Glossario di Programmazione ad Oggetti

Riccardo Graziani

Anno Accademico 2024/2025

Contents

1 Panoramica del linguaggio C++

1.1 Generalitá del C++

In generale, un programma é costituito da:

- un insieme di algoritmi;
- un insieme di dati su cui operano gli algoritmi;

Quando si pone enfasi sugli algoritmo si parla di **programmazione procedurale** mentre se si sottolineano i dati, o meglio i tipi di dati, si parla di **programmazione ad oggetti**.
Il C++ é un linguaggio orientato agli oggetti, a **tipizzazione statica** che mette a disposizione strumenti per supportare entrambi gli stili di programmazione, ossia:

- le funzioni, che permettono al programmatore di estendere il linguaggio con nuove **istruzioni**
- le classi, che permettono al programmatore di estendere il linguaggio con nuovi **tipi di dato**

Il C++ implementa al suo interno le caratteristiche principali della **OOP** (Object Oriented Programming) legate al concetto di classe:

- Incapsulamento e information hiding;
- Polimorfismo;
- Ereditarietá;

Inoltre il C++ fornisce strumenti per integrare i paradigmi di programmazione procedurale e ad oggetti quali:

- Template di funzione;
- Template di classe;
- Gestione delle eccezioni;

1.2 Richiami di C++

1.2.1 I namespace

Nella programmazione modulare uno dei problemi principali é l'inquinamento dello spazio dei nomi, ovvero non capire chi sta richiamando cosa. Il meccanismo dei namespace permette di incapsulare dei nomi che inquinerebbero il namespace globale, ad esempio:

```
// file "Lib_Uno.h"

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

Per poter usare le strutture di tale namespace devo utilizzare l'operatore di scoping **namespace::nome_dichiarazione**:

```
#include "Lib.Uno.h"

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

Posso definire un **alias** per un namespace per renderlo piú breve o generico:

```
#include "Lib.Uno.h"

namespace UNO = SPAZIO_UNO;

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

Per non dover utilizzare sempre l'operatore di scoping si puó ricorrere alle direttive d'uso **using namespace nome_namespace**:

```
#include "Lib.Uno.h"

using namespace SPAZIO_UNO;
```

```
void funzione() {
    Complex var1;
    Complex f(var1);
}
```

Tale pratica é peró scorretta e sconsigliata, e si preferisce ad essa la dichiarazione d'uso **using nome_namespace::nome_dichiarazione**:

```
#include "Lib.Uno.h"

using SPAZIO_UNO::Complex;
using SPAZIO_UNO::f;

void funzione() {
    Complex var1;
    Complex f(var1);
}
```

Le componenti di tutte le librerie del C++ standard, in particolare la **Standard Template Library** (STL) e la libreria di I/O sono dichiarate nel namespace **std**.

1.2.2 Argomenti di default

Un argomento di default é un valore dato nella dichiarazione di un parametro formale x di una funzione F() che il compilatore inserisce automaticamente quando non viene fornite un valore attuale esplicito per x, ad esempio:

```
// La potenzaa é sempre intesa, in questo caso, di grado 2
double potenza(double x, int n = 2) {
    ...
}
```

Bisogna ricordare che non é possibile avere un argomento di default seguito da un argomento **non di default**:

```
void F(double x, int n = 3, string s) { ... } // Illegale
void G(double x, int n = 3, string s = "ciao") { ... } // OK

G(3.2); G(); G(3, 3); G(3, 3, "pippo");
// OK; Illegale; OK; OK
```

1.2.3 Puntatori a funzione

In C++ possiamo definire dei **puntatori a funzione**, ovvero un puntatore che contiene l'indirizzo in memoria di una data segnatura di funzione incluso il suo tipo di ritorno:

```cpp
#include<math.h> //ci interessano sin(double) e cos(double)
#include<iostream>
using namespace std;

double F(double d) { return d + 3.14; }

int main() {
// pf é un puntatore a funzione con lista dei
// parametri (double) e tipo di ritorno double
double (*pf) (double) // parentesi in (*pf) obbligatorie
pf = &sin; cout << (*pf)(0) << endl; // stampa 0
pf = &cos; cout << (*pf)(0) << endl; // stampa 1
}
```

1.2.4 Keyword typedef

La keyword **typedef** permette di dichiarare un alias per un tipo giá esistente:

```cpp
typedef unsigned short int Intero;
typedef const int* PuntoCostante;
typedef logn double ArrayReale[20];
typedef struct {
    double parte_reale;
    double parte_immaginaria;
} Complesso;
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
```

1.2.5 Operatore virgola

L'operatore virgola separa delle espressioni e le valuta procedendo da sinistra verso destra e ritorna il valore solamente dell'ultima espressione:

```cpp
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
}
```

1.2.6 Stringhe

Una classe definita nella libreria STL del C++, e presenta diversi metodi predefiniti:

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

int main() {
```

```
    // Due modi di dichiarare una stringa
    string st(""Ciao");
    string st = "Ciao";
    cout << "Lunghezza: " << st.size(); // Ritorna la lunghezza di st
    cout << "Stringa vuota?" << st.empty(); // Ritorna bool
}
```

Inoltre su string sono definiti gli operatori:

`==, !=, <, <=, >, >=, =, +`

2 Classi e Oggetti

2.1 Tipi di dato astratti (ADT)

Un tipo di dato astratto o **ADT** (Abstract Data Type) è un tipo di dato le cui istanze possono essere manipolate con modalità che dipendono esclusivamente dalla **semantica** (anche detto comportamento) del dato e non dalla sua realizzazione.
Il tipo di dato viene definito distinguendo nettamente la sua **interfaccia** dalla sua implementazione interna (incapsulamento - information hiding). I tipi primitivi come int e string **rispettano** le proprietá degli ADT, mentre una **struct no** (la sua rappresentazione é visibile all'esterno).
Un esempio di ADT é quello dei numeri complessi:

```
// File "complessi.h"
struct comp {
    double re, im;
};
comp iniz_comp(double, double);
double reale(comp);
double immag(comp);
comp somma (comp, comp);

// File "complessi.cpp"
#include <complessi.h>

comp iniz_comp(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;
}

// File "main.cpp"
#include<complessi.h>
#include<iostream>
using std::cout;

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

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

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

2.2 Le classi

La possibilitá di definire delle classi permette al programmatoree di estendere il linguaggio con nuovi tipi di dato astratti. Una classe viene specificata in due parti **separate**:

- la definizione dell'interfaccia della classe (anche detta **dichiarazione della classe**) che consiste nella dichiarazione dei campi dati e dei metodi della classe;
- la definizione (anche detta **implementazione**) dei suoi metodi;

Prendiamo come esempio la classe **orario** che rappresenta un orario della giornata:

```
// Dichiarazione della classe
class orario {
    // Campi dati della classe
    private:
    // Rappresentiamo un orario come il n. di
    // sec passati dalla mezzanotte
    int sec;
```

```
    // Metodi della classe
    public:
    // Selettori per ore, minuti e secondi
    int Ore();
    int Minuti();
    int Secondi();
}

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

Sarebbe possibile unire le due fasi in un unico file, definendo i metodi con modalitá **inline**, la quale é peró una pratica sconsigliata.
Ora possiamo dichiarare variabili di tipo orario dove la dichiarazione della classe é visibile:

```
int main() {
    orario mezzanotte;
    // Chiediamo di eseguire il metodo Secondi() sull'oggetto
    // mezzanotte di tipo orario
    cout << mezzanotte.Secondi(); << endl;
}
```

2.2.1 Il puntatore this

Quando viene dichiarato un oggetto di tipo orario viene risevata una parte di memoria per il valore del campo sec, mentre vi é **un'unica copia** in memoria del codice dei metodi della classe. I metodi di una classe possiedono un campo implicito **this** di tipo puntatore ad oggetti della classe stessa, ovvero:

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

// Esplicito il parametro this
int orario::Secondi(orario* this) { return ((*this).sec) % 60 }

// Nella chiamata del metodo
int s = mezzanotte.Secondi();
int s = Secondi(&mezzanotte);
```

2.3 Parte pubblica e privata

Nella dichiarazione della classe orario vengono uasti i **modificatori di accesso** public e private. Dall'esterno della classe é possibile accedere solo a ció che é marcato public, mentre dentro la classe si puó accedere liberamente alla parte private (se un membro non é **esplicitamente marcato** é di default private).

2.4 I costruttori

I costruttori sono metodi che hanno lo stesso **nome** della classe e non possiedono tipo di ritorno. Vengono invocati automaticamente quando un oggetto viene dichiarato (e quindi costruito). Essi sono generalmente dichiarati nella parte public di una classe:

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

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

int main() {
    orario mezzanotte; // Invoca il costruttore di default
    cout << mezzanotte.Ore() << endl; // Stampa 0
}
```

Si possono definire piú costruttori tramite **overloading** dell'identificatore del metodo, in questo caso cambiando il numero dei parametri:

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

Da notare un caso particolare, ovvero:

```
orario o;
o = orario(12, 33, 35);
```

in cui il costruttore a tre parametri crea un cosiddetto **oggetto anonimo**, ossia un oggetto a cui non é associato alcun identificatore di orario. Il suo tempo di vita termina non appena viene usato per l'assegnazione all'oggetto o.
Se in una classe non viene dichiarato alcun costruttore, il C++ rende disponibile il **costruttore standard**, un costruttore di default con il seguente comportamento:

- lascia indefiniti i valori dei campi dati dei tipi primitivi;
- per i campi dati di tipo classe richiama il corrispondente costruttore di default, che potrá essere esplicito o standard;

Da notare che se dichiariamo un costruttore qualsiasi in una classe non é piú disponibile quello standard. Altri modi per dare valore ad un oggetto orario sono:

```
orario adesso(11, 55); // Costruttore ore - minuti
orario copia; // Costruttore di default
copia = adesso; // Assegnazione
orario copia1 = adesso; // Costruttore di copia
orario copia2(adesso); // Costruttore di copia
```

L'invocazione del **costruttore di copia** crea un nuovo oggetto copia1 mediante una copia campo dati per campo dati dell'oggetto adesso. Il costruttore di copia di una generica classe C ha sempre forma C(const C&). L'operatore di assegnazione permette di assegnare il valore di ogni campo dati di adesso al corrispondente campo dati di copia.

2.4.1 Costruttori come convertitori di tipo

I costruttori ad un parametro funzionano anche da **convertitori di tipo**, cioé possono essere usati per conversioni implicite: se C(T) é un costruttore della classe C allora esso viene invocato quando compare un valore di tipo T dove ci si aspetterebbe un valore di tipo C. Da notare che la conversione implicita avviene a **run-time** e provoca i seguenti effetti sul seguente esempio:

```
orario x; x = 8;

// 1. viene invocato il costruttore orario(int) con parametro
// attuale 8 che crea un oggetto temporaneo anonimo di orario
// 2. l'oggetto temporaneo viene assegnato ad x
// 3. viene deallocato l'oggetto temporaneo
```

2.4.2 Keyword explicit

Se non vogliamo che un costruttore ad un parametro sia richiamato implicitamente come convertitore di tipo basta apporre la keyword **explicit** davanti a tale costruttore.

