

Francesco Ranzato
Università di Padova

**Appunti di
PROGRAMMAZIONE
AD OGGETTI**

II edizione



EDIZIONI LIBRERIA PROGETTO PADOVA

Indice

Prefazione	ii
1 Preliminari	1
1.1 Generalità sul linguaggio C++	1
1.2 Richiami di C++	2
1.2.1 Namespace	2
1.2.2 Argomenti di default	6
1.2.3 Puntatori a funzione	7
1.2.4 La keyword <code>typedef</code>	8
1.2.5 L'operatore virgola	8
1.2.6 Il tipo <code>string</code>	9
2 Classi e Oggetti	13
2.1 Tipi di dato astratti	13
2.2 Le classi	15
2.2.1 Il puntatore <code>this</code>	16
2.3 Parte privata e pubblica	18
2.4 Costruttori	19
2.4.1 Costruttori come convertitori di tipo	23
2.4.2 Operatori esplicativi di conversione	26
2.5 Metodi costanti	26
2.6 Campi dati e metodi statici	30
2.7 Overloading di operatori	32
2.7.1 Assegnazione standard	34
2.7.2 Costruttore di copia standard	34
2.7.3 Overloading di operatori con funzioni esterne	36
2.8 Incapsulamento e modularizzazione	40
2.8.1 Il preprocessore	41
2.8.2 Compilazione e linking	44
2.8.3 Il comando <code>make</code>	45
2.8.4 Modularizzazione delle classi	47

INDICE

2.9	Campi dati costanti	50
2.10	Liste di inizializzazione nei costruttori	53
3	Classi Collezione e Argomenti Correlati	57
3.1	Classi annidate	59
3.2	Il problema dell'interferenza	60
3.3	Copie profonde	63
3.3.1	Assegnazione profonda	63
3.3.2	Costruttore di copia profonda	65
3.4	Oggetti come parametri di funzione	66
3.5	Tempo di vita delle variabili	70
3.6	Distruttore	71
3.7	Nascondere la parte privata di una classe	78
3.8	Array di oggetti	80
3.9	Cast	81
3.10	Funzioni amiche	84
3.11	Classi amiche e iteratori	85
3.12	Dichiarazioni incomplete di classi	91
3.13	Condivisione controllata della memoria	92
3.13.1	Puntatori smart	109
3.14	Progetto di metà corso: Polinomi in una variabile	119
4	Template	123
4.1	Template di funzione	124
4.2	Modelli di compilazione dei template di funzione	127
4.2.1	Compilazione per inclusione	128
4.2.2	Compilazione per separazione	129
4.3	Template di classe	130
4.3.1	Istantiazione di un template di classe	133
4.3.2	Metodi di template di classe	134
4.3.3	Dichiarazioni friend in template di classe	137
4.3.4	Membri statici in template di classe	143
4.3.5	Template di classe annidati	144
4.3.6	Tipi e template impliciti in template di classe	145
4.4	Esempio di template di classe: alberi binari di ricerca	146
5	I Contenitori della Libreria STL	155
5.1	Classi contenitore	155
5.2	Iteratori	157
5.3	Sequenze	158
5.3.1	Sequenze ad accesso casuale	160
5.4	Contenitori associativi	160

INDICE

5.5	Contenitore <code>vector</code>	161
5.6	Altri contenitori	163
5.6.1	<code>list</code>	164
5.6.2	<code>deque</code>	164
5.6.3	<code>set</code> e <code>multiset</code>	165
5.6.4	<code>map</code> e <code>multimap</code>	166
6	Ereditarietà	169
6.1	Sottoclassi	169
6.1.1	Tipo statico e tipo dinamico	172
6.1.2	Gerarchie di classi	173
6.1.3	Accessibilità	173
6.2	Ridefinizione di metodi e campi dati	181
6.3	Costruttori, assegnazione e distruttore	190
6.4	Ereditarietà e template	200
6.5	Metodi virtuali	202
6.5.1	<code>vtable</code>	211
6.5.2	Distruttori virtuali	213
6.5.3	Metodi virtuali puri	214
6.6	Identificazione di tipi a run-time	219
6.7	Ereditarietà multipla	222
6.7.1	Derivazione virtuale	226
7	Eccezioni	241
7.1	<code>throw</code> e <code>try/catch</code>	241
7.1.1	Ricerca della clausola <code>catch</code>	246
7.1.2	<code>catch</code> generica	246
7.1.3	Rilanciare eccezioni	248
7.1.4	Specifiche di eccezioni	249
7.2	La gerarchia <code>exception</code>	250
8	La Gerarchia di Classi per l'I/O	251
9	Libreria Qt	261
9.1	Introduzione a Qt	262
9.2	Hello world	263
9.3	Segnali e slot	264
9.4	Layout manager	267
9.5	Connessioni tra segnali	268

INDICE

10 C++11	273
10.1 Lambda espressioni	273
10.2 Riferimenti rvalue	275
10.3 Inferenza automatica di tipo	277
10.4 Inizializzazione uniforme	278
10.5 <code>default</code> e <code>delete</code>	279
10.6 Overriding esplicito	280
10.7 <code>nullptr</code>	281
10.8 Chiamate di costruttori	282
11 Esercizi Riepilogativi	283
A Soluzioni degli Esercizi	
A.1 Esercizi del Capitolo 11	299
A.2 Altri esercizi del testo	314

Capitolo 1

Preliminari

1.1 Generalità sul linguaggio C++

In generale, un programma è costituito da:

- un insieme di algoritmi e
- un insieme di dati su cui operano gli algoritmi.

Quando l'accento è posto sugli algoritmi si parla di *programmazione procedurale*. Quando l'accento è posto sui dati, o meglio sui tipi di dato, si parla di *programmazione ad oggetti*. Il C++ è un linguaggio di programmazione *orientato agli oggetti* (si usa spesso l'acronimo “linguaggio OO”) e a *tipizzazione statica* che fornisce strumenti per supportare entrambi gli stili di programmazione ma non solo. Lo scienziato svedese Bjarne Stroustrup è il padre del C++, il cui primo progetto risale alla metà degli anni 1980. Attualmente, il C++ è uno dei linguaggi di programmazione più diffusamente utilizzati in qualsiasi ambito applicativo, in particolare quando l'efficienza dell'applicazione è un fattore critico. Il C++ ha fortemente influenzato lo sviluppo di altri diffusi linguaggi di programmazione orientati agli oggetti, quali Java e C#.

Lo strumento C++ per la programmazione procedurale è la definizione di *funzioni* che permettono al programmatore di estendere il linguaggio con nuove “istruzioni”. Lo strumento C++ per la programmazione ad oggetti è la definizione di *classi* che permettono al programmatore di estendere il linguaggio con nuovi tipi di dato. Una funzione descrive un algoritmo ma può contenere anche la definizione di alcuni tipi di dato usati dall'algoritmo. Una classe descrive un tipo di dato ma può contenere, anzi quasi sempre contiene, anche la definizione di funzioni che operano su tali dati.

Esamineremo come sono realizzate in C++ le seguenti caratteristiche fondamentali di ogni linguaggio OO legate al concetto di classe:

1. Encapsulation (incapsulamento) e information hiding (occultamento dell'informazione);
2. Polymorphism (polimorfismo);
3. Inheritance (ereditarietà).

Il C++ fornisce inoltre strumenti che permettono di integrare ed estendere i paradigmi di programmazione procedurale e ad oggetti. I più importanti tra questi sono:

- I template di funzione;
- I template di classe;
- La gestione delle eccezioni.

Va infine rimarcato che il C++ è un linguaggio di programmazione con tipizzazione statica (strongly typed in inglese), cioè il tipo di ogni variabile (numerico intero o in virgola mobile, carattere, booleano, classe, etc.), è sempre esplicitamente determinato nel codice sorgente mediante apposite parole chiave che identificano i tipi. Esistono vari linguaggi che supportano forme di tipizzazione dinamica (ad esempio Python e Ruby) dove le variabili possono riferirsi a valori di qualsiasi tipo, che quindi possono cambiare dinamicamente durante l'esecuzione del programma.

1.2 Richiami di C++

In questa sezione richiameremo sinteticamente alcuni utili argomenti di programmazione in C++ usualmente inclusi nel programma di un corso di Programmazione che utilizzi il linguaggio C++.

1.2.1 Namespace

Uno dei problemi nella programmazione a moduli è il cosiddetto inquinamento dello spazio dei nomi. Come ci intendiamo che la visibilità incondizionata dei nomi (di variabili, funzioni, tipi, etc.) può provare dei conflitti fra identificatori nella programmazione modulare, come dimostra il seguente esempio.

1.2 Richiami di C++

```
// file "Complex.h"
struct Complex {
    ... // implementazione1
    ... // di Complex
};

double module(Complex);
```



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

// Dichiarazione ILLEGALE
// errore in compilazione
struct Complex {
    ... // implementazione2
    ... // di Complex
};

void f() {
// si vorrebbero usare
// entrambi i tipi Complex
}
```

Il meccanismo dei namespace permette di incapsulare dei nomi che altrimenti inquinerebbero il namespace globale. In generale, si usano i namespace quando ci si aspetta che il codice sia usato in ambienti software esterni.

```
// file "Definizioni.h"

namespace newSpace {
    struct Complex {
        ...
    };

    double module(Complex);

    ... // altri membri del namespace newSpace
}
```

Il namespace newSpace identifica uno spazio dei nomi separato dal namespace globale. All'interno di un namespace si possono mettere dichiarazioni e definizioni qualsiasi. Il namespace non cambia il significato delle dichiarazioni che contiene, ne modifica soltanto la visibilità.

```
// file "Lib_UNO.h"

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



```
// file "Lib_DUE.h"

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

1 PRELIMINARI

Le dichiarazioni di un namespace non sono immediatamente visibili al programma, ma sono invece accessibili tramite l'operatore di scoping: `namespace::nome_dichiarazione`.

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

void funzione() {
    SPAZIO_UNO::Complex var1; SPAZIO_UNO::f(var1);
    SPAZIO_DUE::Complex var2; SPAZIO_DUE::g(var2);
    SPAZIO_DUE::g(var1); // errore in compilazione!
}
```

Un alias di namespace permette di associare a un namespace esistente un nome alternativo, tipicamente più breve o generico, per contrastare i nomi eccessivamente ingombranti di qualche namespace.

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

namespace UNO = SPAZIO_UNO;
namespace DUE = SPAZIO_DUE;

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

Spesso la proverbiale pigrizia dei programmatore porta a preferire un accesso non qualificato (cioè senza usare l'operatore di scoping) ai nomi di un namespace. Quando l'identificatore di un namespace è particolarmente lungo, in effetti l'accesso qualificato potrebbe risultare eccessivamente pedante da utilizzare. La direttiva d'uso rende le dichiarazioni di un namespace visibili in modo tale che si possa fare riferimento a loro senza qualificazione.

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

// direttiva d'uso
// rende visibili tutti i nomi del namespace SPAZIO_UNO
using namespace SPAZIO_UNO;

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

1.2 Richiami di C++

Il namespace a cui ci si riferisce tramite la keyword `using` deve essere stato già dichiarato (nell'esempio precedente c'è il corrispondente `include`).

La dichiarazione d'uso fornisce invece un meccanismo più selettivo di visibilità dei nomi, permettendo di rendere visibile in un namespace una singola dichiarazione, come illustrato nel seguente esempio.

