

#include<typeinfo>
#include<iostream>
using namespace std;
int main() {
  D d;
  B& b = d; // upcast
  try {
    X& xr = dynamic_cast<X&>(b);
  } catch(bad_cast e) {
      cout << "Cast fallito!" << endl;
    }
}
```



Derivano da **exception** anche le seguenti classi di eccezioni:

- **bad_cast**, le cui eccezioni sono lanciate dal **dynamic_cast** per riferimenti
- **bad_alloc**, lanciata dalla **new** quando lo heap è esaurito (il gestore di default invoca la **terminate()**).
- **bad_typeid**, viene lanciata dall'operatore **typeid** quando ha come argomento un puntatore nullo.

Di C++ 11 non consideriamo:

- Puntatori smart (unique_ptr, shared_ptr, comunque da me spiegati)
- Multithreading (concorrenza)
- Lambda expressions
- R-value reference type T&& (temporanei modificabili) → se si volesse approfondire su questo (https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2027.html#:~:text=An%20rvalue%20reference%20is%20a,lvalue%20reference%20and%20rvalue%20reference)

Compilazione C++11 (dalla versione 4.7)

```
g++ -std=c++11
```

Cambiamenti utili a noi: inferenza automatica di tipo (deduzione del tipo in base al contesto di invocazione), tramite la keyword auto.

```
vector< vector<int> >::const_iterator cit=v.begin();
```

Keyword: **auto**

Dichiarazioni di variabili senza specifica del loro tipo.

```
auto x = 0;   // x ha tipo int perché 0 è un litterale di tipo int
auto c = 'f'; // char
auto d = 0.7; // double
auto debito_nazionale = 2500000000000L; // long int
auto y = qt_obj.qt_fun(); // y ha il tipo di ritorno di qt_fun
```

Permette di evitare alcune verbosità dello strong typing, specialmente per i template.

```
void fun(const vector<int> &vi){
 vector<int>::const_iterator ci=vi.begin();
 ...
}

// posso rimpiazzarlo con

void fun(const vector<int> &vi){
 auto ci=vi.begin();
 ...
}
```

```
void fun(vector<int> &vi){
 auto ci=vi.begin(); // tipo: vector<int>::iterator
 ...
}
```

La determinazione statica del tipo delle espressioni avviene attraverso la keyword *decltype*. Similmente, l'inizializzazione in linea, alla C, con una serie di graffe dei vari array.

```
int x = 3;
decltype(x) y = 4;
```

```
std::vector<int> v(1);
auto a = v[0];          // a ha tipo int
decltype(v[1]) b = 1; // b ha tipo int
auto c = 0;             // c ha tipo int
auto d = c;             // d ha tipo int
decltype(c) e;          // e ha tipo int
decltype(0) f;          // f ha tipo int
```

Inizializzazione uniforme per **array**

```
// inizializzazione di array dinamico
int* a = new int[3] {1,2,0};

class X {
  int a[4];
public:
  X() : a{1,2,3,4} {} // inizializzazione di campo dati array
};
```

Inizializzazione uniforme per **contenitori STL**

```
// inizializzazione di contenitori in C++11
std::vector<string> vs = {"first", "second", "third"};

std::map<string,string> singerPhones =
  { {"SferaEbbasta", "347 0123456"},
    {"RogerWaters", "348 9876543"} };

void fun(std::list<double> l);
fun({0.34, -3.2, 5, 4.0});
```

Per ogni classe sono disponibili le versioni **standard** di:

1) costruttore di default

2) costruttore di copia

3) assegnazione

4) distruttore

In C++11 tali funzioni standard si possono rendere esplicitamente di default oppure non disponibili.

```
class A {
public:
 A(int) {}                 // costruttore ad 1 argomento
 A() = default;            // costruttore altrimenti non disponibile
 virtual ~A() = default; // distruttore virtuale standard
};
```

```
class NoCopy {
public:
 NoCopy& operator=(const NoCopy& ) = delete;
 NoCopy (const NoCopy&) = delete;
};

int main() {
  NoCopy a,b;
  NoCopy b(a); // errore in compilazione
  b=a;         // errore in compilazione
}
```