2.4.3 Operatori espliciti di conversione

É possibile definire una conversione implicicta tra due classi C e T tramite l'uso di **operatori espliciti di conversione**:

```
class orario {
    public:
```

```
    // Conversione orario -> int
    operator int() { return sec; }
};

// Richiama operator int() sull'oggetto o
orario o(14, 37); int x = o;
```

2.5 Metodi costanti

L'invocazione di un metodo su un oggetto puó provocare modifiche al suo stato (ossia modificare i suoi **campi dati**). Si dice che il metodo provoca **side effects** sull'oggetto di invocazione. Supponendo di ampliare la classe orario con:

```
class orario {
    private:
    ...

    public:
    ...
    orario UnOraPiuTardi();
    vodi AvanzaUnOra();

};

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

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

Per evitare modifiche indesiderate all'oggetto di invocazione si puó dichiare un metodo come **costante**:

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

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

Il compilatore controlla che metodi marcati costanti non possiedano istruzioni che possano alterare l'oggetto di invocazione come **assegnazione ai campi o invocazione di metodi non costanti sull'oggetto o sui suoi campi dati**. Ad esempio il metodo UnOraPiuTardi deve essere marcato costante poiché non

causa side effects sull'oggetto di invocazione.
Per i metodi constanti vale la regola: nel corpo di un metodo costante di una classe C il puntatore this ha tipo **const C*** e l'oggetto di invocazione ha tipo **const C**.
Da ricordare che un oggetto costante si puó usare come oggetto di invocazione **solamente** per metodi dichiarati costanti. I costruttori sono l'eccezione, poiché non sono dichiarati costanti ma possono accettare oggetti di invocazione costanti.

2.6 Campi dati e metodi statici

Studiando la nostra classe orario potremmo voler creare un metodo OraDiPranzo() che ritorni sempre lo stesso orario. Potremmo operare cosí:

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

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

Il problema é che ci servirebbe un oggetto costante per poter chiamare OraDiPranzo. Possiamo usare la keyword **static** che permette di associare sia metodi che campi dati all'intera classe invece che a singoli oggetti:

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

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

int main() {
    cout << "Si pranza alle: " << orario::OraDiPranzo().Ore()
         << " e " << orario::OraDiPranzo().Minuti();
}
```

Da notare che metodi static non hanno il parametro implicito this.
É possibile dichiarare anche i campi dati come static, facendo ció la memoria per tale campo **é unica e viene condivisa** tra tutti gli oggetti della classe. Tali campi dati vanno peró inizializzati al di fuori della definizione della classe.

2.7 Overloading degli operatori

Volendo sommare oggetti della classe orario potremmo definire il seguente metodo di somma:

```
orario orario::Somma(orario o) {
    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.Somma(DUE_ORE_E_UN_QUARTO);
}
```

Un modod piú elegante é quello di definire un **overloading** dell'operatore +:

```
class orario {
    public:
    orario operator+(orario);
    ...
};

orario orario::operator+(orario o) {
    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;
}
```

Il C++ permette di fare overloading di molti operatori tra i quali:

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

L'overloading non puó cambiare le proprietá sinattiche dell'operatore ovvero:

- posizione (prefissa, infissa o postfissa);
- numero di operandi;
- precedenza;
- associativitá;

Fra gli argomenti dell'operatore ridefinito deve comparire almeno un argomento di tipo definito da utente, mentre gli operatori:

```
= [] () ->
```

possono essere ridefiniti solo con metodi propri: ció assicura che il primo operando sia un **l-valore**.

2.7.1 Assegnazione standard

Se a e b sono due oggetti della stessa classe C e non é stato ridefinito l'operatore = per C allora a = b; ha come effetto quello di assegnare ad ognuno dei campi dati di a il valore corrispondente dei **campi dati di b**. La segnatura dell'assegnazione standard é:

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

2.7.2 Costruttore di copia standard

Il costruttore di copia per una generica classe C ha la seguente firma:

```
C(const C&);
```

e viene invocato automaticamente in tre casi:

- quando un oggetto viene **dichiarato** e **inizializzato** con un altro oggetto della stessa classe;
- quando un oggetto viene passato per **valore** come parametro attuale in una chiamata di funzione;
- quando una funzione ritorna per valore tramite l'istruzione return un oggetto (come ad esempio **return aux** nella somma di orario);

Il caso tre viene ottimizzato dal compilatore ed evita la creazione del temporaneo "inutile": nel corso si assume che tale ottimizzazione sia disattivata con il comando **-fno-elide-constructors**.

2.7.3 Overloading di operatori con funzioni esterne

Vogliamo definire una funzione che stampi sullo stream di output cout un oggetto orario. La scelta naturale é quella di effettuare un overload su **operator«**:

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

Per invocare tale operatore correttamente dovremmo usare la seguente sintassi:

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

Questo perché il compilatore interpreta la prima e seconda occorrenza di "«" come le seguenti chiamate al metodo:

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

Mentre noi vorremmo poter scrivere:

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

Possiamo ottenere ció tramite l'overloading di operator« come **funzione esterna** alla classe:

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

2.8 Modularizzazione delle classi

Le classi, una volta definite, possono essere utilizzate per definire modularmente altre classi piú complesse. Un esempio é la seguente classe **telefonata** che sfrutta la classe orario:

```
// file "telefonata.h"
#ifndef TELEFONATA.H
#define TELEFONATA.H
#include<iostream>
#include"orario.h"
using std::ostream;

class telefonata {
    private:
    orario inizio, fine;
    int numero;

    public:
    telefonata(orario, orario, int);
    telefonata();
    orario Inizio() const;
    orario Fine() const;
    int Numero() const;
    bool operator==(const telefonata&) const;
}
#endif
```

Consideriamo una classe C con campi dati $x_1, ..., x_k$ di qualsiasi tipo, preferibilmente di qualche altra classe. **L'ordine dei campi dati** é determinato dall'ordine in cui essi appaiono nella definizione della classe C. Supponiamno di avere un costruttore nella classe C:

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

Il comportamento di tale costruttore é il seguente:

- per ogni campo dati x_i di tipo T_i, **primitivo o derivato**, viene allocato un corrispondete spazio in memoria per contenere un valore di tipo T_i ed il valore viene lasciato indefinito;
- ogni campo dati x_i di tipo classe T_i viene costruito mediante una invocazione del costruttore di default $T_i()$;
- infine viene eseguito il corpo del costruttore di C;

Ritornando alla classe telefonata, definiamo i metodi:

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

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

// I campi orario sono costruiti dal loro costruttore di default
telefonata::telefonata() { numero = 0; }

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

2.9 Campi dati costanti

Risulta ragionevole supporre che una volta costruito un oggetto telefonata il numero non sará modificato in seguito:

```
class telefonata {
    private:
    orario inizio, fine;
    const int numero;

    public:
```

```
    ...
}
```

Tale dichiarazione rende peró il costruttore senza argomenti invalido, poiché tenta di assegnare un valore a un campo costante. Dobbiamo richiamare esplicitamente il **costruttore di copia** per il campo dati costante invece di allocare memoria con il **costruttore standard** degli interi. La nuova sintassi é:

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

Possiamo richiamare esplicitamente il costruttore di copia per qualsiasi campo dati anche non costante. Quindi la definizione del costruttore di telefonata diventa:

```
telefonata::telefonata(orario i, orario f, int n)
                      : inizio(i), fine(f), numero(0) {}
```

Questo nuovo costruttore non usa i costruttori di default per i campi per poi cambiarne valore con le assegnazioni ma usa direttamente i costruttori di copia. Per evitare di richiamare i costruttori di copia per il passaggio dei parametri per valore possiamo dichiarare per riferimento costante tali parametri formali. Ció permette sia a degli oggetti costanti che degli oggetti anonimi di essere passati come parametri attuali:

```
telefonata::telefonata(const orario& i, const orario& f, int n)
                      : inizio(i), fine(f), numero(0) {}
```

2.10 Liste di inizializzazione nei costruttori

In una classe C con lista ordinata di campi dati $x_1, ..., x_k$ un costruttore con lista di inizializzazione per i campi dati $x_{i1}, ..., x_{ij}$ é definito tramite la seguente sintassi: C(T1,...,Tn) : $x_{i1}(...), ..., x_{ij}(...)$
Il comportamento del costruttore é il seguente:

- per ogni campo dati x_i con $1 \leq i \leq k$ viene richiamato un costruttore in modo esplicito tramite $x_i(...)$ o implicito tramite $x_i()$;
- viene eseguito il codice del costruttore C;

Notiamo che:

- parliamo di costruttori e costruttori di copia anche per campi dati di tipo non classe: la lista di inizializzazione puó includere inizializzazioni anche per questi campi dati;
- la chiamata implicita al costruttore di default per un campo dati di tipo non classe alloca lo spazio in memoria ma lascia indefinito il valore;
- l'ordine con cui vengono invocati i costruttori, esplicitamente o implicitamente, é sempre determinato dalla lista ordinata dei campi dati, qualsiasi sia l'ordine delle chiamate nella lista di inizializzazione;

É possibile dichiarare un campo dati c di tipo riferimento T& dove T é un qualsiasi altro tipo. Questo campo dati c deve essere obbligatoriamente inizializzato tramite una chiamata al costruttore di copia inclusa nella lista di inizializzazione.

2.10.1 Costruttori standard e di copia di default

Il comportamento dei costruttori standard e di copia di default é il seguente:

- una chiamata C() al costruttore di default standard di una classe C invoca ordinatamente per ogni campo dati x di C il corrispondente costruttore di default (standard o esplicito), dove per i tipi non classe semplicemente alloca la memoria per il campo dati x;
- una chiamata C(obj) la costruttore di copia standard di una classe C invoca ordinatamente per ogni campo dati x di C il corrispondente costruttore di copia (standard o esplicito) sul relativo campo dati obj.x dell'oggetto parametro attuale obj, dove per tipi non classe alloca la memoria per x e la inizializza;

3 Classi collezione e argomenti correlati

Un oggetto di una classe **collezione** (o contenitore) rappresenta una collezione di elementi che puó essere gestita tramite varie funzionalitá quali l'inserimento e la rimozione. Usiamo come esempio una classe che rappresenta l'insieme di telefonate memorizzate in una lista, chiamata **bolletta**:

```
// file "bolletta.h"
#ifndef BOLLETTA.H
#define BOLLETTA.H
#include "telefonata.h"

class bolletta {
    private:
    class nodo {
        public:
        nodo();
        nodo(const telefonata&, nodo*);
        telefonata info;
        nodo* next;
    };
    nodo* first

    public:
    bolletta() : first(0) {}
    bool Vuota() const;
    void Aggiungi_Telefonata(telefonata);
    void Togli_Telefonata(telefonata);
```

```
    telefonata Estrai_Una();
}
#endif

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

bolletta::nodo::nodo() : next(0) {}

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

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

void bolletta::Aggiungi_Telefonata(telefonata t) {
    first = new nodo(t, first);
}

void bolletta::Togli_Telefonata(telefonata t) {
    nodo* p = first, *prec = 0;
    while (p && (p->info == t)) {
        prec = p; p = p->next;
    }
    if (p) {
        if (!prec) {
            first = p->next;
        } else {
            prec->next = p->next;
        }
        delete p;
    }
}

telefonata bolletta::Estrai_Una() {
    nodo* p = first;
    first = first->next;
    telefonata aux = p->info;
    delete p;
    return aux;
}
```

3.1 Classi annidate

Nella definizione di bolletta notiamo che nodo é definita internamente alla classe bolletta. La classe nodo é dunque una **classe annidata** ed é a tutti gli effetti un membro di bolletta. Se nodo fosse privata non potrei definire oggetti di nodo

al di fuori di bolletta usando lo scoping. Tra le due classi valgono le stesse regole di accessibilitá ai membri.