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

// dichiarazione d'uso: rende visibili
// il nome Complex del namespace SPAZIO_UNO
// il nome g del namespace SPAZIO_DUE
using SPAZIO_UNO::Complex;
using SPAZIO_DUE::g;

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

Per impedire che inquinino il namespace globale, le componenti di tutte le librerie del C++ standard, in particolare la Standard Template Library STL e la libreria di input/output, sono dichiarate in un namespace chiamato `std`. Infatti, nel C++ standard, il seguente codice non compila¹.

```
#include <iostream>

int main() {
    cout << "Ciao!" << endl;
}
// errore di compilazione per g++:
// 'cout' was not declared in this scope
// 'endl' was not declared in this scope
```

Mediante una direttiva d'uso del namespace `std` il codice diventa corretto.

```
#include <iostream>

using namespace std;

int main() {
```

¹Per il compilatore g++ ciò è vero a partire dalla versione 3.x che aderisce al C++ standard per quanto concerne i namespaces.

```
cout << "Ciao!" << endl;  
}  
// Compila correttamente
```

Tutte le dichiarazioni nel file header `<iostream>` appartengono al namespace `std`, e quindi per renderle visibili senza usare esplicitamente l'operatore di scoping va usata la direttiva d'uso del namespace `std`. Alternativamente, dobbiamo usare un accesso qualificato dall'operatore di scoping.

```
#include <iostream>  
  
int main() {  
    std::cout << "Ciao!" << std::endl;  
}  
// Compila correttamente
```

Una direttiva d'uso è considerata generalmente una scelta discutibile per rendere visibili i nomi dichiarati nel namespace `std`. Infatti, questo fa riemergere il problema dell'inquinamento del namespace globale che il namespace `std` cerca invece di evitare. In un buono stile di programmazione è meglio usare delle dichiarazioni d'uso.

```
#include <iostream>  
  
using std::cout;  
using std::endl;  
  
int main() {  
    cout << "Ciao!" << endl;  
}
```

Per gli esempi di dimensione limitata che useremo nel corso, nella maggior parte dei casi ometteremo direttive e dichiarazioni di uso. Inoltre, anche le inclusioni dei file header di dichiarazione (come `#include<iostream>`) saranno spesso omesse. Inclusioni di file e direttive/dichiarazioni d'uso dovranno essere incluse per poter compilare correttamente gli esempi del testo (ad esempio con il compilatore `g++`).

1.2.2 Argomenti di default

Nel testo useremo la notazione `F()` per indicare che `F` è un identificatore di una qualche funzione o metodo. Un argomento di default è un valore dato nella dichiarazione di un parametro formale `x` di una funzione `F()` che il compilatore inserisce automaticamente quando non viene fornito alcun argomento attuale esplicito per il parametro `x` in una chiamata a `F()`. Gli argomenti di default sono piuttosto utili e convenienti poiché permettono di usare un unico nome di funzione in situazioni differenti. Ad esempio:

1.2 Richiami di C++

```
double potenza(double x, int n = 2) {  
...  
}
```

L'utilizzo degli argomenti di default nella definizione di una funzione deve rispettare la seguente regola: non è possibile avere un argomento di default seguito da un argomento non di default. Quindi, se in una chiamata ad una funzione F() con argomenti di default si utilizza un argomento di default allora tutti i successivi argomenti attuali nella lista degli argomenti di F() saranno di default. Vediamo alcuni esempi.

```
void F(double x, int n = 3, string s) { ... } // Illegale  
  
void G(double x, int n = 3, string s = "ciao") { ... } // OK  
  
G(3.2);           // OK: G(3.2,3,"ciao");  
G();             // Illegale  
G(3,3);           // OK: G(3,3,"ciao");  
G(3,3,"pippo"); // OK: G(3,3,"pippo");
```

1.2.3 Puntatori a funzione

Ricordiamo che è possibile definire dei *puntatori a funzione*. Un puntatore a funzione contiene l'indirizzo in memoria di una data (segnatura di) funzione (tipo di ritorno incluso), ovvero l'indirizzo iniziale in memoria del codice di una funzione. I puntatori a funzione possono essere dereferenziati, passati a funzioni, restituiti da funzioni, memorizzati in array e assegnati ad altri puntatori di funzione. La sintassi da usare è richiamata nell'esempio seguente.

```
#include<math.h>  
#include<iostream>  
using namespace std;  
  
// math.h contiene le dichiarazioni delle funzioni  
// double sin(double)  
// double cos(double)  
  
double F(double d) {return d+3.14;}  
  
int main() {  
    // pf è un puntatore ad una funzione con lista dei  
    // parametri (double) e tipo di ritorno double  
    double (*pf)(double); // parentesi (*pf) obbligatorie!  
    pf = &sin; cout << (*pf)(0) << endl; // stampa: 0  
    pf = &cos; cout << (*pf)(0) << endl; // stampa: 1
```

```
pf = &F; cout << (*pf)(0) << endl; // stampa: 3.14
}
```

1.2.4 La keyword `typedef`

La keyword `typedef` permette di dichiarare un alias per un tipo già esistente. Non si introduce quindi un nuovo tipo ma semplicemente si introduce un nuovo identificatore per riferirsi ad un tipo preesistente. Tipicamente viene introdotto un alias di tipo mediante `typedef` al fine di ottenere un identificatore compatto e dal nome significativo per un qualche tipo complesso, ad esempio che coinvolge array o puntatori. Vediamo alcuni esempi.

```
typedef unsigned short int Intero;

typedef const int* PuntCostante;

typedef long double ArrayReale[20];

typedef struct {
    double parte_reale;
    double parte_immaginaria;
} Complesso;

typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
```

Notiamo che nel caso dell'alias `Complesso`, la definizione del tipo struttura che precede l'introduzione dell'identificatore `Complesso` dichiara una cosiddetta struttura anonima e quindi l'uso di `typedef` definisce un alias per quel tipo struttura. Analogamente per l'alias `Giorno` del tipo enumerativo. Si ricorda che è possibile dichiarare tipi anonimi solo per i costrutti di tipo `struct`, `enum` ed `union`.

1.2.5 L'operatore virgola

L'operatore virgola (o sequenza) separa delle espressioni, le valuta procedendo da sinistra verso destra e ritorna il valore solamente dell'ultima espressione. Si noti quindi che la valutazione delle espressioni che precedono l'ultima può essere utile solo per gli eventuali effetti collaterali di queste valutazioni. L'operatore virgola è raramente usato in pratica. Vediamo un esempio.

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

1.2 Richiami di C++

```
cout << "c = " << c << endl; // stampa: 3  
}
```

1.2.6 Il tipo **string**

Per dare una prima idea delle classi in C++ rivediamo alcuni concetti basilari del tipo **string** che è una classe definita nella libreria standard STL.

Nello stile C le stringhe vengono trattate come degli array di caratteri, ovvero puntatori a char, delimitati sempre dal carattere finale nullo '/0' (il carattere con codice ASCII 0).

```
char* st = "ecco una stringa/0";  
char* st = "ecco una stringa"; // equivalente!
```

Questo stile alla C rende le stringhe non facilmente gestibili e possibile fonte di errori di programmazione.

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

string è un tipo del C++ definito come una classe nella libreria STL. Più precisamente, **string** è una istanziazione di un template di classe. Per usare il tipo **string** occorre quindi includere il file header di classe associato (e includere la direttiva d'uso di std). Possiamo quindi dichiarare variabili di tipo **string**.

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

Per una stringa **st**, tramite l'invocazione di funzione **st.size()** si ottiene la lunghezza di **st** come valore restituito dall'invocazione.

1 PRELIMINARI

```
cout << "La lunghezza di '" << st << "' è di " << st.size()
<< " caratteri;"
<< " l'eventuale carattere nullo terminante non è contato";
```

`size()` non è una funzione ordinaria (come le funzioni studiate nel corso di Programmazione), ma è invece una funzione tipica della programmazione ad oggetti. Notiamo la particolare sintassi per invocare questa funzione “sull’argomento attuale” `st`: `st.size()`. Dipende dal fatto che `string` è una *classe*, `st` è un *oggetto* della classe `string` e `size()` è un *metodo* della classe `string`. Altri modi per dichiarare una stringa sono:

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

Tramite l’invocazione di funzione `st.empty()`, che ritorna un valore booleano, si controlla se `st` è la stringa vuota. Su `string` sono definiti anche gli operatori di uguaglianza e disuguaglianza `==` e `!=` e gli operatori d’ordine `<`, `<=`, `>`, `>=` che implementano l’ordine lessicografico tra stringhe. Sono definiti inoltre l’operatore di assegnazione `=` e l’operatore di concatenazione tra stringhe `+`, ad esempio:

```
st1 = st + ", " + st2;
```

Sulle stringhe sono inoltre definite molte altre funzionalità che vedremo quando si presenterà la necessità di usarle. Consideriamo ora il seguente esempio.

```
string cifre = "0123456789";
char c;
int pos;
...
pos = cifre.find(c);
if (pos == string::npos)... // non è una cifra
else ... // pos è la posizione della prima
          // occorrenza di c nella stringa cifre
```

Dalla documentazione di STL ricaviamo che la funzione `find()` ritorna un valore di tipo `size_type`. `size_type` è un tipo integrale senza segno definito in `string`. L’invocazione `cifre.find(c)` cerca nella stringa `cifre` la prima occorrenza del carattere `c`. Se la trova ne ritorna la posizione (partendo da 0) altrimenti ritorna la costante predefinita `string::npos` (“non_posizione”, che “vale -1 ”, o, più precisamente, `size_type(-1)`).

Il seguente esempio illustra alcune ulteriori funzionalità del tipo `string`.

```
string st, st1;
int pos;
```

1.2 Richiami di C++

```
getline(cin,st); // legge una riga da cin e la memorizza in st
pos = st.find("rosso"); // cerca sottostringa "rosso"
if (pos == string::npos)
    // in questo caso "rosso" non è sottostringa di st
else
    // altrimenti pos è la posizione del carattere 'p' della
    // prima occorrenza di "rosso" come sottostringa in st.

// Nel seguito consideriamo l'esempio
// st == "FLAG: rosso bianco verde". Quindi, pos == 6.

st1 = st.substr(pos,5);
// ritorna la sottostringa di st di lunghezza 5 a partire dalla
// posizione pos (Nota Bene: pos parte da 0)
// Quindi, nell'esempio st1 == "rosso".

st.replace(pos,5,"blu");
// sostituisce in st i 5 caratteri a partire
// dalla posizione pos con la stringa "blu".
// Quindi, nell'esempio st diventa "FLAG: blu bianco verde"
```

1 PRELIMINARI

Capitolo 2

Classi e Oggetti

2.1 Tipi di dato astratti

Un tipo di dato astratto (acronimo ADT dall'inglese Abstract Data Type) è costituito da un insieme di valori e da un insieme di operazioni che permettono di manipolare tali valori, dette operazioni proprie del tipo o metodi pubblici del tipo. La specifica di un ADT deve distinguere l'interfaccia pubblica dell'ADT, ovvero la semantica del comportamento delle operazioni proprie, dalla sua rappresentazione interna, ovvero il modo in cui i valori sono rappresentati e l'implementazione delle operazioni proprie basata su tale rappresentazione interna dei valori. La rappresentazione interna di un ADT non deve essere accessibile all'utente dell'ADT, pertanto un cambio di rappresentazione interna dovrà essere del tutto trasparente all'utente. Si potranno quindi definire oggetti di un ADT che possono essere manipolati solamente mediante i metodi pubblici forniti dal tipo.

Su un ADT possono essere definite dall'utente delle operazioni aggiuntive, mediante la definizione di funzioni, che però non possono far uso della rappresentazione interna del tipo. Se la rappresentazione interna dovesse essere modificata allora sarà necessario modificare conseguentemente solo l'implementazione delle operazioni proprie mentre non subiranno alcuna modifica le parti di programma che usano oggetti di quel tipo, operazioni aggiuntive incluse.

I tipi primitivi del C++ sono esempi di ADT. Ad esempio, il tipo `int` fornisce delle operazioni (somma, moltiplicazione, uguaglianza, etc.) che sono realizzate sfruttando la rappresentazione interna dei valori interi (ad esempio, rappresentazione su 4 byte in complemento a due). Il tipo `string` della libreria STL è un ulteriore esempio di ADT. Invece, un tipo `struct` non rispetta il concetto di ADT, in quanto la sua rappresentazione interna è visibile all'esterno, ovvero una qualunque funzione esterna può direttamente usare i vari campi dati che costituiscono la definizione di una `struct`.

Il concetto object oriented di classe permette di implementare nei linguaggi di programmazione OO gli ADT. In C++, non sarebbe possibile realizzare compiutamente l'idea

2 CLASSI E OGGETTI

di ADT senza usare gli strumenti del linguaggio legati al concetto di classe. Vediamo un esempio di realizzazione in C++ di un ADT che rappresenta i numeri complessi usando la struct e le funzioni.