```
class OnlyDouble {
public:
 static void fun(double) {}
 template <class T> static void fun(T) = delete;
 // NESSUNA CONVERSIONE A DOUBLE PERMESSA
};

int main() {
 int a=5; float f=3.1;
 OnlyDouble::fun(a); // ILLEGALE: use of deleted function with T=int
 OnlyDouble::fun(f); // ILLEGALE: use of deleted function with T=float
}
```

La dichiarazione di funzione esplicitamente default è una nuova forma di dichiarazione di funzione introdotta nello standard C++11, che consente di aggiungere lo specificatore '=default;' alla fine di una dichiarazione di funzione per dichiararla come funzione esplicitamente predefinita. In questo modo il compilatore genera le implementazioni predefinite per le funzioni esplicitamente predefinite, che sono più efficienti delle implementazioni di funzioni programmate manualmente.
Ad esempio, ogni volta che dichiariamo un costruttore parametrizzato, il compilatore da solo In questo caso, possiamo usare lo specificatore default per crearne uno predefinito.

Quali sono i vantaggi di '=default' quando si potrebbe semplicemente lasciare un corpo vuoto della funzione usando '{}'?
Anche se i due comportamenti possono essere identici, ci sono comunque dei vantaggi nell'usare default rispetto al lasciare un corpo vuoto del costruttore.

I punti seguenti spiegano come:

- Dare un costruttore definito dall'utente, anche se non fa nulla, rende il tipo non un aggregato e nemmeno banale. Se si vuole che la classe sia un aggregato o un tipo banale (o, per transitività, un tipo POD), è necessario usare '= default'.
- L'uso di '= default' può essere utilizzato anche con i costruttori e i distruttori di copia. Un costruttore di copie vuoto, ad esempio, non farà la stessa cosa di un costruttore di copie predefinito (che eseguirà la copia dei suoi membri). L'uso della sintassi '= default' in modo uniforme per ognuna di queste funzioni membro speciali rende il codice più facile da leggere.

Prima del C++ 11, l'operatore delete aveva un solo scopo, quello di deallocare una memoria allocata dinamicamente.
Lo standard C++ 11 ha introdotto un altro uso di questo operatore, ovvero: Disabilitare l'uso di una funzione membro. Ciò avviene aggiungendo lo specificatore =delete; alla fine della dichiarazione della funzione.
Qualsiasi funzione membro il cui utilizzo è stato disabilitato utilizzando lo specificatore '=delete' è nota come funzione esplicitamente cancellata.
Anche se non si limita a queste, di solito viene fatto per le funzioni implicite.

Quali sono i vantaggi di delete esplicito delle funzioni?

- L'eliminazione di funzioni membro speciali fornisce un modo più pulito per evitare che il compilatore generi funzioni membro speciali che non vogliamo. (Come dimostrato nell'esempio "Disabilitazione dei costruttori di copie").
- L'eliminazione di normali funzioni membro o non membro impedisce che promozioni di tipo problematiche causino la chiamata di una funzione non voluta (come dimostrato nell'esempio "Disabilitazione della conversione indesiderata degli argomenti").

La già discussa keyword *override*:

keyword: **override**
Per dichiarare esplicitamente quando si definisce un overriding di un metodo virtuale

```
class B {
public:
  virtual void m(double) {}
  virtual void f(int) {}
};

class D: public B {
public:
  virtual void m(int) override {} // ILLEGALE
  virtual void f(int) override {} // OK
};
```

Serve per evitare di definire, o di dimenticare, inavvertitamente degli overriding

Per dichiarare l'ultimo overriding, si usa le keyword *final*.

```
class B {
public:
  virtual void m(int) {}
};

class C: public B {
public:
  virtual void m(int) final {} // final overrider
};

class D: public C {
public:
  virtual void m(int) {}; // ILLEGALE
};
```

final non richiede che la funzione sovrascriva qualcosa in primo luogo. Il suo effetto è definito come:
"Se una funzione virtuale f in una classe B è marcata con lo specificatore final e in una classe D derivata da B una funzione D::f sovrascrive B::f, il programma è malformato."
Questo è quanto. Ora override final significherebbe semplicemente
"Questa funzione sovrascrive una funzione della classe base (override) e non può essere sovrascritta essa stessa (final)".
final da solo imporrebbe un requisito più debole. override e final hanno un comportamento indipendente.

Si noti che final può essere usato solo per le funzioni virtuali

Quindi la dichiarazione:

void foo() final;

è effettivamente uguale a

virtual void foo() final override;

Poiché entrambe richiedono che foo sovrascriva qualcosa, la seconda dichiarazione utilizza override, mentre la prima è valida se e solo se foo è implicitamente virtuale, cioè quando foo sovrascrive una funzione virtuale chiamata foo in una classe base, il che rende foo nella classe derivata automaticamente virtuale. Pertanto, l'override sarebbe superfluo nelle dichiarazioni in cui è presente final, ma non virtual. Tuttavia, quest'ultima dichiarazione esprime l'intento in modo molto più chiaro e dovrebbe essere preferita.

"keyword" `override final`

Note that neither `override` nor `final` are language keywords. They are technically identifiers; **they only gain special meaning when used in those specific contexts. In any other location, they can be valid identifiers.**

Scritto da Gabriel

Esercizio: final può permette al compilatore di ottimizzare la devirtualizzazione.
(link di riferimento: https://quuxplusone.github.io/blog/2021/02/15/devirtualization/#when-we-know-a-proof-of-leafness-for-its-static-type)

Il caso di esempio è:

```
void test() {
   Apple o;
   o.f();
}
```

Non importa se *Apple::f* è virtuale; tutto ciò che il dispatch virtuale fa è invocare il metodo sul tipo dinamico effettivo dell'oggetto, e in questo caso sappiamo che il tipo dinamico effettivo è proprio *Apple*. Il dispatch statico e dinamico dovrebbe dare lo stesso risultato in questo caso.
Un compilatore sufficientemente intelligente userà l'analisi del flusso di dati per ottimizzare casi non banali come:

```
Derived d;
Base *p = &d;
p->f();
```

Si scopre che anche questo semplice espediente è sufficiente per ingannare MSVC (compilatore di Microsoft Visual Studio) e ICC (Intel C++ Compiler). Il prossimo caso di test è:

```
Derived da, db;
Base *p = cond ? &da : &db;
p->f();
```

Questo è troppo per Clang, ma GCC riesce a sopravvivere... finché non si spostano le conversioni in *Base** all'interno di una condizione! Qui è dove anche l'analisi di GCC fallisce (analisi con Godbolt/compilatore creato da un dipendente di Google con GCC/Clang supportati (in tutte le versioni ed architetture), dando una documentazione, highlighting del codice e filtri in tempo reale):

```
Derived da, db;
Base *p = cond ? (Base*)&da : (Base*)&db;
p->f();
```

Quando conosciamo una "proof of leafness (dimostrazione per ramificazione)" per il suo tipo static

Supponiamo di ricevere un puntatore da un'altra parte del sistema. Conosciamo il suo tipo statico (ad esempio *Derived**), ma non conosciamo il tipo dinamico dell'istanza dell'oggetto a cui punta. Tuttavia, il compilatore può devirtualizzare una chiamata a *Derived::f* se riesce a dimostrare in qualche modo che nessun tipo dell'intero programma può mai sovrascrivere *Derived::f*.

Dimostrazione con *final*

La più semplice "proof of leafness" si ha quando si segna *Derived* come final.

```
struct Base {
   virtual int f();
};
struct Derived final : public Base {
   int f() override { return 2; }
```

Scritto da Gabriel

```
};
int test(Derived *p) {
   return p->f();
}
```

Un puntatore di tipo *Derived** deve puntare a un'istanza di oggetto che sia "almeno Derived", cioè Derived o uno dei suoi figli. Poiché Derived è finale, non può avere figli; pertanto, il tipo dinamico dell'istanza deve essere esattamente Derived e il compilatore può devirtualizzare questa chiamata.
Oppure si può contrassegnare il metodo specifico *Derived::f* come final.

La stessa analisi dovrebbe essere applicata indipendentemente dal fatto che Derived::f sia dichiarato in Derived stesso o ereditato da Base. Quindi, per esempio, il compilatore dovrebbe essere ugualmente in grado di devirtualizzare il metodo.

```
struct Base {
   virtual int f() { return 1; }
};
struct Derived final : public Base {};
int test(Derived *p) {
   return p->f();
}
```

GCC, Clang e MSVC superano questo test (Godbolt, caso uno); ICC 21.1.9 viene ingannato.

Una prova assolutamente bizzarra è quella di osservare che quando il distruttore della classe C è final, C deve essere senza figli, perché se C avesse un figlio, il figlio dovrebbe avere un distruttore (dato che non si può creare una classe senza un distruttore), che quindi sovrascriverebbe il distruttore di C, il che non è consentito. Clang avverte i distruttori finali e li ottimizza.

Prova per collegamento interno

Una classe il cui nome ha un legame interno non può essere chiamata al di fuori dell'unità di traduzione corrente. Pertanto, non può nemmeno essere derivata dall'esterno dell'unità di traduzione corrente! Se non ha figli nell'unità di traduzione corrente - o almeno non ha figli che sovrascrivono i suoi metodi - le chiamate alle sue funzioni virtuali sono devirtualizzabili.