3.2 Il problema dell'interferenza

Il metodo Aggiungi_Telefonata() di bolletta modifica l'oggetto di invocazione. Supponendo di aver effettuato l'overloading dell'operatore di output, avendo il seguente main:

```
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(t1);
    b1.Aggiungi_Telefonata(t2);
    cout << b1;
    bolletta b2;
    b2 = b1;
    b2.Togli_Telefonata(t1);
    cout << b1 << b2;
}
```

L'output che otterremo sará:

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

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

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

L'effetto che accade viene chiamato **interferenza tra oggetti o aliasing**, ed é dovuta a due cause concomitanti:

- vi é **condivisione di memoria** tra gli oggetti;
- vi sono funzioni che **modificano** gli oggetti

Per evitare la condivisione di memoria occorre ridefinire sia l'assegnazione che il costruttore di copia in modo tale che essi effettuino una **copia profonda**: ogni volta che copiano un puntatore devono eseguire anche una copia dell'oggetto puntato.

3.3 Copie profonde

3.3.1 Assegnazione profonda

Nel contesto della bolletta vogliamo ridefinire l'operator+ per poter effettuare copie profonde: a tal scopo definiamo due metodi statici copia (che copia in maniera profonda una lista) e distruggi (che dealloca una data lista).

```
// file "boilletta.h"
class bolletta {
    private:
    static nodo* copia(nodo*);
    static void distruggi(nodo*);

    public:
    bolletta& operator=(const bolletta&);
};

// file "bolletta.cpp"
bolletta::nodo* bolletta::copia(nodo* p) {
    if (!p) return 0;
    nodo* primo = new nodo;
    primp->info = p->info;
    nodo* q = primo;
    while (p->next) {
        q->next = new nodo;
        p = p->next;
        q = q->next;
        q->info = p->info;
    }
    q->next = 0;
    return primo;
}

void bolletta::distruggi(nodo* p) {
    nodo* q;
    while (p) {
        q = p;
        p = p->next;
        delete q;
    }
}
```

Usando le funzioni private appena definite, definiamo operator= come:

```
bolletta& bolletta::operator=(const bolletta& b) {
    first = copia(b.first); // Assegnazione tra puntatori
```

```
    return *this; // Ritorna l'oggetto di invocazione
}
```

Saltano subito fuori due nuovi problemi, ovvero:

- quando si esegue un'assegnazione y = x l'oggetto y puó giá avere una sua lista di elementi e tale memoria non viene deallocata;
- se si esegue un'assegnazione del tipo x = x viene effettuata inutilmente la copia della lista di elementi;

Migliorando la nostra definizione otteniamo il risultato:

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

3.3.2 Costruttore di copia profonda

Anche il costruttore di copia puó creare condivisione di memoria. Modifichiamo in modo coerente la classe bolletta:

```
// file "bolletta.h"
class bolletta {
    private:
    ...

    public:
    bolletta(const bolletta&);
    ...
};

// file "bolletta.cpp"
bolletta::bolletta(const bolletta& b) : first(copia(b.first)) {}
```

3.4 Oggetti come parametri di funzione

Ogni volta che una funzione viene invocata i suoi parametri formali vengono allocati nello stack (nel **record di attivazione** della funzione) e inizializzati con i corrispondenti parametri attuali. Esistono due meccanismi per passare parametri in C++:

- per valore: in questo caso i parametri formali vengono inizializzati tramite delle copie degli **r-valori** dei parametri attuali. Questo metodo é costoso in termini di tempo che di spazio in memoria e le modifiche fatte sui parametri formali non si riflettono sui parametri attuali;

- per rifermento: dal prototipo di funzione T fun(T& x) il parametro x é un riferimento ad una variabile di tipo T e quindi il parametro attuale a di una chiamata fun(a) deve essere un'espressione indirizzabile di tipo T, ovvero deve possedere un **l-valore**. Il parametro x diventa un **alias** del parametro attuale a e le modifiche apportate si riflettono;

Vogliamo scrivere uan funzione Somma_Durate esterna alla classe bolletta:

```
orario Somma_Durate(bolletta b) {
    orario durata;
    while(!b.Vuota()) {
        telefonata t = b.Estrai_Una();
        durata = durata + t.Fine() - t.Inizio();
    }
    return durata;
}
```

Eseguendo il seguente main:

```
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(t1);
    b1.Aggiungi_Telefonata(t2);
    cout << b1;
    cout << "LA SOMMA DELLE DURATE -> " << Somma_Durate(b1) << endl;
    cout << b1;
}

//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 -> 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
```

In questo esempio b1 é passato per valore in Somma_Durate(b1), b é un oggetto locale a Somma_Durate() ed é copia profonda di b1. Risulta che la funzione Estrai_Una() provoca **side effects** su b.
Definiamo la funzione esterna Chiamate_A:

```
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;
}
```

Eseguendo il seguente main:

```
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
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
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:
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:
1) INIZIO 12:17:5 FINE 12:22:8 NUMERO 2121212
2) INIZIO 9:23:12 FINE 10:4:53 NUMERO 2121212
```

La funzione Chiamata_A presenta un problema: crea due oggetti **selezionate** e **resto** che esistono solo durante la sua esecuzione. Dopo che resto viene copiata profondamente in b non viene recuperata la memoria dinamica allocata. La lista

selezionate viene ritornata alla fine, ma la return usa il costruttore di copia per creare un oggetto anonimo: siccome anceh il costruttore di copia effetua una copia profonda anche questa memoria non viene recuperata, oltre alla memoria iniziale di selezioante.

3.5 Tempo di vita delle variabili

Il tempo di vita delle variabili é l'intervallo di tempo in cui la variabile viene mantenuta in memoria:

- le variabili di **classe automatica** sono definite all'interno di qualche blocco (tipicamente di funzione), vivono nello stack e sono deallocate quando il loro blocco termina la sua esecuzione;
- le variabili di **classe statica** sono le variabili **globali** definite all'esterno di funzioni e classi, i campi dati statici e le variabili statiche: le prime due vengono allocate all'inizio dell'esecuzione del programma mentre le altre sono allocate quando l'esecuzione raggiunge la loro definizione. Tutte vengono deallocate al termine dell'esecuzione del programma;
- le variabili **dinamiche** sono sempre allocate nello **heap**, vengono allocate quando viene eseguito l'operatore **new** e deallocate quando viene invocato l'operatore **delete**;

3.6 Distruttore

Quando termina il tempo di vita di un oggetto di qualche classe viene richiamato automaticamente un particolare metodo detto **distruttore**. Nella sua versione standard esso rilascia la memoria occupata dall'oggetto, se l'oggetto possiede campi dati puntatore allora non verrá deallocata la memoria a cui puntano i campi puntatore. Dobbiamo quindi ridefinire il distruttore in modo che effettui una distruzione profonda degli oggetti, ovvero deallocando anche la memoria a cui puntano i campi puntatore. Nella classe bolletta definiamo il distruttore:

```
// file "bolletta.h"
class bolletta {
    private:
    ...

    public:
    ~bolletta();
    ...
}

//file "bolletta.cpp"
bolletta::~bolletta() {
    distruggi(first);
}
```

Le regole di invocazione dei distruttori sono le seguenti:

- per gli oggetti di classe **statica**, al termine del programma
- per gli oggetti di classe **automatica** definiti in un blocco, all'uscita del blocco in cui sono definiti (in particolare per i parametri formali di funzione);
- per gli oggetti **dinamici** (allocati sullo heap) quando viene eseguito l'operatore delete sui corrispondenti puntatori, altrimenti al termine del programma;
- per gli oggetti che sono **campi dati** di qualche oggetto x, quando x viene distrutto;
- gli oggetti con lo stesso tempo di vita (oggetti definiti nello stesso blocco oppure oggetti statici) vengono distrutti nell'ordine inverso a quello in cui sono stati creati;

In particolare il distruttore viene invocato nei seguenti casi:

- sulle variabili **locali** di una funzione al termine della sua esecuzione;
- sui paramentri di una funzione passati per **valore** al termine della sua esecuzione;
- sull'**oggetto anonimo** ritornato come risultato di una funzione non appena esso sia stato usato

In particolare, la distruzione al ritorno di una chiamata di funzione segue questo ordine:

- vengono distrutte le **variabili locali**
- viene distrutto l'**oggetto anonimo** ritornato per valore non appena sia stato usato;
- vengono distrutti i **parametri** passati per valore

Supponiamo che lista ordinata di dichiarazione dei campi dati di una classe C sia $x_1, ..., x_n$. Quando viene distrutto un oggetto di tipo C viene invocato automaticamente il distruttore della classe C (standard o ridefinito) con il seguente comportamento:

- viene eseguito il corpo del distruttore della classe C se esiste;
- vengono richiamati i distruttori per i campi dati nell'ordine inverso alla loro lista di dichiarazione. Per un campo dati di tipo non classe viene rilasciata la memoria, mentre per i tipi classe viene invocato il loro distruttore (standard o ridefinito);

Il distruttore standard ha corpo vuoto, quindi in una classe C si limita a richiamare i distruttori per i campi dati di C in ordine inverso alla loro dichiarazione. Definiamo un distruttore per la classe nodo:

```
bolletta::nodo::~nodo() {
    if (next != 0) {
        delete next;
    }
}
```

In questo modo il distruttore di bolletta diventa:

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

Dobbiamo peró modificare il metodo Estrai_Una():

```
telefonata bolletta::Estrai_Una() {
    nodo* p = first;
    first = first->next;
    telefonata aux = p->info;
    p->next = 0;
    delete p;
    return aux;
}
```