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

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

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

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

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

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

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

```
#include "complessi.h"
#include <iostream>
using std::cout;

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

    // possiamo pero' usare la rappresentazione interna dell'ADT!
    comp x2 = {0.3,3.1}, y2 = {3,6.3};
```

```

comp z2;
z2.re=x2.re+y2.re; z2.im=x2.im+y2.im;

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

```

2.2 Le classi

La possibilità di definire delle classi permette al programmatore di estendere il linguaggio con nuovi tipi di dato astratti. Introdurremo le caratteristiche delle classi in C++ mediante un esempio: la classe `orario` i cui elementi saranno oggetti che rappresentano un orario della giornata come “13:02:45”. Partiremo introducendo la classe `orario` nella sua forma più semplice per poi introdurre passo passo tutte le caratteristiche delle classi.

Una classe si specifica in due “parti” o “fasi” tipicamente separate: la definizione dell’interfaccia della classe, detta anche dichiarazione della classe, e la definizione (o implementazione) dei suoi metodi. La definizione dell’interfaccia consiste nella dichiarazione dei campi dati (o attributi) della classe e dei metodi della classe.

La dichiarazione della classe `orario` nella sua forma più semplice è la seguente:

```

class orario {
public:           // metodi della classe
    int Ore();    // selettore delle ore
    int Minuti(); // selettore dei minuti
    int Secondi(); // selettore dei secondi

private:          // unico campo dati della classe
    int sec;      // scegiamo di rappresentare un orario mediante
                  // il numero di secondi trascorsi dalla mezzanotte
};

```

L’implementazione (o definizione) dei metodi (o funzioni proprie) è la seguente:

```

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

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

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

```

La sintassi “`::`” identifica l’operatore di risoluzione di visibilità (operatore di scoping in inglese). La sintassi `orario::Ore()` denota quindi la funzione `Ore()` che è stata dichiarata nel contesto della classe `orario`.

2 CLASSI E OGGETTI

Non è strettamente necessario che le due parti di dichiarazione e definizione di una classe siano separate. Infatti avremmo anche potuto scrivere:

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

In questo caso diciamo che i metodi sono definiti *inline*. Per ragioni che vedremo in seguito è opportuno tenere separate la dichiarazione e la definizione di una classe ed anzi normalmente esse vengono memorizzate su due file distinti.

Possiamo quindi dichiarare variabili di tipo `orario` in ogni punto del programma in cui la dichiarazione della classe `orario` sia visibile. Tali variabili vengono dette *oggetti della classe* `orario`.

```
int main() {  
    orario mezzanotte;  
    cout << mezzanotte.Secondi() << endl;  
}
```

La chiamata di funzione `mezzanotte.Secondi()` assume il seguente significato: "esegui il metodo `Secondi()` sull'oggetto `mezzanotte` di tipo `orario`".

2.2.1 Il puntatore `this`

È importante conoscere come vengano implementati gli oggetti di una classe e come siano eseguiti i metodi della classe su tali oggetti.

Quando viene dichiarato un oggetto come `mezzanotte` di tipo `orario` viene riservata una zona di memoria per il valore del campo dati intero `sec`. Quindi ogni oggetto di tipo `orario` ha un proprio campo dati `sec`.

Vi è invece in memoria un'unica copia del codice (oggetto) dei metodi `Secondi()`, `Minuti()` e `Ore()` della classe `orario`. Ed è tale codice che viene eseguito quando vengono effettuate le chiamate

```
mezzanotte.Secondi();  
mezzanotte.Minuti();  
mezzanotte.Ore();
```

2.2 Le classi

che possiamo interpretare come: "esegui i metodi `Secondi()`, `Minuti()` e `Ore()` sull'oggetto `mezzanotte` di tipo `orario`". In una invocazione come `mezzanotte.Secondi()` diremo che `mezzanotte` è l'*oggetto di invocazione* del metodo `Secondi()`.

Le implementazioni dei tre metodi `Secondi()`, `Minuti()` e `Ore()` usano il campo dati `sec`. Ogni oggetto ha un proprio campo dati `sec`. Dunque, se il metodo `Secondi()` viene invocato dalla chiamata `mezzanotte.Secondi()` esso deve usare il campo `sec` dell'oggetto `mezzanotte` mentre se viene invocato con `mezzogiorno.Secondi()` esso deve usare il campo `sec` dell'oggetto `mezzogiorno`.

Come avviene questo? I metodi di una classe possiedono un parametro implicito `this` (`this` è l'identificatore di tale parametro ed è una parola chiave del C++) di tipo puntatore ad oggetti della classe stessa. Ad esempio, per i metodi della classe `orario` il parametro implicito `this` è di tipo `orario*`. Quando un metodo viene invocato su qualche oggetto di invocazione `obj`, al parametro implicito `this` viene automaticamente assegnato l'indirizzo di `obj`, cioè `this` punterà ad `obj`. Possiamo rendere chiaro questo fatto immaginando di esplicitare il parametro implicito `this` nella dichiarazione del metodo `Secondi()` e nella sua chiamata:

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

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

Durante l'esecuzione del metodo `Secondi()` invocato dalla chiamata `mezzanotte.Secondi()` il puntatore `this` punta quindi all'oggetto di invocazione `mezzanotte` mentre durante l'esecuzione del metodo `Secondi()` conseguente alla chiamata `mezzogiorno.Secondi()` `this` punta all'oggetto di invocazione `mezzogiorno`.

`this` è una keyword del C++. All'interno di un metodo ci si può riferire all'oggetto di invocazione tramite la dereferenziazione `*this`. Ad esempio, avremmo potuto scrivere:

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

anche se, come abbiamo visto, nella definizione di `Secondi()` l'oggetto di invocazione si può sottintendere e normalmente si segue questa prassi.

A volte l'utilizzo esplicito del puntatore `this` diviene necessario nella definizione di qualche metodo. L'esempio più semplice è quello in cui un metodo deve restituire l'oggetto stesso di invocazione:

```

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

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

```

2.3 Parte privata e pubblica

Nella precedente dichiarazione della classe `orario` appaiono le due keyword `private` e `public`. Si tratta dei cosiddetti *specificatori* (o *modificatori*) d'accesso. Indicano che il campo dati `sec` appartiene alla *parte privata* della classe `orario` mentre i tre metodi appartengono alla *parte pubblica*. Dall'esterno di una classe si può accedere solamente alla sua parte pubblica, mentre i membri della parte privata sono inaccessibili. Ad esempio:

```

orario o;
cout << o.Ore() << endl; // OK: Ore() è pubblico
cout << o.sec << endl;   // Errore: sec è privato

```

Invece, come abbiamo visto nella definizione di `Ore()`, `Minuti()` e `Secondi()`, i metodi della classe ovviamente possono accedere alla parte privata: non solo a quella dell'oggetto di invocazione ma anche alla parte privata di qualsiasi altro oggetto della classe (ad esempio i parametri del metodo).

Tipicamente la parte privata contiene le dichiarazioni dei campi dati ed eventuali metodi di utilità che il progettista della classe usa per l'implementazione della classe ma che non fanno parte della sua interfaccia pubblica. La parte pubblica contiene invece le dichiarazioni dei metodi che definiscono l'interfaccia pubblica dell'ADT e che quindi il progettista vuole rendere disponibili agli utenti della classe.

I membri di una classe che seguono lo specificatore d'accesso `private` appartengono alla parte privata, mentre quelli che seguono lo specificatore `public` appartengono alla parte pubblica. Un membro che non segue nessuno specificatore di accesso appartiene per default alla parte privata.

Vediamo l'esempio della classe `complesso` che rappresenta i numeri complessi mediante la rappresentazione cartesiana e della classe `complesso2` che usa invece la rappresentazione polare.

```

class complesso {
    private:

```

2.4 Costruttori

```
double re, im;
public:
    void iniz_compl(double, double);
    double reale();
    double immag();
};

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

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

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

```
#include<math.h> // libreria di funzioni matematiche

// rappresentazione polare: modulo mod e argomento arg
class complesso2 {
private:
    double mod, arg;
public:
    void iniz_compl(double, double);
    double reale();
    double immag();
};

void complesso2::iniz_compl(double r, double i) {
    mod = sqrt(r*r + i*i); arg = atan(i/r);
}

double complesso2::reale() { return mod*cos(arg); }

double complesso2::immag() { return mod*sin(arg); }
```

2.4 Costruttori

Vediamo come assegnare dei valori ad un oggetto. Come fare perché all'oggetto `mezzanotte` corrisponda effettivamente l'orario, ovvero valore, "00:00:00"? Visto che la rappresentazione è in "secondi trascorsi dalla mezzanotte" dobbiamo assegnare al campo `datisec` di `mezzanotte` l'intero 0. Tuttavia,

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

non funziona perché il campo dati `sec` è un membro privato.

2 CLASSI E OGGETTI

Si devono invece usare i cosiddetti *costruttori*: sono dei metodi con lo stesso nome della classe e senza tipo di ritorno che vengono invocati automaticamente quando viene dichiarato (e quindi costruito) un oggetto. I costruttori saranno normalmente dichiarati nella parte pubblica di una classe al fine di renderli accessibili all'utente. È possibile dichiarare un costruttore privato, ma in tal caso esso sarà inutilizzabile all'esterno della classe (generalmente si tratta di un errore logico di programmazione). Vediamo il seguente esempio.

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

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

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

Si possono definire più costruttori purché differiscano nella lista dei parametri, ovvero nel numero e/o tipo dei parametri. È come sovraccaricare (cioè fare overloading del) l'identificatore del metodo che ha lo stesso nome della classe. Il costruttore senza parametri, se definito, viene detto *costruttore di default*.

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

La definizione dei costruttori di *orario* è quindi la seguente.

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

// costruttore ore-minuti
```

2.4 Costruttori

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

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

Vediamo qualche esempio d'uso dei costruttori: gli oggetti vengono "creati" tramite una invocazione di un costruttore.

```
// uso il costruttore ore-minuti-secondi
orario adesso_preciso(14,25,47);

// uso il costruttore ore-minuti
orario adesso(14,25);

// uso il costruttore senza parametri
orario mezzanotte;

// uso il costruttore senza parametri
orario mezzanotte2;

// uso il costruttore ore-minuti
orario troppo(27,25);

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

Nella seguente assegnazione:

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

il costruttore a tre parametri crea un cosiddetto *oggetto anonimo*, cioè un oggetto a cui non è associato alcun identificatore, della classe `orario`. Il tempo di vita dell'oggetto anonimo termina non appena esso sia stato usato per assegnarlo all'oggetto `o`.

2 CLASSI E OGGETTI

Quindi, un'espressione come `orario(12, 33, 25)` può comparire in ogni punto in cui è richiesto un valore di tipo `orario`. Possiamo considerare tali espressioni come i "valori" del tipo `orario`.

Viene invocato un costruttore anche quando si crea dinamicamente sullo heap un oggetto tramite l'operatore `new`.

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

cout << ptr->Ore() << endl; // stampa: 0
cout << ptr1->Ore() << endl; // stampa: 14
// si ricordi che ptr->Ore() è una abbreviazione di (*ptr).Ore()
```

Facciamo il punto sulla dichiarazione della classe `orario`.

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

Se in una classe non viene dichiarato esplicitamente alcun costruttore, è automaticamente disponibile il cosiddetto *costruttore standard*. Il costruttore standard non ha parametri ed è quindi un costruttore di default. Il suo comportamento è il seguente. Esso lascia indefiniti i valori dei campi dati dei tipi primitivi (`int`, `float`, `bool`, etc.) e derivati (puntatori, riferimenti e array), mentre per i campi dati di tipo classe (possibilmente anche di qualche libreria) richiama il corrispondente costruttore di default, che potrà essere dichiarato esplicitamente oppure potrà essere quello standard.

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

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

2.4 Costruttori

Attenzione (errore comune!): quando viene dichiarato almeno un costruttore (costruttore di copia, che vedremo più avanti, incluso) il costruttore standard non è più disponibile!

```
class orario {  
public:  
    orario(int,int); // solamente costruttore ore-minuti  
    ...  
};  
  
orario o; // Errore, non compila: manca un costruttore di default
```

Vediamo altri modi di dare valore ad oggetti di tipo orario.