```
namespace {
   class BaseImpl : public Base {};
}
int test(Base *p) {
   return static_cast<BaseImpl*>(p)->f();
}
```

Se p punta davvero a un'istanza di un oggetto che è "almeno BaseImpl", il compilatore può dimostrare che l'istanza deve essere esattamente BaseImpl. (E se p non punta a un'istanza che sia "almeno BaseImpl", il programma ha comunque un comportamento non definito).

Nelle situazioni reali, è comune avere una classe base esposta pubblicamente nel file di intestazione e poi una o più implementazioni derivate strettamente legate a un singolo file .cpp. Se si fa il passo più lungo della gamba e si mettono le implementazioni derivate in namespace anonimi, si può aiutare la logica di devirtualizzazione del compilatore. Naturalmente, per definizione, qualsiasi beneficio sarà limitato a quel singolo file .cpp!

Scritto da Gabriel

Un altro modo in cui il nome di un tipo può avere un collegamento interno è quando si tratta dell'istanziazione di un modello di classe in cui uno dei parametri del modello coinvolge un nome con collegamento interno. Se il nome T ha un legame interno, allora anche E<T> ha un legame interno, anche se E stesso ha un legame esterno, perché non si può nominare E<T> senza nominare T. (Si noti che in questo caso T deve essere un "vero nome"; non stiamo parlando di alias di tipo).

È anche possibile creare un tipo il cui nome ha un collegamento esterno, ma in cui il compilatore può dimostrare che il tipo deve essere incompleto in ogni altro TU (Translation Unit/output del preprocessore). Per esempio,

```
namespace {
   class Internal {};
}
class External { Internal m; };
```

Qualsiasi altro TU può dichiarare la classe External; come tipo incompleto, ma questi output non possono mai completare il tipo, perché non possono nominare i tipi dei suoi membri. Non si può derivare da un tipo incompleto. Quindi, tutti i tipi derivati da External (se ce ne sono) devono essere presenti in questo TU; se non ce ne sono, beh, questa è una prova di assenza! Solo GCC rileva questa situazione.

Ulteriore riferimento: https://devblogs.microsoft.com/cppblog/the-performance-benefits-of-final-classes/

Lo specificatore final in C++ indica una classe o una funzione membro virtuale che non può essere derivata o sovrascritta. Ad esempio, si consideri il codice seguente:

```
struct base {
  virtual void f() const = 0;
};

struct derived final : base {
  void f() const override {}
};
```

Se si tenta di scrivere una nuova classe che deriva da `*derived*`, si ottiene un errore del compilatore:

```
struct oh_no : derived {
};
```

```
<source>(9): error C3246: 'oh_no': cannot inherit from 'derived' as it has been declared as 'final'
<source>(5): note: see declaration of 'derived'
```

Lo specificatore *final* è utile per esprimere ai lettori del codice che una classe non deve essere derivata e far sì che il compilatore la faccia rispettare, ma può anche migliorare le prestazioni favorendo la devirtualizzazione.

Devirtualizzazione

Le funzioni virtuali richiedono una chiamata indiretta attraverso la vtable, che è più costosa di una chiamata diretta a causa delle interazioni con la branch prediction (predizione dei salti nella CPU) e la cache delle istruzioni, e dell'impossibilità di effettuare ulteriori ottimizzazioni dopo l'inlining della chiamata.

La devirtualizzazione è un'ottimizzazione del compilatore che cerca di risolvere le chiamate a funzioni virtuali in fase di compilazione anziché in fase di esecuzione. In questo modo si eliminano tutti i problemi

sopra descritti e si possono migliorare notevolmente le prestazioni del codice che utilizza molte chiamate virtuali1.

Ecco un esempio minimo di devirtualizzazione:

```
struct dog {
  virtual void speak() {
    std::cout << "woof";
  }
};

int main() {
  dog fido;
  fido.speak();
}
```

In questo codice, anche se dog::speak è una funzione virtuale, l'unico risultato possibile di main è l'emissione di "bau". Se si osserva l'output del compilatore, si noterà che MSVC, GCC e Clang riconoscono questo aspetto e inseriscono la definizione di dog::speak in main, evitando la necessità di una chiamata indiretta.

Il beneficio di final

Lo specificatore *final* può fornire al compilatore maggiori opportunità di devirtualizzazione, aiutandolo a identificare più casi in cui le chiamate virtuali possono essere risolte in fase di compilazione. Torniamo all'esempio iniziale:

```
struct base {
  virtual void f() const = 0;
};

struct derived final : base {
  void f() const override {}
};
```

Si consideri la funzione:

```
void call_f(derived const& d) {
  d.f();
}
```

Poiché *derived* è contrassegnato come *final*, il compilatore sa che non può essere derivato da altri. Ciò significa che la chiamata a f chiamerà sempre e solo derived::f; quindi, la chiamata può essere risolta in fase di compilazione. Come prova, ecco l'output del compilatore per call_f su MSVC quando derived o derived::f sono contrassegnati come final:

```
ret 0
```

Si può notare che il *derived::f* è stato inserito nella definizione di *call_f*. Se togliessimo lo specificatore final dalla definizione, l'assembly avrebbe questo aspetto:

```
mov rax, QWORD PTR [rcx]
rex_jmp QWORD PTR [rax]
```

Questo codice carica la vtable da d, quindi effettua una chiamata indiretta a *derived::f* attraverso il puntatore alla funzione memorizzato nella posizione corrispondente.

Il costo del caricamento del puntatore e del salto può sembrare poco, dato che si tratta di due sole istruzioni, ma ricordate che questo potrebbe comportare un errore di predizione del ramo e/o una mancanza di cache delle istruzioni, con conseguente stallo della pipeline. Inoltre, se ci fosse più codice in call_f o nelle funzioni che la chiamano, il compilatore potrebbe essere in grado di ottimizzarlo in modo molto più aggressivo, data la piena visibilità del codice che verrà eseguito e l'analisi aggiuntiva che ciò consente.

Conclusione

Contrassegnare le classi o le funzioni membro come definitive può migliorare le prestazioni del codice, dando al compilatore più opportunità di risolvere le chiamate virtuali in fase di compilazione.
Valutate se nelle vostre basi di codice ci sono punti che ne trarrebbero beneficio e misuratene l'impatto!

Per dichiarare i puntatori nulli, normalmente si usa la keyword *nullptr*.
Nel C++ Standard, ciò corrisponde anche a 0.

```
void f(int);
void f(char*);

int main() {
  f(0);         // quale f invoca? invoca f(int)
}
```

keyword: **nullptr**
Può sostituire la macro **NULL** ed il valore 0.

nullptr ha come tipo **std::nullptr_t** che è convertibile implicitamente a **qualsiasi tipo puntatore ed a bool**, mentre non è convertibile implicitamente ai tipi primitivi integrali

```
void f(int);
void f(char*);

int main() {
  f(nullptr);   // quale f invoca? invoca f(char*)
}
```

```
const char* pc = str.c_str();
if (pc != nullptr) std::cout << pc << endl;
```

Un costruttore nella sua lista di inizializzazione può invocare un **altro costruttore della stessa classe**, un meccanismo noto come **delegation** e disponibile in linguaggi come Java

```
class C {
 int x, y;
 char* p;
public:
 C(int v, int w) : x(v), y(w), p(new char [5]) {}
 C(): C(0,0) {}
 C(int v): C(v,0) {}
};
```

È una alternativa al meccanismo degli argomenti di default dei costruttori; alternativa considerata **preferibile** da alcuni esperti di programmazione

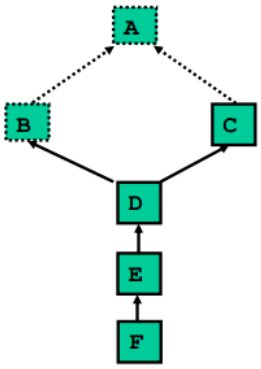
```
class A {
public:
  virtual void m() =0;
};

class B: virtual public A {};

class C: virtual public A {
public:
  virtual void m() {}
};

class D: public B, public C {
public:
  virtual void m() {}
};

class E: public D {};

class F: public E {};

char G(A* p, B& r) {
  C* pc = dynamic_cast<E*>(&r);
  if(pc && typeid(*p)==typeid(r)) return 'G';
  if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
  if(!dynamic_cast<F*>(pc)) return 'A';
  else if(typeid(*p)==typeid(E)) return 'S';
  return 'E';
}
```

Si consideri inoltre il seguente statement.