3.7 Nascondere la parte privata di una classe

Supponiamo di aver definito una classe C e di voler nascondere la parte **privata** di C all'utente. All'interno di C possiamo dichiarare una classe C_privata internamente e privatamente alla classe C_handle che conterrá la parte privata di C:

```
class C {
    private:
    ...

    public:
    ...
};

// file "C_handle.h"
class C_handle {
    private:
    class C_privata;
    C_privata* punt;

    public:
    ...
```

```
};

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

3.8 Array di oggetti

É possibile definire array **statici** o **dinamici** di oggetti: per la loro costruzione viene richiamato implicitamente il costruttore di default, per ogni oggetto dell'array. Per al distruzione viene richiamato implicitamente il distruttore per ogni oggetto dell'array.

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

3.9 Cast

Il C++ introduce delle notazione esplicite per differenziare i vari tipi di **cast** possibili. Le categorie di conversione di tipo sono:

- conversioni **implicite** o coercions;
- conversione **esplicite**;
- conversioni **predefinite** dal linguaggio;
- conversioni **definite da utente**;
- conversioni **con o senza perdita** di informazioni (narrow o wide conversions);

Inoltre un'espressione e si dice **convertibile implicitamente** a T se:

```
// Questa dichiarazione deve essere compilabile
T t = e;
```

3.9.1 Static cast

```
static_cast<Tipo>(Espressione)
```

Questo cast rende esplicito l'uso di **tutte** le conversioni implicite o meno, previste e permesse o definite dal programmatore. I cast statici si basano esclusivamente sulle informazioni disponibili a compile-time. La tabella delle conversioni implicite sicure é:

```
T& -> T // Non viceversa
T[] -> T* // int[2] a = {3, 1}
T* -> void* // Generic pointer int* p = &x;
T -> const T // int x = 5; const int y = x;
const notPointer -> Pointer // C* const -> C*
T* -> const T* // int* p = &x; const int* q = p;
T -> const T& // int x = 4; const int& r = x;
// Tra tipi primitivi
```

Esistono due tipi di conversioni:

- conversioni **wide** che modificano un valore in un tipo di dati che puó consentire qualsiasi valore possibile dei dati originali (**conservano** il valore di partenza ma ne **modificano** la rappresentazione);
- conversioni **narrow** che modificano un valore in un tipo di dati che potrebbe non essere in grado di contenere alcuni dei valori possibli (**perdita** di precisione);

3.9.2 Const cast

```
const_cast<T*>(puntatore); const_cast<T&>(riferimento)
// Esempio
const int i = 5;
int* p = const_cast<int*>(&i);

void F(const C& x) { const_cast<C&>(x).metodo_non_costante(); }

int j = 7;
const int* q = &j; // Ok per cast implicito
```

Questo cast permette di convertire un **puntatore** o **riferimento** ad un tipo const T ad un puntatore o riferimento T (quindi perdendo il modificatore const).

3.9.3 Reinterpret cast

```
reinterpret_cast<T*>(puntatore); reinterpret_cast<T&>(riferimento)
// Esempio
Classe c;
int* p = reinterpret_cast<int*>(&c);
const char* a = reinterpret_cast<const char*>(&c);
string s(a); cout << s;
```

Questo cast si limita a reinterpretare a **basso livello** la sequenza di bit con cui é rappresentato il valore puntato da puntatore come fosse un valore di tipo T.

3.9.4 Dynamic cast

```
dynamic_cast<T*>(puntatore); dynamic_cast<T&>(riferimento)
```

In questo cast il "tipo dinamico" di puntatore o riferimento non é noto a compile-time ma solo a **run-time**.

3.10 Funzioni amiche

Supponiamo di voler definire l'operatore di output per la classe bolletta. Definiamolo all'esterno della classe:

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

Il C++ rende disponibile una keyword **friend** tramite la quale funzioni esterne a una classe possono accedere ai suoi campi privati, nel nostro caso:

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

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

3.11 Classi amiche e iteratori

Per poter scorrere gli elementi di una collezione é possibile definire una classe **iteratore** i cui oggetti sono indici di elementi della classe contenitore. Un

esempio di una classe contenitore e iteratore é:

```
class contenitore {
    friend class iteratore; // Non necessaria da C++03
    private:
    class nodo {
        ...
    };
    nodo* first;

    public:
    class iteratore {
        friend class contenitore;
        ...
    }
    iteratore begin() const;
    iteratore end() const;
    int& operator[](const iteratore&) const;
};

contenitore::iteratore contenitore::begin() const {
    iteratore aux;
    aux.punt = first;
    return aux;
}

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

int& contenitore::operator[](const contenitore::iteratore& it) const
{ return it.punt->info; }

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

Le regole da ricordare quando si crea un iteratore sono:

- va sempre preceduta da una dichiarazione di amicizia nella parte privata della classe contenitore;

- anche la classe contenitore deve essere friend della classe iteratore (perché necessita di avere accesso ai suoi campi privati per definire i metodi pubblici);
- la classe iteratore va dichiarata prima della dichiarazione dei metodi che usano iteratori;
- nella classe iteratore non vanno ridefinitri **assegnazione, costruttori di copia e distruttori**;

Portando gli iteratori nella classe bolletta:

```
class iteratore {
    friend class bolletta;
    private:
    bolletta::nodo* punt;

    public:
    bool operator==(const iteratore&) const;
    bool operator!=(const iteratore&) const;
    iteratore& operator++();    // Prefisso
    iteratore& operator++(int); // Postfisso
}

class bolletta {
    private:
    ...

    public:
    iteratore begin() const;
    iteratore end() const;
    telefonata& operator[](const iteratore&) const;
    ...
}
```

3.12 Dichiarazioni incomplete di classi

Una classe C puó usare puntatori e riferimenti ad una classe D che non é definita ma é solamente dichiarata tramite una **dichiarazione incompleta**.

3.13 Condivisione controllata della memoria

La ridefinizione in bolletta di costruttore di copia e assegnazione evita la condivisione di memoria, ma effettuare molte copie e distruzioni profonde ha un costo non indifferente. Possiamo ridurre tale costo evitando copie profonde inutili, ad esempio se passiamo un parametro per valore a una funzione che non lo modifica. La prima tecnica che vediamo é il **reference counting**:

```
class bolletta {
    private:
    class nodo {
        public:
        nodo();
        nodo(const telefonata&, nodo*):
        telefonata info;
        nodo* next;
        int riferimenti;
    }
    nodo* first;

    public:
    ...
}
```

In essenza si incapsula in una classe il puntatore nodo* e si ridefinisce **assegnazione, costruzione di copai e distruzione**. Si definisce poi un cosiddetto **smart pointer**, i quali dovranno essere dotati di interfaccia pubblica che permette all'utente di usarli come fossero puntatori ordinari.

3.13.1 Smart pointers

Uno smart porinter é una classe wrapper su un puntatore con un operatore come * e -> overloaded. Tali puntatori possono deallocale e liberare la memoria dell'oggetto distrutto.
L'idea é di prendere una classe con un puntatore, un distruttore e operatori overloaded come * e ->. Poiché il distruttore viene richiamato automaticamente quando un oggetto esce dall'ambito, la memoria allocata dinamicamente viene automaticamente cancellata (o il conteggio dei riferimento puó essere decrementato).

4 Template

Vogliamo definire una funzione che calcola il minimo tra due valori: dovremo creare divere funzioni per i diversi tipi di valore. Una soluzione attraente sarebbe la seguente:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
// Funziona con sostituzioni semplici
min(10, 20) -> 10 < 20 ? 10 : 20
// Ma sbaglia in casi complessi
int i = 3, j = 6;
cout << min(++i, --j); // Stampa 5 non 4
```

La soluzione al problema sono i **template di funzione** che, assieme ai **template di classe**, implementano in C++ il **polimorfismo parametrico**.

L'idea dei template é quella di passare il **tipo di dato** come parametro, in modo da riutilizzare il codice per tipi diversi, che saranno determinati a run-time: ció viene definita **programmazione generica**.

4.1 Template di funzione

Riformuliamo la funzione di minimo con i template:

```
template <class T> // Oppure <typename T>
T min(T a, T b) {
    return a < b ? a : b;
}

int main() {
    int i, j, k;
    orario r, s, t;
    ...
    // Istanziazione implicita del template
    k = min(i, j);
    t = min(r, s);
    // Istanziazione esplicita del template
    k = min<int>(i, j);
    t = min<orario>(r, s);
}
```

I parametri di un template possono essere:

- parametri **di tipo**, ovvero che si possono istanziare con un tipo qualsiasi;
- parametri **valore** di qualche tipo, ossia si possono istanziare con un valore costante del tipo indicato;

Va ricordato che un template non é **compilabile**, ma puó essere istanziato in modo implicito o esplicito. Se implicito i tipi del template sono dedotti, ma non viene dedotto il tipo di ritorno dell'istanza.
L'algoritmo di deduzione procede esaminando i parametri attuali passati da sinistra a destra, se trova uno stesso parametro T del template che appare piú 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 perché si deducono due diversi argomenti
}
```

Sono comunque ammesse le seguenti conversioni in fase di deduzione:

```
1) conversione l-value (T&) -> r-value (T)
2) conversione da array (T[]) a puntatore (T*)
3) conversione di qualificazione costante (T a const T)
4) conversione da r-value (T) a riferimento costante (const T&)
```

L'istanziazione esplicita degli argomenti dei parametri del template di funzione permette l'applicazione di qualsiasi conversione di tipo implicita.

4.2 Modelli di compilazione dei template di funzione

Il C++ standard prevede due modelli di compilazione dei template: **per inclusione** e **per separazione**.

4.2.1 Compilazione per inclusione

Il template viene definito in un header file che deve essere **sempre incluso** dal codice che necessitá di istanziare tale template. Saltano all'occhio due problemi:

- i dettagli di definizione del template sono visibili all'utente, pertanto si perde il principio dell'information hiding;
- se la stessa istanza di template viene usata in piú file compilati separatamente il compilatore genera del codice ripetuto;

Se al primo problema non c'é soluzione, per risolvere il secondo dobbiamo forzare il compilatore a generare le istanze del template usate nel programma con delle **dichiarazioni esplicite di istanziazione**:

```
template <class T>
T min(T a, T b) { return a < b ? a : b; }
// Genera una dichiarazione esplicita del tipo int
template int min(int, int);
// Per forzare il compilatore usare il flag
// -fno-implicit-template
```

4.2.2 Compilazione per separazione

La dichiarazione del template viene separata dalla sua dichiarazione:

```
// file "min.h"
template <class T>
T min(T a, T b);

// file "min.cpp"
export template <class T> // Necessaria keyword export
T min(T a, T b) { return a < b ? a : b; }
```

4.3 Template di classe

Un **template di classe** é essenzialmente un modello che il compilatore puó usare per generare automaticamente istanze particolari di una classe che differiscono per il tipo di alcuni membri. Prendiamo come esempio la classe Queue:

```
template <class T>
class Queue {
    private:
    ...

    public:
    Queue();
    ~Queue();
    bool is_empty() const;
    void add(const T&);
    T remove();
}
```

Nel template di classe posso avere due tipi di parametri:

- parametri di **tipo**, sono preceduti dalla keyword class o typename;
- parametri di **valore**, sono preceduti dal tipo del valore;

Entrambi i parametri possono avere valori di default.
La classe Queue si appoggia alla classe QueueItem:

```
template <class T>
// Per ore tutto esterno e public
class QueueItem {
    public:
    T info;
    QueueItem* next;
    QueueItem(const T&);
}

template <class T>
class Queue {
    private:
    QueueItem<T>* primo:
    QueueItem<T>* ultimo:

    public:
    Queue();
    ~Queue();
    bool is_empty() const;
    void add(const T&);
    T remove();
}
```

4.3.1 Istanziazione di un template di classe

Nella dichiarazione o definizione di un template 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
```

4.3.2 Metodi di template di classe

In un template di classe é possibile definire un metodo sia in maniera **inline** sia in maniera **esterna**:

```
template <class T>
class Queue {
    ...

    public:
    Queue() : primo(0), ultimo(0) {}
}

// Definizione esterna
template <class T>
Queue<T>::Queue() : primo(0), ultimo(0) {}
```

Un metodo di un template di classe é un **template di funzione**, ma esso non viene istanziato quando viene istanziata la classe ma se e solo se il programma usa effettivamente tale metodo.

4.3.3 Dichiarazioni friend in template di classe

In un template di classe possono comparire tre tipi di dichiarazioni di amicizia:

- 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();
  }
  ```

  In questo caso tutte i campi marcati friend sono amici di **tutte le istanze** del template di classe C.