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

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 qualsiasi classe C ha segnatura C (const C&). L'operatore di assegnazione permette di assegnare il valore di ogni campo dati dell'oggetto adesso al corrispondente campo dati dell'oggetto copia.

Vedremo in seguito come e perché assegnazione e costruttore di copia si possano ridefinire.

2.4.1 Costruttori come convertitori di tipo

I costruttori con un solo parametro come

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

oltre a poter essere usati normalmente come gli altri costruttori funzionano anche da *convertitori di tipo*, cioè possono essere usati per conversioni implicite (cast impliciti). Sia C(T) un costruttore della classe C ad un solo parametro di tipo T. Allora, il costruttore C(T) viene automaticamente invocato ogni volta che compare un valore di tipo T dove invece ci si aspetterebbe un valore di tipo C. Consideriamo ad esempio le seguenti istruzioni:

```
orario x;
x = 8;
```

Nota importante: La conversione implicita avviene effettivamente a run-time mediante una invocazione del costruttore ad un argomento che costruisce un oggetto temporaneo.

Nell'assegnazione `x = 8`; si tenta di assegnare il valore intero 8 all'oggetto `x` di tipo `orario`. Ciò provoca quindi la conversione implicita dell'intero 8 in un oggetto di tipo `orario` mediante l'invocazione del costruttore ad un argomento intero `orario(int)`. Gli effetti dell'assegnazione `x = 8`; sono quindi i seguenti:

1. viene invocato il costruttore `orario(int)` con parametro attuale 8 che crea un oggetto temporaneo anonimo della classe `orario`;
2. l'oggetto temporaneo viene assegnato all'oggetto `x`;
3. infine, viene "distrutto", ossia deallocated, l'oggetto temporaneo.

Ad esempio:

```
orario x,y;
x = 8;           // OK: equivale a x = orario(8);
y = 8+12;        // OK: equivale a y = orario(8+12);
```

Tutto ciò vale anche se si definiscono costruttori con argomenti di default. Il seguente costruttore con tre argomenti di default:

```
class orario {
public:
    orario(int o = 0, int m = 0, int s = 0);
    ...
};

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

funziona come costruttore a 0, 1, 2 e 3 parametri e pertanto anche come convertitore di tipo da `int` ad `orario`.

Quindi, definendo un costruttore `C(T)` con un solo parametro di tipo `T` in una classe `C` si introduce anche una conversione implicita dal tipo `T` del parametro del costruttore al tipo `C` della classe. Questo è vero anche se definiamo un costruttore con più di un parametro dei

2.4 Costruttori

quali il primo è di tipo T (che può anche avere un valore di default) e per tutti i successivi parametri è previsto un valore di default.

Facciamo il punto sulla classe `orario` e vediamo un esempio di codice (che possiamo inserire in qualche `main()`) che usa tale classe.

```
// file "orario.cpp"
class orario {
public:
    // costruttore a tre parametri con argomenti di default
    orario(int =0,int =0,int =0);
    int Ore();
    int Minuti();
    int Secondi();
private:
    int sec;
};

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

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

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

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

La keyword `explicit`

In qualche caso possiamo non volere che un costruttore ad un parametro sia richiamato implicitamente come convertitore di tipo. Basta allora premettere la parola chiave `explicit` a questo costruttore ad un parametro:

```
class orario {  
public:  
    explicit orario(int); // costruttore esplicito ad un parametro  
...  
};  
  
orario x = 8; // Errore, non compila: non vi è una invocazione  
// implicita del costruttore orario(8)
```

2.4.2 Operatori esplicativi di conversione

È anche possibile definire in una classe C una conversione implicita dal tipo C ad un altro tipo T tramite degli *operatori esplicativi di conversione* nel seguente modo.

```
class orario {  
public:  
    operator int() { return sec; } // conversione orario -> int  
...  
};  
  
orario o(14,37);  
int x = o; // OK: viene richiamato implicitamente l'operatore  
// int() sull'oggetto o
```

Questo vale per qualsiasi tipo T, in particolare anche per un tipo classe.

2.5 Metodi costanti

Naturalmente, l'invocazione di un metodo su un oggetto di invocazione può modificare lo stato di quell'oggetto, ossia i suoi campi dati. Si dice anche che il metodo provoca *effetti collaterali* (in inglese *side effects*) sull'oggetto d'invocazione. Continuiamo con la nostra classe esempio `orario`. Supponiamo di aggiungere alla parte pubblica di `orario` i seguenti due metodi.

```
class orario {  
public:  
    ...
```

2.5 Metodi costanti

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

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

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

int main() {
    orario mezzanotte;
    cout << mezzanotte.Ore();           // stampa: 0
    orario adesso(15);
    cout << adesso.Ore();             // stampa: 15
    adesso = mezzanotte.UnOraPiuTardi();
    cout << adesso.Ore();             // stampa: 1
    cout << mezzanotte.Ore();         // stampa: 0
    mezzanotte.AvanzaUnOra();
    cout << mezzanotte.Ore();         // stampa: 1
}
```

Per evitare modifiche non volute dell'oggetto di invocazione da parte di un metodo si può dichiarare quel metodo *costante*. In altri termini, la marcatura `const` di qualche metodo `m()` dichiara esplicitamente che ogni invocazione di `obj.m()` non provocherà side effects su `obj`. Ad esempio:

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

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

Il compilatore controlla che nella definizione del metodo dichiarato costante non compaia alcuna istruzione che possa alterare il valore dell'oggetto di invocazione come, ad esempio, assegnazioni ai campi dati dell'oggetto di invocazione o invocazioni di metodi

non costanti sull'oggetto di invocazione o sui suoi campi dati. Se le trova, viene segnalato un errore in compilazione. Si assume quindi che ogni metodo non costante possa provocare effetti collaterali sull'oggetto di invocazione, sebbene un metodo dichiarato non costante non necessariamente provoca delle modifiche allo stato dell'oggetto di invocazione. Ad esempio, il metodo `UnOraPiùTardi()` della classe `orario` può (anzi dovrebbe) essere dichiarato costante in quanto non provoca effetti collaterali all'oggetto di invocazione, a differenza del metodo `AvanzaUnOra()`, per cui la dichiarazione come metodo costante provocherebbe un errore in compilazione.

Naturalmente, anche un oggetto può essere dichiarato costante: ciò significa che tale oggetto non può venire modificato dopo che è stato costruito.

```
const orario LE_DUE(14);
const orario LE_TRE(15);
LE_DUE = LE_TRE;      // Assegnazione illegale, non compila
```

Inoltre, tutti i campi dati di un oggetto costante diventano costanti, come mostra il seguente esempio.

```
class orario {
    ...
    void m() { // qualche metodo di orario
        ...
        const orario LE_DUE(14);
        // LE_DUE.sec = 50400; // assegnazione illegale, non compila
        ...
    }
}
```

Per i metodi costanti vale la seguente regola: l'oggetto di invocazione di un metodo costante diventa costante. Piú precisamente ciò significa che nel corpo di un metodo costante di qualche classe C, il puntatore `this` all'oggetto di invocazione ha tipo `const C*` invece che `C*` e conseguentemente l'oggetto di invocazione `*this` ha tipo `const C`. Ciò spiega perché nel corpo di un metodo costante non è possibile fare delle assegnazioni ai campi dati dell'oggetto di invocazione: infatti, poiché l'oggetto di invocazione è costante anche i campi dati dell'oggetto di invocazione sono costanti e pertanto non possono essere modificati tramite assegnazioni.

Regola importante: un oggetto costante si può usare come oggetto di invocazione soltanto per metodi dichiarati costanti. Se si tenta di invocare un metodo non costante su un oggetto costante il compilatore segnala un errore.

```
const orario LE_TRE(15);
LE_TRE.StampaSecondi(); // OK, stampa: 54000
orario o;
```

2.5 Metodi costanti

```
o = LE_TRE.UnOraPiuTardi(); // Errore, non compila perché  
// UnOraPiuTardi() non è stato dichiarato metodo costante
```

Vale una ovvia eccezione alle precedente regola: i costruttori sono dei metodi non dichiarati costanti che possono venire invocati su oggetti dichiarati costanti. Se così non fosse, non sarebbe possibile costruire degli oggetti costanti!

La precedente regola spiega perché nel corpo di un metodo costante non è possibile invocare metodi non costanti sull'oggetto di invocazione: infatti, l'oggetto di invocazione è costante e quindi posso invocare su di esso solamente metodi costanti.

Supponiamo che qualche metodo `m()` non provochi effetti collaterali sull'oggetto di invocazione. In questo caso, è prassi comune e buona regola di programmazione da seguire dichiarare quel metodo `m()` come costante. In particolare, questo permette di invocare quel metodo `m()` anche su oggetti costanti: se `m()` non fosse dichiarato costante ciò non sarebbe possibile e questo sarebbe in contrasto con il comportamento effettivo di `m()` che non modifica l'oggetto di invocazione.

Esercizio 2.5.1. Quali delle istruzioni numerate nel seguente `main()` sono legali (cioè compilano) e quali sono illegali?

```
class C {  
private:  
    int x;  
public:  
    C(int n = 0) {x=n;}  
    C F(C obj) {C r; r.x = obj.x + x; return r;}  
    C G(C obj) const {C r; r.x = obj.x + x; return r;}  
    C H(C& obj) {obj.x += x; return obj;}  
    C I(const C& obj) {C r; r.x = obj.x + x; return r;}  
    C J(const C& obj) const {C r; r.x = obj.x + x; return r;}  
};  
  
int main() {  
    C x, y(1), z(2); const C v(2);  
    z=x.F(y);           // (1)  
    v.F(y);            // (2)  
    v.G(y);            // (3)  
    (v.G(y)).F(x);    // (4)  
    (v.G(y)).G(x);    // (5)  
    x.H(v);            // (6)  
    x.H(z.G(y));      // (7)  
    x.I(z.G(y));      // (8)  
    x.J(z.G(y));      // (9)  
    v.J(z.G(y));      // (10)  
}
```

2.6 Campi dati e metodi statici

I metodi della classe `orario` che abbiamo considerato finora sono stati pensati per operare su di uno specifico oggetto, l'oggetto di invocazione. Quando scriviamo

```
orario adesso(14,33);
cout << adesso.Minuti();
```

intendiamo stampare i minuti dello specifico oggetto `adesso`. Potremmo avere l'esigenza che la classe `orario` ci fornisca un valore `OraDiPranzo` di tipo `orario` che sia sempre lo stesso, una specie di costante della classe `orario`. Per ottenere ciò potremmo definire un metodo costante `OraDiPranzo()` che ritorna un oggetto della classe `orario`.

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

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

Notiamo però che in questo modo per stampare l'ora di pranzo abbiamo necessariamente bisogno di un qualche oggetto di tipo `orario` da usare come oggetto di invocazione, anche se tale oggetto non verrà comunque usato dal metodo `OraDiPranzo()`. Sarebbe quindi un metodo logicamente mal definito.