```
  cout << G(new X1,*new Y1) << G(new X2,*new Y2) << G(new X3,*new Y3) << G(new X4,*new Y4)
       << G(new X5,*new Y5) << G(new X6,*new Y6) << G(new X7,*new Y7) << G(new X8,*new Y8);
```

Definire opportunamente le incognite di tipo `Xi` e `Yi` tra i tipi `A`, `B`, `C`, `D`, `E`, `F` della precedente gerarchia in modo tale che:

1. Lo statement non includa piú di una chiamata della funzione `G` con gli stessi parametri attuali
2. La compilazione dello statement non produca illegalità
3. L'esecuzione dello statement non provochi errori a run-time
4. L'esecuzione dello statement produca in output esattamente la stampa **SAGGEZZA**.



$TD(r) \in \{D,E,F\}$
$TD(*p) \in \{C,D,E,F\}$

output G $\in \{(E,E), (F,F)\}$
$TD(r) \leqq E \;\&\; TD(*p) = TD(r)$

output Z $\in \{(C,D), (D,D)\}$
$\neg G \;\&\; TD(r) \not\leq E \;\&\; TD(*p) \leqq D$

output A $\in \{(C,E)\ (D,E), (E,D), (F,E), (F,D)\}$
$\neg Z \;\&\; \neg G \;\&\; TD(r) \not\leq F$

output S $\in \{(E,F)\}$
$\neg Z \;\&\; \neg G \;\&\; \neg A \;\&\; TD(r) = F \;\&\; TD(*p)=E$

output E $\in \{(C,F), (D,F)\}$
$\neg Z \;\&\; \neg G \;\&\; \neg A \;\&\; \neg S$

```
class A {
public:
  virtual void m() =0;
};

class B: virtual public A {};

class C: virtual public A {
public:
  virtual void m() {}
};

class D: public B, public C {
public:
  virtual void m() {}
};

class E: public D {};

class F: public E {};

char G(A* p, B& r) {
  C* pc = dynamic_cast<E*>(&r);
  if(pc && typeid(*p)==typeid(r)) return 'G';
  if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
  if(!dynamic_cast<F*>(pc)) return 'A';
  else if(typeid(*p)==typeid(E)) return 'S';
  return 'E';
}
```

S=(E,F) A=(C,D) G=(E,E) G=(F,F)
E=(C,F) Z=(D,D) Z=(E,D) A=(C,E)

Si consideri inoltre il seguente statement.

```
  cout << G(new X1,*new Y1) << G(new X2,*new Y2) << G(new X3,*new Y3) << G(new X4,*new Y4)
       << G(new X5,*new Y5) << G(new X6,*new Y6) << G(new X7,*new Y7) << G(new X8,*new Y8);
```

Definire opportunamente le incognite di tipo `Xi` e `Yi` tra i tipi `A`, `B`, `C`, `D`, `E`, `F` della precedente gerarchia in modo tale che:

1. Lo statement non includa piú di una chiamata della funzione `G` con gli stessi parametri attuali
2. La compilazione dello statement non produca illegalità
3. L'esecuzione dello statement non provochi errori a run-time
4. L'esecuzione dello statement produca in output esattamente la stampa **SAGGEZZA**.

```
// dichiarazione incompleta
template<class T> class D;

template<class T1, class T2>
class C {
  // amicizia associata
  friend class D<T1>;
private:
  T1 t1; T2 t2;
};

template<class T>
class D {
public:
  void m(){C<T,T> c;
           cout << c.t1 << c.t2;}
  void n(){C<int,T> c;
            cout << c.t1 << c.t2;}
  void o(){C<T,int> c;
           cout << c.t1 << c.t2;}
  void p(){C<int,int> c;
           cout << c.t1 << c.t2;}
  void q(){C<int,double> c;
           cout << c.t1 << c.t2;}
  void r(){C<char,double> c;
           cout << c.t1 << c.t2;}
};
```

I seguenti main() compilano?

```
main() {D<char> d; d.m();} // C
main() {D<char> d; d.n();} // NC
main() {D<char> d; d.o();} // C
main() {D<char> d; d.p();} // NC
main() {D<char> d; d.q();} // NC
main() {D<char> d; d.r();} // C
```

```
class Z {
private:
  int x;
};

class B {
private:
  Z x;
};

class D: public B {
private:
  Z y;
public:
  // ridefinizione di operator=
  ...
};
```

Ridefinire l'assegnazione `operator=` della classe `D` in modo tale che il suo comportamento coincida con quello dell'assegnazione standard di `D`.

```
D& operator=(const D& d) {
  B::operator=(d);
  y=d.y;
  return *this;
}
```