- dichiarazione nel template di classe C di un template (di classe o funzione) friend **associato**, ossai avente tra i suoi parametri alcuni dei parametri del template C:

```
template <class T> class A { ... int fun(); ... };
template <class T> class B { ... };
// Dichiarazione incompleta del template di classe C
template <class T1, class T2> class C;
// Dichiarazione del template di test associato a C
template <class T1, class T2> bool test(C<T1, T2>);

template <class T1, class T2>
class C {
    friend int A<T1>::fun();
    friend class B<T2>;
    friend bool test<T1, T2>(C);
};
```

  Ad ogni istanza del template C rimane associata una ed una sola istanza del template B, di fun() e test.

- dichiarazione nel template di classe C di un template (di classe o funzione) friend **non associato**, cioé aventi i parametri disguinti dai parametri di C. In questo caso i template dichiarati friend sono amici di ogni istanza del template di C:

```
template <class T>
class C {
    T t;

    template<class Tp>
    friend int A<Tp>::fun();

    template<class Tp>
    friend class B;

    template<class Tp>
    friend bool test(C<Tp>);
}
```

4.3.4 Membri statici in template di classe

Anche in un template di classe possono essere dichiarati campi e metodi statici. In tal caso ogni istanza del template di classe ha dei proprio campi dati e metodi statici:

```
template <class T>
class Queue {
```

```
    private:
    static int contatore;
    ...
};
```

Ad ogni istanza di Queue é associato un campo statico contatore diverso.

4.3.5 Template di classe annidati

All'interno di un template possono essere dichiarati altri template di classe **annidati**, sia associati che non. Ad esempio per impedire all'utente di usare direttamente il template QueueItem possiamo fare:

```
template <class T>
class Queue {
    private:
    // Template iomplicito di classe annidato associato
    class QueueItem {
        public:
        QueueItem(const T& val);
        T info;
        QueueItem* next;
    };
    ...
};
```

4.3.6 Tipi e template impliciti in template di classe

L'uso di tipi e template che dipondono da un parametro di tipo devono essere **disambiguati** tramitela keyword **typename** per i tipi e **template** per i template: non é quindi permesso l'uso implicito di tipi e template che dipendono da parametro di tipo

```
template <class T>
class C {
    public:
    class D {
        public:
        T x;
    };

    template <class U>
    class E {
        public:
        T x;
        void fun1() { return; }
    };
```

```
    template <class U>
    void fun2() {
        T x; return;
    }
};

// Modo d'uso
template <class T>
// C<T>::D é un uso di un tipo che effettivamente
// dipende dal parametro T
void templateFun(typename C<T>::D d) {
    // C<T>::D é un uso di un tipo che comunque
    // dipende dal parametro T
    typename C<T>::X x;

    // C<T>::D é un uso di un tipo che effettivamente
    // dipende dal parametro T
    typename C<T>::D d2 = d;

    // E<int> é un uso del template di classe annidata
    // che dipende dal parametro T
    // C<T>::E<int> é un uso di un tipo che
    // dipende dal parametro T
    typename C<T>::templateE<int> e;
    e.fun1();

    // c.fun2<int> é un uso del template di funzione
    // che dipende dal parametro T
    C<T> c;
    c.template fun2<int>();
}
```

5 Contenitori della STL

I template caratterizzano una serie di classi **container** e i loro relativi **iteratori** contenuti nella **Standard Template Library**, che possiede quattro componenti:

- algoritmi (template di funzione);
- containers (template di classe);
- funzioni;
- iteratori;

5.1 Il container vector

La classe **vector** é uguale agli array dinamici e le sue caratteristiche principali sono:

- il supporto per l'accesso **casuale** agli elementi (accesso arbitrario in O(1));
- inserimento e rimozione in **coda** in tempo **costante ammortizzato**;
- inserimento e rimozione **arbitraria** in tempo **lineare ammortizzato**;
- dimensione **variabile** in modo dinamico;
- gestione della memoria **automatica**;

I principali metodi per scorrere vector sono:

```
// I vari metodi ritornano:

// iteratore che punta al primo item di vector
vector.begin()
// iteratore che punta all'item teorico successivo
// all'ultimo item
vector.end()
// iteratore inverso che punta all'ultimo item di
// vector e si sposta verso il primo
vector.rbegin()
// iteratore inverso che punta all'item teorico
// che precede il primo item
vector.rend()
// iteratore costante che punta al primo item di vector
vector.cbegin()
// iteratore costante che punta all'item teorico
// che segue l'ultimo item
vector.cend()
// iteratore costante inverso che punta all'ultimo
// item e si sposta verso il primo
vector.crbegin()
// iteratore costante inverso che punta all'item
// teorico che precede il primo item
vector.crend()
```

Ci sono due modi di usare un vector:

```
// Stile array di C
int a[10];
// Stile STL di C++
vector<int> v(10);
// Per accedere agli elementi con operator[]
```

```
int n = 5;
vector <int> v(n);
int a[5] = {2, 4, 5, 2, -2};
for (int i = 0; i < n; i++) {
    v[i] = a[i] + 1;
}
// Costruzione e assegnazione di copia
vector<string> v(10), w;
cout << v.size() << " " << v.capacity(); // 10 10
vector<string> u(v);
w = u;
```

Il metodo size ritorna il numero di **elementi contenuti in vector** mentre il metodo capacity ritorna la **capacitá del vector** (quindi v.size <= v.capacity). La classe vector possiede due costruttori:

```
// Costruttore vector(size_type): gli item sono inizializzati
// con il costruttore di default
// Costruttore vector(size_type n, const T& t): specifica
// un valore iniziale t da cui sono costruiti in copia
// tutti gli item
```

É possibile inizializzare un vector con una lista di elementi a partire da C++11; I metodi fondamentali di vector sono:

```
 // Inserisce un item in coda con il costruttore di copia
void push_back(const T&)
void pop_back();
T& front();
T& back();
// Iteratori di vector
iterator begin();
iterator end();
```

Ogni classe contenitore C della STL sono associati due tipi di iteratori:

```
// Usato se necessito di effettuare operazioni di R/W
C::iterator
// Se non devo fare W meglio usare quello costante
C::const_iterator
// Tutti gli iteratori offrono le funzionalitá seguenti
Cont<T> x;
Cont<T>::[const_]iterator i;

x.begin(); x.end();
*i // Elemento puntato da i
i++; ++i; // Puntatore all'item successivo
i--; --i; // Puntatore all'item precedente
```

Gli iteratori di vector e deque (contenitori ad **accesso casuale**) permettono di avanzare o retrocedere di un numero arbitrario di item in tempo O(1) (iteratori ad **accesso casuale**). Possiamo inizializzare un vector con un segmento di un array o vector tramite il **template di costruttore**:

```
int main() {
    int ia[20];
    // vector<T> name(start_incluso, end_escluso);
    vector<int> iv(ia, ia + 6);
    cout << iv.size() << endl; // Stampa 6
    vector<int> iv2(iv.begin(), iv.end() - 2);
    cout << iv2.size() << endl; // Stampa 4
}
```

Altri metodi di vector:

```
vector<int> v1, v2;
vector<int>::iterator i;
...
v1.push_back(1); // Inserisce 1 in coda a v1
v2.insert(i, 2); // Inserisce 2 subito prima di *i
v2.insert(i, 5, 10); // Inserisce 5 volte 10 subito prima di *i
v1.pop_back(); // Toglie l'ultimo elemento
```

Su tutti i contenitori sono definiti gli operatori di confronto. L'overloading dell'operator[] é disponibile solo per **vector** e **deque**.

5.2 Il container list

List é un contenitore di sequenze che consente l'allocazione di memoria **non contigua** (a differenza di vector). É implementata come una **doubly-linked list** e quindi é molto efficente nelle operazioni di inserimento e rimozione in posizione arbitraria, ma soffre nell'accesso agli elementi (deve scorrere **tutta la list**).
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.

5.3 Il container deque

Si tratta di una **Double Ended Queue**, ovvero una coda a due estremi, ed offre accesso indicizzato efficiente per **lettura/scrittura** e inserimento/eliminazione agli **estremi** efficiente.

5.4 I container associativi

Essi permettono di accedere ad un elemento mediante il **valore** dell'elemento stesso o una **parte** di esso, cioé la **chiave di accesso** all'elemento. Le chiavi

sono mantenute **ordinate** e quindi il tipo della chiave deve supportare gli **operatori di confronto**.
Alcuni esempi di tali contenitori sono:

- set e multiset;
- map e multimap;

5.5 Algoritmi della STL

La STL mette a disposizione diversi algoritmi efficienti per i suoi container quali: sort(), count(), copy(), reverse(), max_element(), binary_search(), ...
L'algoritmo di sort garantisce una complessitá media e pessima di O(N logN).