```
const orario inutile;
cout << "Si pranza alle " << inutile.OraDiPranzo().Ore() << " e "
    << inutile.OraDiPranzo().Minuti() << " minuti";
```

La dichiarazione `static` permette di associare sia metodi che campi dati all'intera classe invece che a singoli oggetti della classe.

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

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

Dichiareremo quindi un metodo `static` quando l'azione del metodo è indipendente dall'oggetto di invocazione, cioè l'oggetto di invocazione non è necessario per la definizione del metodo. Ciò permette di invocare un metodo statico senza un oggetto di invocazione.

2.6 Campi dati e metodi statici

I metodi dichiarati `static` non hanno pertanto il parametro implicito `this`. All'esterno della classe si invocano premettendo al loro identificatore l'operatore di scoping “`::`” per quella classe.

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

Nelle implementazioni degli altri metodi della classe i metodi dichiarati `static` si possono comunque invocare senza l'operatore di scoping `classe::` (analogamente ai metodi non statici che si possono invocare nella definizione degli altri metodi della classe senza premettere l'oggetto di invocazione).

Anche un campo dati di una classe può essere dichiarato `static`. Quando viene costruito un oggetto di una classe non viene allocata memoria per i suoi campi dati statici. La memoria per i campi dati statici è unica per tutti gli oggetti ed è allocata una volta per tutte all'inizio. Anche ad un campo dati statico di una classe `C` si accede dall'esterno di `C`, sempre che il campo sia pubblico (poiché un campo dati ovviamente può essere marcato `public` oppure `private`), usando l'operatore di scoping `C::`, mentre internamente a `C` l'operatore di scoping non è necessario.

I campi dichiarati `static` non si possono inizializzare all'interno della definizione della classe. La loro inizializzazione deve essere necessariamente esterna alla classe ed è sempre richiesta qualora il campo dati statico venga effettivamente utilizzato (altrimenti il linker segnala un errore). Ad esempio, se avessimo aggiunto alla classe `orario` il campo dati statico

```
static int Sec_di_una_Ora;
```

avremmo dovuto inizializzarlo all'esterno della definizione della classe `orario` con:

```
int orario::Sec_di_una_Ora = 3600;
```

Un campo dati statico permette quindi di memorizzare informazioni globali, cioè appartenenti alla classe nel suo complesso e non alle sue singole istanze. È quindi una specie di variabile globale della classe. Rispetto a una ordinaria variabile globale, un campo dati statico ha i vantaggi di permettere il controllo dell'accesso, pubblico o privato, e di essere comunque a tutti gli effetti un membro della classe. Un uso tipico dei campi dati statici consiste nella definizione di costanti di classe, cioè di campi dati pubblici, statici e costanti, che vedremo nel seguito.

Il prossimo esempio mostra come un campo dati statico possa essere utilizzato per contare il numero di istanze di una classe costruite da un programma.

Esempio 2.6.1.

```
class C {
    int dato;           // privato
public:
```

```

C(int);           // costruttore ad un argomento
static int cont; // campo dati statico pubblico
};

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

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

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

```

2.7 Overloading di operatori

Vogliamo ora aggiungere alla classe orario una operazione che permetta di sommare una durata temporale ad un orario. Ad esempio, comando 2:15:00 alle 20:30:00 si devono ottenere le 22:45:00 e comando 3:45:00 alle 22:30:00 si devono ottenere le 2:15:00. Aggiungiamo quindi un metodo che permetta di sommare in questo modo due oggetti di tipo orario: il metodo somma (orario) che ritorna un oggetto di orario.

```

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

```

Possiamo usare questo metodo nel modo seguente.

```

orario ora(22,45);
orario DUE_ORE_E_UN_QUARTO(2,15);
ora = ora.Somma(DUE_ORE_E_UN_QUARTO);

```

Un modo più elegante per ottenere questa funzionalità di somma consiste nel definire un overloading, cioè una ridefinizione, dell'operatore di somma “+”.

```

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

```

2.7 Overloading di operatori

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

Possiamo quindi usare l'operatore + per oggetti della classe orario nel modo seguente:

```
orario ora(22,45);
orario DUE_ORE_E_UN_QUARTO(2,15);
ora = ora + DUE_ORE_E_UN_QUARTO;
```

In generale, la ridefinizione di un operatore OP si ottiene definendo una funzione il cui identificatore è `operatorOP`. Ogni occorrenza dell'operatore OP equivale quindi ad una invocazione di tale funzione. Un operatore può essere ridefinito mediante un metodo proprio oppure mediante una funzione esterna alla classe. Poiché un metodo ha già un argomento implicito dato dall'oggetto di invocazione, la ridefinizione, ad esempio, di un operatore binario corrisponde ad un metodo che ha un solo argomento esplicito oppure ad una funzione esterna con due argomenti. Analogamente, un operatore unario corrisponde ad un metodo proprio senza argomenti esplicativi.

Il C++ permette di sovraccaricare circa quaranta operatori i più interessanti tra i quali sono i seguenti:

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

Possiamo quindi sovraccaricare sia operatori binari come “+”, “<” e “->”, sia operatori unari come “++” e “--”. Non si possono sovraccaricare l'operatore di selezione di membro “.”, l'operatore condizionale ternario “? :”, l'operatore di scoping “::”, l'operatore “sizeof” e l'operatore di identificazione dinamica di tipo “typeid”.

Non si possono cambiare le proprietà sintattiche dell'operatore, ovvero:

1. posizione (prefissa, infissa o postfissa),
2. numero di operandi,
3. precedenza,
4. associatività.

Fra gli argomenti dell'operatore ridefinito deve comparire almeno un argomento di un tipo definito dall'utente, cosicché gli operatori predefiniti, come ad esempio la somma di interi, non possono essere modificati. Gli operatori “=” (assegnazione), “[]” (operatore di indicizzazione), “()” (operatore di chiamata di funzione) e “->” (operatore di selezione di

membro tramite puntatore) possono essere ridefiniti solamente come metodi propri: questo assicura che il primo operando sia un l-valore (e quindi modificabile).

Se non avessimo ridefinito l'operatore + per la classe `orario` il compilatore non avrebbe accettato l'espressione:

```
ora = ora + DUE_ORE_E_UN_QUARTO;
```

Questo è vero per tutti gli operatori ad eccezione di “=” (assegnazione), “&” (operatore di indirizzo) e “,” (operatore virgola). Il C++ fornisce una definizione standard per questi tre operatori per ogni classe. Naturalmente il programmatore è libero di ridefinirli e in questo caso la definizione data dal programmatore prevale su quella standard. Il programmatore può anche rendere inaccessibili tali definizioni standard per questi tre operatori inserendo le loro dichiarazioni nella parte privata della classe. Nel seguito vedremo vari esempi di ridefinizione di operatori.

Esercizio 2.7.1. Scrivere una definizione degli operatori -, ==, > e < per la classe `orario`. Scrivere una definizione degli operatori +, -, ==, > e < per la classe `complesso`.

2.7.1 Assegnazione standard

Se `a` e `b` sono due oggetti della stessa classe `C` e non è stato ridefinito l'operatore di assegnazione per `C` allora l'istruzione `a = b;` ha come effetto quello di assegnare ad ognuno dei campi dati di `a` il valore del corrispondente campo dati di `b` (si tratta di una assegnazione membro a membro dei campi dati). Per ogni campo dati di qualche tipo `T` si usa l'assegnazione, standard o ridefinita, per quel tipo `T`. Questo è quindi il comportamento dell'*assegnazione standard* che viene automaticamente invocata nel caso in cui il programmatore non ridefinisca esplicitamente l'operatore di assegnazione `=`. La segnatura dell'assegnazione standard per una generica classe `C` è la seguente:

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

Analizzeremo più approfonditamente nel seguito il comportamento dell'operatore di assegnazione.

2.7.2 Costruttore di copia standard

Una funzione che ha un comportamento simile all'assegnazione standard è il *costruttore di copia*. Il costruttore di copia per una generica classe `C` è un costruttore con la seguente segnatura:

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

Il costruttore di copia viene invocato automaticamente nei seguenti tre casi.

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

2.7 Overloading di operatori

```
orario adesso(14,30);
orario copia = adesso; // dichiara ed inizializza copia
orario copial(adesso); // dichiara ed inizializza copial
```

2. Quando un oggetto viene passato per valore come parametro attuale in una chiamata di funzione, come in:

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

dove Somma è la funzione:

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

3. Quando una funzione ritorna per valore tramite l'istruzione `return` un oggetto, come in `return aux;` nella precedente funzione `Somma()`.

In tutti e tre i casi viene costruito un nuovo oggetto di tipo `orario` i cui campi dati vengono inizializzati con i valori dei corrispondenti campi dati di `adesso` (caso 1), di `DUE_ORE_E_UN_QUARTO` (caso 2) e di `aux` (caso 3). Nel primo caso l'oggetto costruito assume il nome `copia` (`copial`), nel secondo caso assume il nome `t` del parametro formale, mentre nel terzo caso viene costruito un oggetto temporaneo anonimo che verrà distrutto (cioè deallocated) non appena sia stato usato come valore di ritorno della funzione.

Attenzione al caso 3. Il compilatore `g++` (dalla versione 3.x) compie di default la seguente ottimizzazione riportata dal manuale: “*The C++ standard allows an implementation to omit creating a temporary which is only used to initialize another object of the same type. Specifying this option disables that optimization, and forces g++ to call the copy constructor in all cases*”. L'ottimizzazione è prevista dallo standard ed è piuttosto sottile: dice che ogni qualvolta si dovrebbe creare un oggetto temporaneo “inutile” usato solamente per inizializzare un nuovo oggetto dello stesso tipo, tale oggetto temporaneo non viene creato. La specifica di tale ottimizzazione non è sempre completamente chiara e quindi, per i nostri scopi didattici, nel seguito ignoreremo tale ottimizzazione. L'opzione `-fno-elide-constructors` all'invocazione del compilatore `g++` rimuove questa ottimizzazione dal compilatore ristabilendo il comportamento non ottimizzato previsto dal C++ standard. Nel seguito la compilazione `g++` sarà sempre intesa con questa opzione.

Esempio 2.7.2.

```
// compilazione: g++ -fno-elide-constructors
C fun(C a) { return a; }
```

```

int main() {
    C c;
    fun(c);
    // 2 invocazioni del costruttore di copia
    C y = fun(c);
    // 3 invocazioni del costruttore di copia (e non 2!)
    C z; z = fun(c);
    // 2 invocazioni del costruttore di copia
    fun(fun(c));
    // 4 invocazioni del costruttore di copia (e non 3!)
}

```

Questo è il comportamento del *costruttore di copia standard* che viene usato automaticamente nel caso in cui il programmatore non ne definisca uno esplicitamente. La differenza con l'assegnazione standard è che il costruttore di copia crea un nuovo oggetto inizializzandolo da un altro oggetto dello stesso tipo mentre l'assegnazione cambia il valore ad un oggetto preesistente. Discuteremo in seguito le motivazioni e i modi per ridefinire assegnazione e costruttore di copia.

Esercizio 2.7.3. Il seguente programma compila correttamente. Quali stampe provoca la sua esecuzione?

```

class Puntatore {
public:
    int* punt;
};

int main() {
    Puntatore x, y;
    x.punt = new int(8);
    y = x;
    cout << "*(" << y.punt << " = " << *y.punt << endl;
    *y.punt = 3;
    cout << "*(" << x.punt << " = " << *(x.punt) << endl;
}

```

2.7.3 Overloading di operatori con funzioni esterne

Vediamo come sia possibile sovraccaricare degli operatori non come metodi propri della classe come abbiamo già visto ma come funzioni esterne alla classe. Mostreremo con un esempio le motivazioni e l'utilità di ciò.

Vogliamo definire una funzione che stampi sullo stream di standard output `cout` un oggetto di tipo orario. Ad esempio, l'oggetto

2.7 Overloading di operatori

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

dovrebbe essere stampato come "15:0:0". Naturalmente, l'operatore che permette di stampare un valore di un tipo predefinito sullo stream cout è operator<< e questo operatore è tra quelli ridefinibili. La scelta che pare naturale consiste nell'aggiungere il seguente metodo alla parte pubblica della classe orario:

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

Osserviamo che per poter ripetere le invocazioni dell'operatore << in una unica istruzione occorre che esso ritorni lo stream per riferimento. L'utilizzo dell'operatore sovraccaricato così definito presenta tuttavia un problema. Per invocarlo correttamente dobbiamo 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 primo e seconda occorrenza di "<<" come le seguenti chiamate al metodo operator<< della classe orario:

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

in cui il primo argomento deve essere l'oggetto di invocazione. Vorremmo invece poter invocare ripetutamente l'operatore di output in una unica istruzione al solito semplice modo:

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

Si può ottenere una soluzione adeguata tramite l'overloading di operator<< come funzione esterna alla classe. (Si veda la sezione 3.4 per la giustificazione del tipo const orario& per il parametro formale o.)

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

Ora possiamo scrivere:

```
orario le_tre(15,0), dodici(12,0);
cout << "Adesso sono le " << le_tre << endl;
// stampa: Adesso sono le 15:0:0
cout << "Fra dodici ore saranno le " << le_tre + dodici << endl;
// stampa: Fra dodici ore saranno le 3:0:0
```

2 CLASSI E OGGETTI

Confrontiamo le due diverse definizioni.

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

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

Nella seconda definizione non appare l'oggetto di invocazione di tipo `orario`, mentre con la prima definizione esso veniva assunto come primo operando. Esso è sostituito nella seconda definizione da un secondo parametro formale che viene assunto come secondo operando. Tuttavia la nuova definizione, non essendo un metodo della classe `orario` non ha accesso alla parte privata della classe. Tale accesso non è però necessario in quanto i metodi usati `Ore()`, `Minuti()` e `Secondi()` sono pubblici.

Vediamo un altro esempio. La combinazione delle due dichiarazioni

```
orario orario::operator+(orario); // operatore di somma
orario::orario(int); // costruttore ad un solo parametro
```

permette di scrivere

```
orario y(12,20), x;
x = y + 4; // OK
```

Infatti, il costruttore con un parametro viene usato implicitamente per convertire l'intero 4 in un valore di tipo `orario`. Invece non funziona

```
x = 4 + y; // Errore, non compila: operatore non definito
```

in quanto il metodo `operator+` può essere invocato solamente se l'operando sinistro, cioè l'oggetto di invocazione, è un oggetto esplicito della classe `orario`.

Se dichiariamo `operator+` come funzione esterna possiamo scrivere sia

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

che

```
x = 4 + y; // OK: conversione del primo parametro
```

Naturalmente in entrambi i casi funziona

```
x = 4 + 5; // OK: somma tra interi e poi conversione del risultato
```

Attenzione: non possiamo però definire la funzione esterna come

2.7 Overloading di operatori

```
orario operator+(orario x, orario y) {
    orario aux;
    aux.sec = (x.sec + y.sec) % 86400;
    return aux;
}
```

perché il campo dati `sec` è privato! Vedremo più avanti un metodo generale per ovviare a tale problema nel caso in cui ciò risulti necessario. Per ora possiamo risolvere il problema attraverso la seguente implementazione di “basso livello” in cui si usano soltanto i metodi `Ore()`, `Minuti()` e `Secondi()` ed il costruttore `orario(int, int, int)` che sono tutti pubblici.

```
orario operator+(orario x, orario y) {
    int sec = x.Secondi() + y.Secondi();
    int min = x.Minuti() + y.Minuti() + sec/60;
    sec = sec % 60;
    int ore = x.Ore() + y.Ore() + min/60;
    min = min % 60;
    ore = ore % 24;
    orario aux(ore,min,sec);
    return aux;
}
```

Esercizio 2.7.4. Il seguente programma compila correttamente. Quali stampe provoca la sua esecuzione?

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

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

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

Esercizio 2.7.5. Definire una classe `Point` i cui oggetti rappresentano un punto nello spazio tridimensionale reale. Includere un costruttore di default, un costruttore a tre argomenti

che inizializza un punto, un metodo `negate()` che trasforma un punto nel suo opposto, un metodo `norm()` che restituisce la distanza di un punto dall'origine e un metodo `print()` che stampa le coordinate di un punto. Separare interfaccia ed implementazione della classe.

Esercizio 2.7.6. Definire una classe `Persona` i cui oggetti rappresentano anagraficamente un personaggio storico caratterizzato da nome, anno di nascita e anno di morte. Includere opportuni costruttori, metodi di accesso ai campi e l'overloading dell'operatore di output come funzione esterna. Separare interfaccia ed implementazione della classe. Si definisca inoltre un esempio di metodo `main()` che usa tutti i metodi della classe e l'operatore di output.

2.8 Incapsulamento e modularizzazione

Nella classe `orario` l'ora del giorno viene rappresentata internamente mediante un campo dati privato `sec` di tipo `int`. In alternativa avremmo anche potuto usare tre campi dati di tipo `int` per memorizzare separatamente le ore, i minuti e i secondi. Le due scelte presentano vantaggi e svantaggi. Ad esempio, la seconda scelta semplifica la realizzazione dei metodi `Ore()`, `Minuti()` e `Secondi()` ma complica invece l'implementazione di `operator<`.

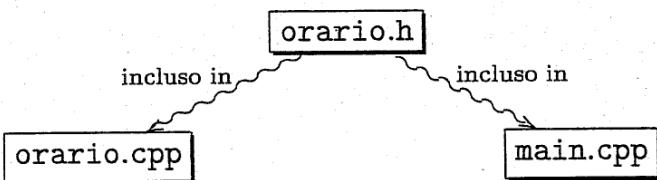
In ogni caso, qualunque sia la scelta per la rappresentazione interna, essa non ha alcuna conseguenza per un utente della classe `orario` in quanto egli non può accedere alla parte privata della classe. Si tratta quindi di una scelta implementativa del progettista della classe. Anzi, se all'utilizzatore della classe viene fornita soltanto la definizione dell'interfaccia pubblica della classe — che è l'unica parte che gli serve per poter usare la classe (infatti rappresenta una sorta di “documentazione” della classe) — allora esso non avrebbe neppure modo di sapere come sono rappresentati i dati o come sono implementati i metodi. Vedremo in seguito un modo per nascondere all'utente esterno, oltre all'implementazione dei metodi, anche la dichiarazione della parte privata della classe, che invece ora rimane visibile nella dichiarazione della classe. Questo segue il principio generale dell'information hiding. Lo sviluppatore della classe può cambiare l'implementazione liberamente (ad esempio per correggere bug o migliorare l'efficienza) senza che venga richiesta alcuna modifica al codice sviluppato dagli utenti di tale classe.

Le classi permettono la *modularizzazione* dei programmi. Abbiamo già accennato al fatto che la dichiarazione di una classe e le definizioni dei suoi metodi vengono messe usualmente in due file separati. La dichiarazione di una classe viene di solito messa in un file header che convenzionalmente ha estensione “`.h`”. Ad esempio, possiamo memorizzare la dichiarazione della classe `orario` nel file `orario.h`.

Le definizioni dei metodi `Ore()`, `Minuti()` e `Secondi()` vengono invece memorizzate in un file che convenzionalmente ha estensione “`.cpp`” (o “`.C`”) al fine di permettere la compilazione separata. Ad esempio, possiamo mettere le definizioni di tali metodi nel file `orario.cpp`, compilare separatamente tale file producendo il file `orario.o` contenente il codice oggetto e quindi fornire agli utilizzatori della classe soltanto il file `orario.o`. Il file

2.8 Incapsulamento e modularizzazione

header `orario.h` risulta necessario sia nella compilazione dei metodi in `orario.cpp` sia nella compilazione del codice che utilizza la classe `orario`, che supponiamo sia contenuto in un file `main.cpp`. La situazione è rappresentata nella figura seguente.



Il file header `orario.h` dovrà quindi essere incluso in entrambi i file `orario.cpp` e `main.cpp`, al pari di eventuali file header di libreria, quali, ad esempio, `iostream` o `string`. Otteniamo ciò mediante la direttiva di inclusione `#include`.

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

L'uso dei delimitatori " " invece di < > implica che il file `orario.h` va cercato nella directory corrente mentre `iostream` va cercato in un insieme standard di directory per il compilatore utilizzato.

2.8.1 Il preprocessore

Il *preprocessore* è quella parte del compilatore (o programma separato invocato dal compilatore) che esegue una elaborazione preliminare del testo sorgente del programma, prima che avvenga l'analisi lessicale e sintattica e quindi la compilazione in codice oggetto. Esso può inserire il contenuto di altri file, espandere dei simboli definiti nel programma, e includere o escludere parti di codice dal testo che effettivamente sarà compilato. Queste operazioni sono controllate dalle *direttive per il preprocessore*, nelle quali il primo carattere diverso da una spaziatura è il carattere "#". Le direttive standard del preprocessore C++ sono le seguenti:

```
#include #error #if #else #elif #endif
#define #ifdef #ifndef #undef #line #pragma
```

La direttiva `#include` indica che un determinato file dovrà essere incluso nell'unità di compilazione.

Una *macro* è un simbolo, eventualmente seguito da una lista di argomenti formali delimitati tra parentesi tonde, che prima della compilazione viene sostituito da una sequenza di elementi lessicali (identificatori, espressioni, etc.) corrispondenti alla definizione della macro. Se la macro ha degli argomenti formali al momento della sostituzione questi argomenti vengono rimpiazzati dagli argomenti attuali corrispondenti. La definizione di una macro è preceduta dalla direttiva `#define`. Ad esempio, il seguente codice sorgente che specifica le macro `COST` e `max`

2 CLASSI E OGGETTI

```
#define COST 123
#define max(A,B) ((A) > (B) ? (A) : (B))

int main() {
    x = COST;
    y = max(4,z+2);
}
```

viene espanso dal preprocessore come

```
int main() {
    x = 123;
    y = ((4) > (z+2) ? (4) : (z+2));
}
```

Se in una macro manca la definizione, come ad esempio in

```
#define ORARIO_H
```

ogni occorrenza del nome della macro viene sostituita con un carattere spazio. La definizione di una macro M ha effetto sino alla fine dell'unità di compilazione (ovvero sino a fine file), o sino ad una corrispondente direttiva di delimitazione #undef M.

Le macro non sostituiscono le funzioni. Il C++ non richiede un uso esteso delle macro, prevedendo i meccanismi dell'espansione inline delle funzioni (ad opera del compilatore e non del preprocessore) e lo strumento dei template di funzione e di classe.

Compilando contemporaneamente più file può succedere che un file header venga incluso più volte in un file provocando un errore di compilazione, in quanto il compilatore incontra delle definizioni multiple di uno stesso identificatore, ad esempio di una classe. Per evitare ciò si inseriscono nei file header delle ulteriori direttive per il preprocessore dette di *compilazione condizionale* (o di *isolamento*). Consideriamo il seguente esempio.

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

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

class D {
```

2.8 Incapsulamento e modularizzazione

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

```
// file "main.cpp"  
#include "C.h"  
#include "D.h"  
  
int main() {  
    C c(3); D d(4);  
    d.print(c);  
}  
// main.cpp non compila.  
// Il compilatore segnala l'errore: "redefinition of class C"
```

Le direttive di compilazione condizionale permettono di includere o escludere dalla compilazione dei segmenti di codice.

```
#ifdef name  
    text  
#endif  
  
#ifndef name  
    text  
#endif
```

Tramite la direttiva `#ifdef name`, se l'identificatore `name` è definito allora il successivo frammento di programma che termina con la direttiva `#endif` — nell'esempio denotato da `text` — viene compilato altrimenti, ovvero nel caso in cui `name` non sia definito, `text` viene escluso dalla compilazione. D'altra parte `#ifndef name` agisce dualmente a `#ifdef name`.

Ad esempio, la prima inclusione del seguente file header "orario.h"

```
// file "orario.h"  
#ifndef ORARIO_H  
#define ORARIO_H  
class orario {  
    ...  
};  
#endif
```

provoca la definizione, mediante la direttiva `#define ORARIO_H`, della macro globale `ORARIO_H`. Le inclusioni successive alla prima non vengono quindi effettuate a causa della presenza del test `#ifndef ORARIO_H`. Vediamo un altro esempio.

```
#define LINUX
#ifndef LINUX
// istruzioni per la versione Linux
#elif defined WINDOWS
// istruzioni per la versione Windows
#endif
```

In questo esempio per ottenere il “porting” del programma da Linux a Windows basta cambiare la direttiva `#define LINUX` in `#define WINDOWS`.

2.8.2 Compilazione e linking

I due file `orario.cpp` e `main.cpp` possono essere compilati separatamente. In particolare, compilando il file `orario.cpp` si ottiene un file oggetto compilato `orario.o`. Il seguente comando di invocazione del compilatore `g++`:

```
g++ -c orario.cpp
```

compila il codice sorgente nel file `orario.cpp` in codice oggetto memorizzandolo automaticamente nel file `orario.o`. Il meccanismo della compilazione separata permette di non fornire agli utilizzatori della classe `orario` il file sorgente `orario.cpp`, evitando in tal modo di rendere disponibile all’utilizzatore le definizioni dei metodi della classe. All’utilizzatore sarà sufficiente avere a disposizione (1) il file header `orario.h` da includere in ogni modulo utente `M` della classe `orario` e (2) il file oggetto `orario.o` che verrà usato dal *linker* per generare il codice eseguibile dal codice oggetto derivante dalla compilazione del modulo `M`.

Il *linker* è il programma parte del (o invocato dal) compilatore che produce il codice eseguibile a partire dal codice oggetto. Il compito principale del linker è quello di risolvere i riferimenti a qualche identificatore *I* non definito in un file di codice oggetto individuando in quale altro file di codice oggetto l’identificatore *I* è invece definito. Il linker rimpiazzera quindi il riferimento simbolico ad *I* con l’indirizzo in memoria della definizione di *I*.

Ad esempio, per un modulo utente `main.cpp` (che ad esempio contiene il `main()`), dovremo compilare e quindi linkare tramite i seguenti comandi di invocazione di `g++`:

```
g++ -c main.cpp      // compila main.cpp in main.o
g++ main.o orario.o // linka main.o e orario.o producendo l'eseguibile
```

Questo meccanismo di compilazione separata del codice, oltre ad essere un semplice esempio di modularizzazione, è conforme al principio generale dell’information hiding nella programmazione orientata agli oggetti.

2.8.3 Il comando make

Sarà spesso capitato di decomprimere dei sorgenti di qualche programma disponibile per Linux trovando un file di nome `Makefile` (o `makefile`) e delle istruzioni di installazione che richiedono l'esecuzione di comandi come `make` o `make install`. Si tratta di invocazioni del ben noto comando di utilità "make". Quando si scrivono programmi composti di diversi file sorgente, ci si trova di fronte al problema della ripetuta ricompilazione dei sorgenti e della successiva fase di link dei file oggetto per generare l'eseguibile. La cosa più conveniente da fare è individuare i file che sono stati modificati dall'ultima compilazione e ricompilare soltanto questi ultimi. Su Linux e nei sistemi Unix-like, si usa generalmente il famoso comando `make`, ideato originalmente nel 1977 da Stuart Feldman (e per cui ha ricevuto l'ACM Software System Award nel 2003). L'utility `make` serve a determinare automaticamente quali file che compongono un programma hanno bisogno di essere ricompilati, e può generare i comandi necessari a farlo, seguendo alcune regole definite nel file `Makefile`.

`make` è molto spesso utilizzato per programmi scritti in C/C++, ma può essere usato con qualsiasi altro linguaggio di programmazione. Infatti, `make` è in grado di invocare qualunque comando di sistema che possa essere eseguito da shell. Quindi, ad esempio, può ricompilare anche file `TEX`, ricostruire database, o comunque eseguire una qualunque serie di operazioni che necessiti di essere automatizzata. Per utilizzare `make`, è necessario creare un file di nome `Makefile` che descriva le relazioni esistenti tra i vari file e i comandi da eseguire per aggiornare ognuno di questi. Tipicamente, in un programma modularizzato in numerosi file, l'eseguibile viene aggiornato dal linker usando i file oggetto, mentre i file oggetto vengono generati dal compilatore usando i file sorgente. Una volta definito un `Makefile` appropriato, basterà invocare da shell il comando

`make`

affinché automaticamente vengano ricompilati i file sorgente che hanno subito qualche modifica dall'ultima compilazione del programma che ha generato l'eseguibile e quindi venga rigenerato l'eseguibile aggiornato.

Illustriamo la struttura di un tipico `Makefile` tramite un esempio. Supponiamo di avere un programma C++ costituito da cinque file sorgente con estensione `.cpp` e tre file header con estensione `.h` utilizzati dai sorgenti `.cpp`. Il programma `make`, quando andrà a ricompilare il programma, ricompilerà prima tutti i file sorgente che sono stati modificati dall'ultima compilazione. Se è cambiato anche qualche file header `.h` allora verrà ricompilato ogni file sorgente `.cpp` che includeva quel file header modificato. Infine, se almeno un file è stato modificato (header o sorgente) e quindi almeno un file oggetto è stato rigenerato allora tutti i file oggetto, nuovi e vecchi, verranno linkati assieme per generare l'eseguibile aggiornato.

Un `Makefile` consiste di alcune regole descritte in generale con la seguente sintassi:

2 CLASSI E OGGETTI

TARGET : DEPENDENCIES ...

COMMAND

Usualmente TARGET è il nome del file eseguibile o del file oggetto da ricompilare, ma può essere anche il nome di una particolare azione da compiere (ad esempio clean spesso identifica l'azione di rimozione di file non necessari). Quindi TARGET è un identificatore dell'azione da compiere: all'invocazione da shell del comando

make clean

verrà eseguito il target clean del Makefile. Le DEPENDENCIES sono usate come input per generare l'azione TARGET e di solito sono più di una. Più genericamente vengono inclusi come DEPENDENCIES tutti i file o le azioni, cioè i TARGET, da cui dipende il completamento dell'azione TARGET. COMMAND è invece il comando da eseguire; può essere più di uno e di solito si applica sulle DEPENDENCIES.

Come guida concreta, un tipico Makefile per il precedente programma esempio composto da cinque file .cpp e tre file header .h potrebbe avere la seguente struttura.

```
# Commento: la variabile CC è il comando che invoca il compilatore
CC=g++

# CCFLAGS: flags per il compilatore
CCFLAGS=-Wall -march=x86-64

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

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

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

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

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

print : print.cpp      # commento: deve essere preceduto da un TAB
       g++ -c print.cpp -o print.o

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

Notiamo l'uso della doppia barra “\\” per dividere un unico comando in più righe per facilitarne la lettura. A questo punto una semplice invocazione da shell del comando `make` eseguirà la prima azione prevista dal `Makefile`, che nel nostro caso è l'azione “`prog_mio`”, la quale, a sua volta, chiama altre azioni, cioè i file oggetto da compilare. È anche possibile specificare esplicitamente una particolare azione che `make` deve eseguire mediante l'invocazione `make azione`. Nel nostro caso, una invocazione `make clean` esegue i comandi per rimuovere i file oggetto. L'utilità del comando `make` dovrebbe quindi essere evidente per programmi composti di molti file e strutturati in modo complesso. Non è altrettanto facile rendersi conto delle potenzialità di questa utility: basterebbe spulciare nei `Makefile` che di solito accompagnano i programmi open source per vedere come attraverso `make` si possa rendere automatico non solo il lavoro di ricompilazione ma qualsiasi altra operazione complessa che richieda degli aggiornamenti. Nonostante il processo di sviluppo del software avvenga spesso all'interno di un cosiddetto IDE (Integrated Development Environment) — quali, ad esempio, Microsoft Visual Studio, Eclipse, Apple Xcode, NetBeans, etc. — `make` rimane una utilità diffusamente utilizzata specialmente in ambienti Unix-like.

2.8.4 Modularizzazione delle classi

Abbiamo visto come la definizione di classi permetta di aggiungere nuovi tipi di dato disponibili al programmatore. Naturalmente, le classi, una volta definite, possono essere usate per definire *modularmente* altre classi più complesse. Analizzeremo questo aspetto tramite l'esempio di una classe `telefonata` che utilizza la classe `orario`. Gli oggetti della classe `telefonata` devono appunto rappresentare una telefonata caratterizzata da: ora di inizio, ora di fine e numero chiamato. Consideriamo dapprima il file `telefonata.h` che contiene la dichiarazione della classe `telefonata` e quindi il file `telefonata.cpp` che contiene la definizione dei metodi.

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

using std::ostream;

class telefonata {
public:
    telefonata(orario, orario, int);
    telefonata();
    orario Inizio() const;
    orario Fine() const;
    int Numero() const;
```

2 CLASSI E OGGETTI

```
    bool operator==(const telefonata&) const;  
private:  
    orario inizio, fine;  
    int numero;  
};  
  
ostream& operator<<(ostream&, const telefonata&);  
#endif
```

Dobbiamo aggiornare il comportamento dei costruttori in presenza di campi dati il cui tipo è una classe. Consideriamo una classe C con campi dati x_1, \dots, x_k di qualsiasi tipo, possibilmente qualche altra classe. L'ordine dei campi dati è determinato dall'ordine in cui essi appaiono nella definizione della classe C. Supponiamo di definire nella classe C un qualsiasi costruttore generico:

```
C(Tipo1, ..., Tipon) { // codice }
```

Il comportamento di tale costruttore è il seguente:

- (1) Per ogni campo dati x_i di tipo T_i non classe (ovvero di tipo primitivo o derivato), viene allocato un corrispondente spazio in memoria per contenere un valore di tipo T_i ed il valore viene lasciato indefinito.
- (2) Ogni campo dati x_i di tipo classe T_i viene costruito mediante una invocazione del costruttore di default $T_i()$.
- (3) Infine, viene eseguito il codice del corpo del costruttore.

I precedenti punti (1) e (2) vengono eseguiti per tutti i campi dati x_1, \dots, x_k della classe C seguendo il loro ordine di dichiarazione nella classe C. Vediamo subito un esercizio per illustrare questo comportamento.

Esercizio 2.8.1. Il seguente programma compila correttamente. Quali stampe provoca la sua esecuzione?

```
class C {  
private:  
    int x;  
public:  
    C() {cout << "C0 "; x=0;}  
    C(int k) {cout << "C1 "; x=k;}  
};  
  
class D {  
private:  
    C c;  
public:
```

2.8 Incapsulamento e modularizzazione

```
D() {cout << "D0 "; c = C(3);}
};

class E {
private:
    char c;
    C c1;
public:
    D d;
    C c2;
};

int main() {
    D y; cout << endl;           // Stampa: C0 D0 C1
    E x; cout << endl;           // Stampa: C0 C0 D0 C1 C0
    E* p = &x; cout << endl; // Def. di puntatore => nessuna stampa
    D& a = y; cout << endl; // Def. di riferim. => nessuna stampa
}
```

Consideriamo quindi una classe C che include un campo dati x di tipo T dove T è qualche altra classe. Supponiamo inoltre di definire nella classe C un qualsiasi costruttore C(T₁, ..., T_n). Vedremo in seguito come sia possibile specificare un comportamento diverso da quello determinato dalle precedenti regole (1) e (2) tramite la cosiddetta *lista di inizializzazione* del costruttore. Vogliamo qui notare che in mancanza della lista di inizializzazione, viene richiesta la disponibilità del costruttore di default della classe T per costruire il campo dati x. Quindi se il costruttore di default (standard o definito esplicitamente) per T non dovesse essere disponibile, il compilatore segnalerà un errore.

Esempio 2.8.2. Il seguente programma compila?