6 Ereditarietá

6.1 Le sottoclassi

Supponiamo di voler modellare il concetto di "orario con data": possiamo riutilizzare la nostra vecchia classe orario:

```
classe orario {
    private:
    int sec;

    public:
    orario(int o = 0, int m = 0, int s = 0);
    int Ore() const;
    int Minuti() const;
    int Sec() const;
    orario operator+(const orario&) const;
    bool operator==(const orario&) const;
    bool operator<(const orario&) const;
    friend ostream& operator<<(ostream&, const orario&);
};
```

Dichiariamo la classe dataora derivata da orario:

```
classe dataora: public orario {
    private:
    int giorno, mese, anno;

    public:
    int Giorno() const;
    int Mese() const;
    int Anno() const;
};
```

La classe dataora si dice **classe derivata/figlio** e la classe orario si dice **classe base/genitore**. La classe derivata eredita tutte le proprietá della classe base senza modificarle e puó aggiungere nuove caratteristiche alla propria.
La relazione che si instaura (**is-a**) induce il fenomeno del **subtyping** (forma di polimorfismo): ogni oggetto della classe derivata é utilizzabile **anche** come oggetto della classe base.
A seconda del contesto di invocazione, il puntatore si riferisce al **supertipo** o al **sottotipo**.

```
// Subtyping: Sottotipo -> Supertipo
D -> B
// Per oggetti si "estrae" il sottooggetto
D -> B
// Subtyping per puntatori/riferimenti
D* -> B*
D& -> B&
```

Il sottotipo puó essere:

- **diretto**, se deriva direttamente da una classe;
- **indiretto**, se deriva in modo transitivo da una classe;

I casi d'uso dell'ereditarietá sono i seguenti (-: eredita da):

- per **estensione**: dataora -: orario;
- per **specializzazione**: QButton -: QComponent;
- per **ridefinizione**: Queue -: List;
- per **riutilizzo del codice** (non é subtyping);

Nel caso di dataorasi memorizza il giorno della settimana con enum:

```
enum giorno {lun, mar, mer, gio, ven, sab, dom};
class dataorasett : public dataora {
    private:
    giorno giornosettimana;

    public:
    giorno GiornoSettimana() const;
};

// GERARCHIA CLASSI: orario <- dataora <- dataorasett
// Tra una classe B e suo sottotipo D valgono le conversioni:
D -> B // Oggetti
D& -> B& // Riferimenti
D* -> B* // Puntatori
```

```
// Quindi vale la seguente conversione
int F(orario o) {...}
dataora d; int i = F(d);

// Non vale il contrario
int G(dataora o) {...}
orario o; int i = G(o); // Illegale
```

6.1.1 Tipo statico e tipo dinamico

Presa una sottoclasse D di B:

```
D d; B b;
D* pd = &d;
B* pb = &b;
// Conversione D* -> B*
pb = pd;
```

Il **tipo statico** di un puntatore p é il tipo T* di **dichiarazione** di p, mentre se in un certo istante dell'esecuzione il tipo dell'oggetto a cui effetivamente punta p é U allora in quell'istante U* é il **tipo dinamico** di p. Ad esempio:

```
D d; B b;
D& rd = d;
// Conversione D& -> B&
B& rb = d;
```

Ci sono due metodi di controllo dei tipi:

- **statico**, avviene in fase di compilazione e non sono utilizzate informazioni disponibili in fase di esecuzione;
- **dinamico**, avviene quando le informazioni sui tipi vengono utilizzate in fase di esecuzione;

6.1.2 Accessibilitá

Se aggiungiamo alla classe dataora un metodo Set2K() non possiamo fare questo:

```
dataora::set2K() {
    sec = 0; // Illegale, parte privata della classe base é
             // inaccessibile alla classe derivata
    giorno = 1; mese = 1;
    anno = 2000;
}
```

Esistono vari tipi di ereditarietá, che seguono i modificatori di accesso:

- **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 lo sono dalle classi derivate (é comunque una violazione dell'**information hiding**);

Le derivazione **protette** e **private** non supportano l'ereditarietá di tipo.
Confronto tra ereditarietá privata e composizione has-a (esempio classe Motote e Auto):

- in entrambi i casi un oggetto Motore é contenuto in ogni oggetto Auto;
- in entrambi i casi per gli utenti esterni Auto* non é convertibile a Motore*;
- la composizione é necessaria se servono piú motori in un auto;
- ereditarietá privata puó introdurre il problema dell'ereditarietá **multipla**;
- ereditarietá privata permette ad Auto di convertire Auto* in Motore*;
- ereditarietá privata permete 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**.
Si ricorda che **non si ereditano le amicizie**, e alcuni oggetti possono essere inaccessibili a seconda del contesto.

6.2 Ridefinizione di metodi

Proviamo a ridefinire l'operator+ per la classe dataora:

```
dataora dataora::operator+(const orario& o) const {
    dataora aux = *this;
    // aux.sec = sec + o.sec da errore perché sec é
    // inaccessibile in dataora anche se protected
    aux.sec = sec + 4600*o.Ore() + 60*o.Minuti() + o.Secondi();
    if(aux.sec >= 86400) {
        aux.sec = aux.sec - 86400;
        aux.AvanzaUnGiorno();
    }
    return aux;
}

int main() {
    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+
}
```

Illegalitá causata dalla **name hiding rule**: una ridefinizone in D del noem di 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 B::.

6.3 Ridefinizione di campi dati

Con la stessa logica possiamo ridefinire i **campi dati**, in cui per ereditarietá 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; } // D::x
    void printAll() { cout << B::x << ' ' << x << endl; }
};

int main() {
    B b; D d;
    b.print();      // Stampa 2
    d.print();      // Stampa 3.14
    d.printAll();   // Stampa 2 3.14
}
```

6.4 Static binding

Il binding statico avviene quando tutte le informazioni necessarie per chiamare una funzione sono disponibili al momento della compilazionee può essere ottenuto utilizzando le normali chiamate di funzione, l'overloading delle funzioni e l'overloading degli operatori.
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 B {
    int x;

    public:
    void f() { x = 2; }
};

class D : public B {
    int y;

    public:
    void f() { B::f(); y = 3; } // Ridefinizione
};

int main() {
    B b; D d;
    B* p = &b;
    p->f(); // Invoca B::f()
    p = &d;
    p->f(); // Invoca B::f()
}
```

6.5 Costruttori, assegnazione e distruttori

La lista di inizializzazione di un costruttore di una classe D derivata **direttamente** da B in generale puó contenere invocazioni di **costruttori** per i campi dati di **D** e l'invocazione di un costruttore della classe base di **B**.
L'esecuzione di un tale costruttore di D avviene come:

- viene sempre invocato per primo un costruttore della classe **base**, implicitamente o esplicitamente il costruttore di default di B quando la lista di inizializzazione **non include una invocazione specifica**;
- successivamente viene eseguito il costruttore di D, ossia vengono costruiti i **campi dati propri di D**;
- infine viene eseguito il corpo del costruttore;

In particolare se nella classe derivata si omette **qualsiasi costruttore**, allora é disponibile il costruttore di default standard di D, il cui comportamento é:

- richiamare il costruttore di **default** di B;
- richiama i costruttori di default per tutti i campi dati di D;

Quindi nel nostro esempio:

```
dataora::dataora(int a, int me, int g, int o, int m, int s)
            : orario(o, m, s), giorno(g), mese(me), anno(a) {}
```

Il costruttore di copia standard di una classe D derivata direttamente da una classe base B invocato su un oggetto x di D:

- costruisce il sottooggetto di B invocando il costruttore di copia **standard** o **ridefinito** di B sul corrispondente sottooggetto di x;
- costruisce ordinatamente i campi dati propri di D invocando i **relativi** costruttori di copia;

L'assegnazione standard di una classe D derivata direttamente da una classe base B:

- invoca preventivamente l'assegnazione (**standard** o **ridefinita**) della classe B sul sottooggetto;
- esegue l'assegnazione ordinatamente membro a membro dei campi dati propri di D invocando le rispettive assegnazioni;

Il distruttore standard di una classe D derivata direttamente da una classe base B richiama implicitamente il distruttore (**standard** o **ridefinito**) di B per distruggere il sottooggetto di B soltanto dopo l'azione di distruzione standard propria di D, cioé la distruzione dei campi propri di D nell'ordine inverso alla loro costruzione tramite l'invocazione dei corrispondenti distruttori. Se il distruttore viene ridefinito in D allora innanzitutto viene eseguito il codice di tale distruttore, quindi avviene la distruzione dei campi di D, quindi viene chiamato il distruttore di B.

6.6 Ereditarietá e template

Sia la classe base che la classe derivata possono essere definite con template di classe:

- classe **base template** e classe **derivata da una istanza** della classe base: ogni oggetto della classe derivata contiene come sottooggetto un oggetto dell'istanza della classe template;
- classe **base non template** e classe **derivata template**: ogni oggetto di ogni istanza della classe template derivata contiene come sottooggetto un oggetto della classe base;

- classe **base e derivata template**: i parametri della classe template derivata devono essere un sovrainsieme di quelli della classe base (derivazione **associata**). Ogni oggetto di una istanza della classe template derivata contiene come sottooggetto un oggetto dell'istanza associata della classe template base;

7 Polimorfismo

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. Ad esempio se abbiamo:

```
// Voglio eseguire il metodo Stampa() in modo polimorfo
orario::Stampa() { ... }
dataora::Stampa() { ... }

void printInfo(const orario& r) { r.Stampa(); }
void printInfo(const orario* p) { p->Stampa(); }
```

In questo caso si usa la keyword **virtual**, che permette di far capire al compilatore a quale classe ci stiamo riferendo.

7.1 Metodi virtuali

Una funzione virtuale é una funzione membro dichiarata in una classe **base** e **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 regole delle funzioni virtuali sono:

- non possono essere **statiche**;
- possono essere **friend** di un'altra classe;
- devono essere accessibili utilizzando un **puntatore** o un **riferimento** al tipo di classe **base** per ottenere il polimorfismo a run-time;
- il loro prototipo deve essere **uguale** sia nella classe base che in quella derivata;
- sono **definite** nella classe **base** e sovrascritte in quella derivata (non é obbligatorio che la classe derivata la sovrascriva; in tal caso si utilizza la versione base);
- una classe puó avere **distruttore** virtuale ma non il **costruttore**;

7.1.1 Dynamic binding

La keyword virtual realizza il cosiddetto **dynamic binding**, che si verifica quando un puntatore o un riferimento viene associato a una funzione membro in base al tipo dinamico dell'oggetto.

7.1.2 La vtable

Per ogni classe C che contiene almeno un metodo **virtuale** il compilatore crea una corrispondente **tabella** contenente gli indirizzi dei metodi virtuali di C (puntatori a funzione) detta **vtable**. Per ogni oggetto di classe C il compilatore include un puntatore a funzione alla vtable di C. La selezione a run-time di quale metodo invocare in una chiamata polimorfa avviene seguendo tali strutture aggiuntive di puntatori.

7.1.3 Overriding

L'**overriding** consente di definire una funzione con stesso **nome** e **firma** sia nella classe base che nella classe derivata con una definizione di funzione diversa. L'override di funzione è un'implementazione del polimorfismo in tempo reale. Pertanto, sovrascrive la funzione in fase di esecuzione del programma.

7.1.4 Distruttori virtuali

Considerando il seguente codice:

```
D* pd = new D;
B* pb = pd;
delete pb;
```

La chiamata delete pb invoca il distruttore di B su un oggetto di tipo D. Dichiarando il distruttore della classe base B come **virtuale** tutti i distruttori delle classi derivate da B diventano virtuali, permettendo al compilatore di decidere a run-time quale distruttore invocare.

7.1.5 Metodi virtuali puri

Potremmo voler creare una classe A di cui non vogliamo mai istanziare oggetti, ma che vogliamo usare come classe base per successive classi derivate. In tal caso A si limita a specificare la lista di operazioni basilari che caratterizzano i suoi sottotipi. Possiamo evitare di includere il corpo dei metodi di A rendendoli **metodi virtuali puri**:

```
class A {
    ...
    // = 0 indica un metodo virtuale puro
    virtual void G() = 0;
    ...
}
```

Se una classe contiene o eredita un metodo virtuale puro essa si dice **classe astratta**.

7.2 Run-Time Type Identification

L'**RTTI** é un meccanismo che espone informazioni sul tipo di dati di un oggetto in fase di esecuzione ed é disponibile solo per le classi che hanno **almeno una funzione virtual**.

7.2.1 Run-time cast

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**. Ne esistono di due tipologie:

- **upcasting**: un puntatore/riferimento di un oggetto di classe **derivata** viene trattato come puntatore alla classe **base**;
- **downcasting**: un puntatore/riferimento di classe **base** viene convertito in un puntatore di classe **derivata**;

L'operatore per eccellenza in questi casi é il **typeid**, che ha come argomento un'espressione o un tipo qualsiasi e ritorna un oggetto della classe **type_info**. Il comportamento di typeid é il seguente:

- se l'espressione operando di typeid é un riferimento ref ad una classe con **almeno un metodo virtuale**, allora typeid restituisce un oggetto type_info che rappresenta il tipo dinamico di ref;
- se l'espressione operando di typeid é un puntatore **dereferenziato** *punt (che punta a un tipo polimorfo), allora typeid restituisce un oggetto type_info che rappresenta il tipo T dove T* é il tipo dinamico di punt;

Ci sono due casi speciali:

- se la classe **non contiene** metodi virtuali typeid ritona il tipo statico del riferimento o del puntatore dereferenziato;
- typeid su un puntatore (**non dereferenziato**) restituisce sempre il tipo statico del puntatore;

7.3 Dynamic cast e polimorfismo

In una gerarchia di ereditarietà, viene utilizzato per il **downcasting** di un puntatore di classe base a una classe derivata. 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:

```
class B {
    public:
    virtual void m();
};

class D : public B {
    public:
    virtual void f();
};

class E : public D {
    void g();
};

B* fun() { // Puó ritornare B*, D*, E*, ... }

int main() {
    b* p = fun();
    // Downcast é possibile
    if (dynamic_cast<D*>(p)) {
        // Effettuo il downcast
        (static_cast<D*>(p))->f();
    }
    E* q = dynamic_cast<E*>(p);
    if (q) {
        q->g();
    }
}
```

Importante notare le seguenti buone pratiche:

- usare il downcasting **solo quando necessario**;
- **non fare** type checking dinamico inutile;
- usare il piú possibile **metodi virtuali nelle classi base** al posto di fare type checking;

8 Il principio S.O.L.I.D

Il principio S.O.L.I.D é un acromino dei cinque principi della OOP:

- **Single responsability**: una classe dovrebbe avere una sola responsabilitá, ossia solo **un motivo** per dover essere **modificata**;
- **Open-Closed**: le classi dovrebbero essere **aperte** per estensioni ma **chiuse** per modifiche, ossia poter aggiungere **nuove funzionalitá** senza **modificare** il codice esistente (ereditarietá e polimorfismo);

- **Liskov subsitution**: un oggetto di una classe **derivata** deve poter essere **sostituito** con un oggetto della classe **base** senza **alterare** il comportamento del programma;
- **Interface segregation**: le classi non dovrebbero essere **costrette** a implementare interfacce che **non utilizzano**, si dovrebbero suddividere in interfacce **piú piccole** e **specifiche**;
- **Dependency inversion**: le classi dovrebbero dipendere da **astrazioni**, non da **implementazioni concrete**;

9 Ereditarietá multipla e gestione virtual

9.1 Ereditarietá multipla

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**. Prendiamo il seguente esempio:

```
class data {
    private:
    int GiorniDelMese() const;
    bool Bisestile() const;

    protected:
    int giorno, mese, anno;
    void AvanzaUnGiorno();

    public:
    data(int = 1, int = 1, int = 0);
    int Giorno() const { return giorno; }
    int Mese() const { return mese; }
    int Anno() const { return anno; }
};

class orario {
    ...
};

// -------------------------------------
#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;
    ...
};
```

Siccome abbiamo un metodo Stampa() in orario e in data, la classe dataora eredita due metodi diversi con stesso **nome** e **segnatura**: problema di **ambiguitá** (anche con segnature diverse). Per risolvere il problema possiamo usare l'operatore di **scoping** oppure dobbiamo **ridefinire** Stampa() in dataora in modo da **nasconderla** nelle classi padre.

9.1.1 Ereditarietá a diamante

Il problema dell'ereditarietá a diamante si verifica quando una classe figlio eredita da **due** classi genitore che **condividono** una classe **nonno** comune. Ad esempio:

```
class A {
    protected:
    int x;

    public:
    A(int y = 0) : x(y) {}
    virtual void print() = 0;
};

class B : public A {
    public:
    B() : A(1) {}
    virtual void print() { cout << x; }
};

class C : public A {
    public:
    C() : A(2) {}
    virtual void print() { cout << x; }
};

class D : public B, public C {
    public:
    // Overriding: di quale x?
    virtual void print() { cout << x; }
};

int main() {
    D d;
```

```
    A* p = &d; // Errore: "A is an ambiguous base of D"
    p->print(); // Chiamata polimorfa illegale
}
```

Per risolvere l'ereditarietà a diamante, usiamo la **derivazione virtuale**. Basterà mettere la parola virtual davanti alla keyword modificatrice di accesso.

```
class A {
    ... // Classe base virtuale
};

class B : virtual public A {
    ...
};

class C : virtual public A {
    ...
};

class D : public B, public C {
    ...
};
```

9.1.2 Unique final overrider

La keyword **override** ha due scopi:

- mostra al programmatore che questo é un metodo **virtuale** che sta sovrascrivendo un metodo virtuale della **classe base**;
- il compilatore **sa** che si tratta di un override, quindi puó controllare che non si stiano **modificando** o **aggiungendo** nuovi metodi che si pensa siano override;

Il problema sorge in questi casi:

```
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 {
    public:
    // Omettendo override il compilatore solleva errore
    // perché la vtable di D deve avere un indirizzo
    // per la entry di print()
    void print() override { cout << "D"; }
};

int main() {
    D d;
    A* p = &d;
    p->print(); // Stampa D
}
```

9.1.3 Derivazione privata e pubblica virtuali

Anche per la derivazione virtuale multipla possiamo avere derivazione **privata**, **pubblica** o **protetta**. Vale la seguente regola:

- derivazione **protetta** prevale su quella **privata**;
- la derivazione **pubblica** prevale su quella **protetta**;

```
class A {
    public:
    void f() { cout << "A"; }
};

class B: virtual private A {};
class C : virtual public A {};
class D : public B, public C {}; // Prevale derivazione pubblica

int main() {
    D d;
    d.f(); // Stampa A ma chiamerebbe C::f()
    d.B::f(); // Illegale poiché A::f() é inaccessibile
}
```

9.2 Costruttori in presenza di basi virtuali

Il loro comportamento é il seguente:

- prima vengono richiamati, **solo una volta**, i costruttori delle classi **base virtuali** che si trovano nella gerarchia di derivazione di D (la ricerca delle classi base avviene in ordine da sx a dx e dall'alto al basso);

- successivamente vengono richiamati i costruttori delle **superclassi dirette** non virtuali di D ed **escludono** di richiamare eventuali costruttori di classi virtuali **giá chiamati**;
- infine viene eseguito il **costruttore proprio** di D (costruiti i campi di D ed eseguito il corpo del suo costruttore);

Le chiamate dei primi due punti, se non sono **esplicite**, vengono automaticamente inserite nella lista di inizializzazione del costruttore di D (chiamate implicite ai costruttori di default).

10 Le classi di I/O

La classe base per le funzioni di I/O é **ios_base**: tale classe usa un'astrazione dei dispositivi di I/O detto **stream**.
Uno stream é **una seuqenza non limitata di celle ciascuna contenete un byte** con queste caratteristiche:

- la posizione delle celle parte da **0**;
- l'I/O effettivo avviene attraverso un **buffer** associato;

10.1 Gerarchia delle classi di I/O

10.1.1 Classe ios

La classe ios deriva da ios_base ed é la **classe base astratta virtuale** della gerarchia che permette di controllare lo stato di funzionamento di uno stream. Lo stream puó trovarsi in **8** stati di funzionamento diversi: viene rappresentato con un **int** in [0, 7[nella classe ios:

```
class ios : public ios_base {
    int state;
    public:
    // 000 - 001 - 010 - 100
    enum io_state {goodbit = 0, eofbit = 1, failbit = 2, badbit = 4};
    int good() const;
    // Stream in posizione end-of-file
    int eof() const;
    // Operazione su stream fallita ma senza perdita di dati
    int fail() const;
    // Operazione su stream fallita con perdita di dati
    int bad() const;
    // Ritorna lo stato come int in [0, 7[
    int rdstate() const;
    // Imposta lo stato (di default a goodbit)
    void clear(int i = 0);
    ...
};
```

10.1.2 Classe istream

Gli oggetti della sottoclasse **istream** rappresentano stream di **input**: l'oggetto cin appartiene a questa classe:

```
class istream : public virtual ios {
    public:
    // Metodi di overload di >> 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** ossia spazi, tab ed enter. L'overloading dell'operatore di input equivale ad eseguire il **parsing** di una sequenza di byte secondo le regole sintattiche del linguaggio.

10.1.3 Classe ostream

Gli oggetti della sottoclasse **ostream** rappresentano stream di **output**: l'oggetto cout e cerr appartengono a questa classe:

```
class ostream : public virtual ios {
    public:
    // Metodi di overload di << con ostream di invocazione
    ostream& operator<<(bool);
    ostream& operator<<(int);
    ostream& operator<<(double);
    ...
};
// Funzioni esterne in std::
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 immessi nelle celle dell'ostream di invocazione.

10.2 I/O binario

L'input binario da uno istream (byte per byte **senza interpretazione**) puó essere fatto tramite alcuni metodi di istream:

```
class istream : public virtual ios {
    public:
    // Preleva un byte e lo converte in intero [0, 255]
```

```
    // Se si legge EOF ritorna -1
    int get();
    // Memorizza il carattere prelevato in c
    istream& get(char& c);
    // Preleva dall'istream di invocazione n caratteri
    // a meno di trovare EOF e li memorizza in p
    istream& read(char* p, int n);
    // Preleva n caratteri ma non li memorizza
    istream& ignore(int n = 1, int e = EOF);
};
```

L'output binario su uno ostream puó essere fatto tramite alcuni metodi di di ostream:

```
class ostream : public virtual ios {
    public:
    // Scrive il carattere c nell'ostream di invocazione
    ostream& put(char c);
    // Scrive nell'ostream di invocazione i primi n caratteri
    // della stringa puntata da p
    ostream& write(const char* p, int n);
};
```

10.3 Stream di file

Gli stream associati a file sono oggetti delle classi **ifstream**, **ofstream** e **fstream**. La classe base ios specifica diverse modalitá di apertura di file in un enum. Tali modalitá sono combinabili tramite **OR**: per default ifstream apre gli oggetti in **lettura**, ofstream li apre in **scrittura** e fstream puó fare **entrambi**.
Il metodo **close** chiude esplicitamente un file: viene invocanto automaticamente dal **distruttore** dello stream.

10.4 Stream di stringhe

Si possono definire stream associato a **stringhe** (sequenze di caratteri memorizzati in RAM). Il caratere nullo di terminazione gioca il ruolo di marcatore di fine stream. Le classi da usare sono **istringstream**, **ostringstream** e **stringstream**. I metodi di lettura/scrittura sono **ereditati** da istream, ostream e iostream.

11 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. La gestione delle eccezioni in C++ si basa su tre parole chiave: **try**, **catch** e **throw**.

11.1 Throw e try/catch

La funzione in cui si verifica la situazione eccezionale **lancia** un'eccezione tramite la keyword **throw**:

```
telefonata bolletta::Estrai_Una() {
    // Lancia una eccezzione della classe Ecc_Vuota
    if (Vuota()) throw Ecc_Vuota();
    telefonata aux = first->info;
    first = first->next;
    return aux;
}
```

L'esecuzione del throw comporta la terminazione dell'esecuzione di Estrai_Una() con il lancio dell'eccezione Ecc_Vuota() (oggetto eccezione **anonimo** costruito con il costruttore di default di Ecc_Vuota) alla funzione chiamante.
Nella funzione chiamante il codice contenete la chiamata a funzione é racchiuso in un blocco **try** ed é seguito dalle clausole di **catch**, che gestiscono eventuali eccezioni sollevate. La funzione abort() (inclusa nella libreria standard) provoca la terminazione anormale del programma:

```
int main() {
    ...
    try {
        b.Estrai_Una();
    } catch (Ecc_Vuota e) {
        cerr << "La bolletta é vuota" << endl;
        abort();
    }
}
```

11.1.1 Flusso di controllo provocato da throw

Quando in una funzione F viene sollevata un'eccezione di tipo T tramite throw si inizia la ricerca della catch in grado di gestirla:

- se la throw si trova in un blocco **try** nel **corpo** di F, allora l'esecuzione **abbandona** il blocco try ed esamina in successione **tutte** le catch associate a tale blocco try;
- se si trova un **type match** per una catch l'eccezione viene **catturata** e viene eseguito il **corpo** di tale catch. Eventualmente al termine dell'esecuzione della catch il controllo torna al primo punto di programma che esegue l'ultimo blocco catch;
- se **non** si trova **type match**, oppure se la throw **non si trovava** in un blocco try della funzione F la ricerca continua nella funzione che ha **invocato** F;

- la ricerca **top-down** sullo **stack** continua fino a quando si trova una **catch giusta** o si arriva al main(), in tal caso viene richiamata la funzione terminate() che per **default** richiama la funzione abort() che termina il programma in **errore**;

11.2 Rilanciare un'eccezione

É possibile che una catch si accorga di non poter gestire direttamente un'eccezione. In tal caso essa puó **rilanciare** l'eccezione alla funzione chiamante con una throw.

11.3 Utilizzo delle risorse

In questo snippet di codice avviene un **problema** in caso si sollevi un'eccezione che **non** viene poi **catturata**:

```
gestore() {
    risorsa rs; // Allocazione di una risorsa
    ...
    // Codice che puó sollevare eccezioni
    ...
    rs.relase(); // Non eseguito in caso di eccezione
}
```

Se viene sollevata un'eccezione non gestita all'interno della funzione, si esce **senza rilasciare** la risorsa (se fosse memoria avremo lasciato del **garbage**).

11.4 Clausa catch generica

Una catch generica é in grado di catturare **tutte le eccezioni** possibili (viene quindi posta per ultima nella sequenza di blocchi catch):

```
gestore() try {
    risorsa rs; // Allocazione di una risorsa
    ...
    // Codice che puó sollevare eccezioni
    ...
    rs.relase(); // Non eseguito in caso di eccezione
} catch (...) { // Catch generica
    rs.release(); // Rilascio corretto della risorsa
    throw; // Rilancio eccezione al chiamante
}
```

11.4.1 Match del tipo di eccezione

La catch che cattura un'eccezione di tipo E é la prima catch incontrata durante la ricerca che abbia un tipo **T compatibile con E**. Le regole che definiscono la compatibilitá tra T ed E sono le seguenti:

- T é **uguale** ad E;
- T é **sottotipo** derivato pubblicamente da T;
- T é un **puntatore** B* ed E é un **puntatore** D* dove D é sottotipo di B;
- T é un **riferimento** B& ed E é un **riferimento** D& dove D é sottotipo di B;
- T é di tipo **void*** ed E un qualsiasi puntatore;

Non possono essere applicate, in questo contesto, conversioni implicite.

11.4.2 Comportamenti tipici di una catch

I comportamenti piú comuni di un blocco catch sono i seguenti:

- **rilanciare** le eccezioni;
- **convertire** un tipo di eccezione in un altro, **rimediando** parzialmente e **rilanciando** 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, **eliminare** la causa e **riprovare** a chiamare la funzione che ha **causato** l'eccezione;
- **esaminare** l'errore ed invocare std::terminate();

11.5 Specifica di eccezioni

La specifica esplicita delle eccezioni indica **quali** eccezioni puó sollevare una data funzione:

```
istream& operator>>(istream& is, orario& o)
throw(err_sint, fine_file, err_ore, err_min, err_sec) {
    ...
}
```

I problemi principali nella specifica delle eccezioni sono i seguenti:

- **run-time checking**, il test di conformitá delle eccezioni avviene a run-time e non a compile-time, quindi non vi é garanzia statica di conformitá;
- **run-time overhead**, il run-time checking richiede al compilatore codice addizionale che potrebbe compromettere alcune ottimizzazioni;
- **inusabilitá con template**, in generale i parametri di tipo del template non permettono di specificare le eccezioni;

11.6 Gerarchia delle eccezioni

In C++ la classe **exception** é la classe **base** da cui derivano **runtime_error** e **logic_error**. Da queste due classi ne derivano molte altre. Sempre da exception derivano le seguenti classi:

- **bad_cast**, lanciato dall'operatore di **dynamic_cast**;
- **bad_alloc()**, lanciata dalla **new** quando lo heap é esaurito;
- **bad_execption**;
- **bad_typeid**, lanciata dall'operatore **typeid** quando ha come argomento un puntatore nullo;

12 Standard C++11

12.1 Inferenza automatica di tipo

Da C++11 tramite la keyword **auto** possiamo dedurre il tipo di una variabile in base al **contesto di invocazione**:

```
auto x = 0;     // int
auto c = 'f';   // char
auto d = 0.7;   // double
auto y = qt_obj.qt_fun(); // y ha il tipo di ritono di qt_fun
```

La determinazione **statica** dei tipi avviene attraverso la keyword **decltype**:

```
int x = 3;
decltype(x) y = 4;
```

12.2 Inizializzazione inline

É possibile inizializzare **array** e **contenitori STL** come si faceva in C:

```
int* a = new int[3] {1, 2, 3};
std::vector<string> vs = {"pippo", "pluto", "topolino"};
```

12.3 Keyword default/delete

Per ogni classe sono disponibili le versioni standard di: **costruttore di default**, **costruttore di copia**, **assegnazione** e **distruttore**. Tali funzioni si possono rendere esplicitamente **di default** o **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; // Non disponibile
    NoCopy(const NoCopy&) = delete; // Non disponibile
};

int main() {
    NoCopy a,b;
    NoCopy b(a); // Errore in compilazione
    b = a; // Errore in compilazione
}
```

12.4 Keyword override

Per dichiarare esplicitamente quando si definisce un **overriding** di un metodo virtuale. Per dichiarare l'ultimo override si usa la keyword **final**:

```
class B {
    public:
    virtual void m(double) {}
    virtual void f(int) {}
};

class D : class B {
    public:
    virtual void m(int) override {} // Illegale, diversa segnatura
    virtual void f(int) override {} // Legale
};

// ---------------------------------------

class A {
    public:
    virtual void m(int) {}
};

class C : public A {
    public:
    virtual void m(int) final {} // Final overrider
};

class D : public C {
    public:
```

```
    virtual void m(int) {}; // Illegale
}
```

12.5 Il std::nullptr

La keyword **nullptr** sostituisce la macro NULL e il valore 0: essa ha come tipo **std::nullptr_t** che é convertibile **implicitamente** a qualsiasi **tipo di puntatore** e a **bool**, mentre non é convertibile implicitamente ai tipi primitivi integrali:

```
void f(int);
void f(char*);

int main() {
    f(0); // Invoca f(int)
    f(nullptr); // Invoca f(char*)
}
```

12.6 Chiamate di costruttori

Un costruttore nella sua lista di inizializzazione puó invocare un altro costruttore della stessa classe, un meccanismo noto come **delegation**. É un'alternativa al meccanismo degli argomenti di default:

```
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 w) : C(0, w) {}
};
```

12.7 Range based for loop

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

for(int& i : v) { // Per riferimento
    std::cout << ++i << ' ';
}

for(auto i : v) { // Per valore, rilevato automaticamente
    std::cout << i << ' ';
}

for(int n : {0, 1, 2, 3, 4, 5}) { // Ciclo direttamente su un litterale di interi
```

```
    std::cout << n << ' ';
}

int a[6] = {0, 1, 2, 3, 4, 5};

for(int n : a) { // Ciclo gli elementi dell'array in lettura
    std::cout << n << ' ';
}
```

12.8 Funtori

Un funtore é un oggetto di una classe che pu;o essere trattato come fosse una **funzione** (o un **puntatore a funzione**), é possibile farlo mediante overloading dell'operator():

```
class FunctorClass {
    private:
    int x;

    public:
    FunctorClass(int n) : x(n) {}
    int operator()(int y) const {return x + y;}
};

int main() {
    FuntorClass sommaCinque(5);
    cout << sommaCinque(6); // Stampa 11
}
```

12.9 Espressioni lambda

Anche detti **funtori anonimi** definite come:

```
[/*Capture list*/] (/*Lista parametri*/) ->/*Tipo di ritorno*/ { /*Corpo*/}
// Sono opzionali lista di parametri e tipo di ritorno (default void)

[] ->int {return 3*9;} // Lista vuota di parametri
[] (int x, int y) {return 3*9;} // Tipo di ritorno implicito int
[] (int x, int y) ->int {return 3*9;} // Tipo di ritorno esplicito int
[] (int& x) {++x;} // Tipo di ritorno implicito void
[] (int& x) ->void {++x;} // Tipo di ritorno esplicito void
```

La capture list elenca la lista delle variabili della closure, cioé variabili all'esterno della lambda espressione usate come **l-valore** (r/w) o **r-valore** (r-only) dalla lambda espressione:

```
[]          // Nessuna variabile esterna catturata
[x, &y]     // x per valore, y per riferimento
```

```
[&]          // tutte le variabili catturate per riferimento
[=]          // tutte le variabili catturate per valore
[&, x]       // tutte le variabili per riferimento, x per valore
```