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

class C {
private:
    int x;
    D d;
public:
    C() {x=5; d=D(4);}
};
```

2 CLASSI E OGGETTI

```
int main() {
    C z; // Errore in compilazione:
          // In method C::C() no matching function for call to D::D()
}
```

Ritorniamo quindi alla nostra classe telefonata e procediamo con la definizione dei metodi.

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

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

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

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

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

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

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

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

Esercizio 2.8.3. La classe telefonata presenta l'inconveniente di non poter rappresentare numeri telefonici che iniziano con lo 0. Scegliere una rappresentazione alternativa per il numero telefonico che risolva l'inconveniente e riscrivere le definizioni dei metodi per tale rappresentazione.

2.9 Campi dati costanti

Talvolta risulta utile poter dichiarare alcuni campi dati di una classe come costanti. Ad esempio, è ragionevole supporre che una volta costruito un oggetto della classe telefonata

2.9 Campi dati costanti

il suo campo dati numero non possa venire in seguito modificato. Abbiamo già visto come dichiarare in una classe dei metodi costanti e come dichiarare e inizializzare degli oggetti costanti in qualche definizione di funzione o metodo. Vedremo ora come si dichiarano dei *campi dati costanti*. Poiché non è possibile usare l'istruzione di assegnazione per inizializzare dei campi costanti vedremo un nuovo modo per inizializzare sia campi costanti che non costanti, che diventerà il modo standard per inizializzare i campi dati di una classe.

Un campo dati si dichiara costante premettendo al suo tipo il modificatore const.

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

Avendo dichiarato costante il campo dati intero numero la seguente definizione del costruttore senza argomenti

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

viene rifiutata dal compilatore, poiché non è permessa l'assegnazione ad un campo costante. Occorre usare una definizione diversa: dobbiamo richiamare esplicitamente il "costruttore di copia" per il campo dati costante intero numero invece di allocare la memoria per numero tramite il "costruttore standard" degli interi e successivamente assegnargli un valore tramite l'assegnazione. La sintassi da usare è la seguente:

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

È possibile richiamare esplicitamente il costruttore di copia per qualsiasi campo dati, anche per quelli non costanti. Quindi, la definizione del costruttore con parametri di telefonata diventa:

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

Per confrontare le due diverse definizioni analizziamo come vengono eseguite le seguenti istruzioni

```
orario o1(12,25,33), o2(12,47,12); // (1)  
telefonata telefonata_a_carlo(o1,o2,333333); // (2)
```

usando il costruttore vecchia maniera

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

2 CLASSI E OGGETTI

e usando il costruttore nel nuovo stile

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

ovviamente nell'ipotesi che non ci siano campi dati costanti e quindi siano legali entrambe le definizioni.

In entrambi i casi viene dapprima richiamato due volte il costruttore a tre argomenti della classe `orario` per costruire i due oggetti `o1` e `o2` dell'istruzione (1). Usando il costruttore vecchia maniera per eseguire l'istruzione (2) viene innanzitutto richiamato due volte il costruttore di copia di `orario` ed una volta il "costruttore di copia" di `int` per il passaggio dei parametri per valore. Ciò corrisponde quindi all'esecuzione delle seguenti tre istruzioni:

```
orario i(o1); orario f(o2); int n(333333);
```

Vengono quindi costruiti i tre campi `inizio`, `fine`, `numero` di `telefonata_a_carlo` usando i corrispondenti costruttori di default. Infine vengono eseguite le tre assegnazioni:

```
inizio = i; fine = f; numero = n;
```

Invece, usando il costruttore nel nuovo stile non vengono usati i costruttori di default per costruire i campi `inizio`, `fine`, `numero` per poi cambiarne i loro valori con le assegnazioni, ma vengono usati direttamente i costruttori di copia. Vengono comunque richiamati i costruttori di copia per il passaggio dei parametri per valore.

Per evitare di richiamare i costruttori di copia per il passaggio dei parametri per valore, possiamo dichiarare per riferimento tali parametri formali. È inoltre opportuno dichiararli come riferimenti costanti affinché sia degli oggetti costanti che degli oggetti anonimi (i "valori" della classe) possano essere passati come parametri attuali.

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

In questo modo vengono eseguiti soltanto i costruttori di copia una sola volta per ogni parametro. Ciò corrisponde quindi alle seguenti istruzioni:

```
orario inizio(o1); orario fine(o2); int numero(333333);
```

Se `C(...)` è un qualsiasi costruttore di una classe `C`, la lista di chiamate esplicite al costruttore di copia nella definizione di `C(...)` appartiene alla cosiddetta lista di inizializzazione del costruttore `C(...)`.

2.10 Liste di inizializzazione nei costruttori

Abbiamo visto come un costruttore possa richiamare i costruttori di copia per creare ed inizializzare alcuni campi dati, in particolare i campi dati dichiarati costanti. In generale, un costruttore può creare ed inizializzare i suoi campi dati tramite una esplicita *lista di inizializzazione*, che consiste in una lista di invocazioni a costruttori, possibilmente di copia.

In una classe C con lista ordinata di campi dati x_1, \dots, x_k , un costruttore con lista di inizializzazione per i campi dati x_{i_1}, \dots, x_{i_j} è definito tramite la seguente sintassi:

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

Il comportamento del costruttore è il seguente:

1. Ordinatamente per ogni campo dati x_i ($1 \leq i \leq k$) viene richiamato un costruttore:
 - (a) o esplicitamente tramite una chiamata ad un costruttore $x_i(\dots)$ definita nella lista di inizializzazione
 - (b) oppure implicitamente (cioè che non appare nella lista di inizializzazione) tramite una chiamata al costruttore di default $x_i()$.
2. Quindi viene eseguito il codice del costruttore.

Notiamo i seguenti punti:

- Naturalmente, parliamo di costruttori e costruttori di copia anche per campi dati di tipo non classe (tipi primitivi o tipi derivati): la lista di inizializzazione può includere inizializzazioni anche per questi campi dati.
- La chiamata implicita al costruttore di default per un campo dati di tipo non classe, come al solito, alloca lo spazio in memoria ma lascia indefinito il valore.
- L'ordine in cui vengono invocati i costruttori, esplicitamente o implicitamente, è sempre determinato dalla lista ordinata dei campi dati, qualsiasi sia l'ordine delle chiamate nella lista di inizializzazione.

Vediamo un esempio.

Esempio 2.10.1. Il seguente programma compila correttamente. Quali stampe provoca la sua esecuzione?

```
class D {
public:
    D() {cout << "D0 ";}
    D(int a) {cout << "D1 ";}
};
```

2 CLASSI E OGGETTI

```
class E {
private:
    D d;
public:
    E(): d(3) {cout << "E0 ";}
    E(double a, int b) {cout << "E2 ";}
    E(const E& a): d(a.d) {cout << "Ec ";}
};

class C {
private:
    int z;
    E e;
    D d;
    E* p;
public:
    C(): p(0), e(), z(4) {cout << "C0 ";}
    C(int a, E b): e(3.7,2), p(&b), z(1), d(3) {cout << "C1 ";}
    C(char a, int b): e(), d(2), p(&e) {cout << "C2 ";}
};

int main() {
    E e; cout << endl;           // stampa: D1 EO
    C c; cout << endl;           // stampa: D1 E0 D0 C0
    C c1(1,e); cout << endl;     // stampa: Ec D0 E2 D1 C1
    C c2('b',2); cout << endl;   // stampa: D1 E0 D1 C2
}
```

Campi dati di tipo riferimento. Anche se non è molto comune, è naturalmente possibile dichiarare un campo dati *c* di tipo riferimento *T&* (ovvero un alias), dove *T* è un qualsiasi altro tipo. Analogamente ai campi dati dichiarati costanti, questo campo dati *c* deve essere obbligatoriamente inizializzato tramite una chiamata al costruttore di copia inclusa nella lista di inizializzazione. Quindi la lista di inizializzazione includerà sempre le inizializzazioni per i campi dati costanti e di tipo riferimento.

Costruttori standard. Il comportamento dei costruttori standard di default e di copia è 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 oppure definito esplicitamente), dove per i tipi non classe (cioè primitivi e derivati) tale “costruttore di default standard” semplicemente alloca la memoria per il campo dati *x* (e naturalmente non la inizializza).

2.10 Liste di inizializzazione nei costruttori

- Una chiamata `C(obj)` al costruttore di copia standard di una classe `C` invoca ordinatamente per ogni campo dati `x` di `C` il corrispondente costruttore di copia (standard oppure definito esplicitamente) sul relativo campo dati `obj.x` dell'oggetto parametro attuale `obj`, dove per i tipi non classe tale “costruttore di copia standard” alloca la memoria per `x` e la inizializza.

Esercizio 2.10.2. Definire un costruttore di default legale per la classe `C`.

```
class E {  
private:  
    int x;  
public:  
    E(int z=0): x(z) {}  
};  
  
class C {  
private:  
    int z;  
    E x;  
    const E e;  
    E& r;  
    int* const p;  
public:  
    C();  
};
```

Esercizio 2.10.3. Il seguente programma compila correttamente (si intende la compilazione g++ con l'opzione -fno-elide-constructors). Quali stampe provoca la sua esecuzione?

```
class D {  
private:  
    int z;  
public:  
    D(int x=0): z(x) {cout << "D01 "}  
    D(const D& a): z(a.z) {cout << "Dc "}  
};  
  
class C {  
private:  
    D d;  
public:  
    C(): d(D(5)) {cout << "C0 "}  
    C(int a): d(5) {cout << "C1 "}  
    C(const C& c): d(c.d) {cout << "Cc "}
```

2 CLASSI E OGGETTI

```
};

int main() {
    C c1; cout << "UNO" << endl;
    C c2(3); cout << "DUE" << endl;
    C c3(c2); cout << "TRE" << endl;
}
```

Esercizio 2.10.4. Si osservino e confrontino le stampe provocate dall'esecuzione dei seguenti due programmi.

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

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

int main() {
    D x; cout << endl;
    // stampa: C0 D0
    D y(x); cout << endl;
    // stampa: Cc
}
```

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

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

int main() {
    D x; cout << endl;
    // stampa: C0 D0
    D y(x); cout << endl;
    // stampa: C0 Dc
}
```

Si noti attentamente che la differenza è spiegata dal fatto che il costruttore di copia standard di una classe D invoca ordinatamente per ogni campo dati x di D il corrispondente costruttore di copia del tipo di x.

Capitolo 3

Classi Collezione e Argomenti Correlati

Un oggetto di una *classe collezione* (o classe contenitore) rappresenta una collezione di elementi (omogenei o anche eterogenei) che può essere gestita tramite varie funzionalità, tra le quali l'inserimento e la rimozione. Tipicamente le classi collezione hanno fra i loro campi dati dei puntatori a strutture dati dinamiche ricorsive, quali liste o alberi. I contenitori della libreria STL, ad esempio `vector`, `list` e `map`, sono degli esempi di (template di) classi collezione.

Usiamo come esempio una classe che rappresenta un insieme di telefonate memorizzate in una lista. Chiameremo bolletta tale classe. Essa dovrà prevedere la possibilità di aggiungere e togliere una telefonata, di controllare se una bolletta è vuota, di stampare l'elenco di tutte le telefonate, di calcolare il costo totale della bolletta ed altro ancora.

```
// file "bolletta.h"
#ifndef BOLLETTA_H
#define BOLLETTA_H
#include "telefonata.h"
class bolletta {
public:
    bolletta() : first(0) {} // definizione inline
    bool Vuota() const;
    void Aggiungi_Telefonata(telefonata);
    void Togli_Telefonata(telefonata);
    telefonata Estrai_Una();
private:
    class nodo { // la classe nodo è interna e privata
public:
    nodo();
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
nodo(const telefonata&, nodo*);  
telefonata info;  
nodo* next;  
};  
nodo* first;  
};  
#endif  
  
// file "bolletta.cpp"  
#include "bolletta.h"  
  
// notare il costruttore di default per il campo dati info  
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); // aggiunge in testa alla lista  
}  
  
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() {  
    // Precondizione: la bolletta non è vuota  
    nodo* p = first;  
    first = first->next;  
    telefonata aux = p->info; // costruttore di copia  
    delete p;  
    return aux;  
}
```

3.1 Classi annidate

Nella definizione della classe `bolletta` notiamo che la classe `nodo`, i cui oggetti rappresentano i nodi della lista dinamica che rappresenta una bolletta, è definita internamente alla classe `bolletta`. Una classe definita internamente ad un'altra classe `C` viene detta *classe annidata* (in inglese nested class) in `C` o *classe interna* a `C`. Una classe `A` annidata in `C` è quindi nello scoping della classe `C` ed è a tutti gli effetti un membro della classe `C`. In particolare, la classe interna `A` può essere pubblica o privata. Se `A` è pubblica allora posso definire oggetti di `A` esternamente a `C` usando l'operatore di scoping, ad esempio con la dichiarazione

```
C::A obj;
```

Se `A` è invece privata allora il tipo `A` è inaccessibile esternamente a `C`. Nella classe annidata `A`, in particolare nel corpo dei suoi metodi, posso usare solamente i membri statici della classe contenitrice `C`. Ad esempio:

```
int x; // variabile globale

class C {
public:
    int x;
    static int s;
    class A {
        void f(int i) {
            int z = sizeof(x); // Illegale: x si riferisce a C::x
            x = i;             // Illegale: x si riferisce a C::x
            s = i;             // OK
            ::x = i;           // OK: assegno alla variabile globale x
        }
        void g(C* p, int i) {
            p->x = i; // OK
        }
    };
};
```

I membri della classe `A` annidata in `C` non hanno dei privilegi speciali di accesso ai membri di `C`, così come i membri di `C` non hanno dei privilegi speciali di accesso ai membri della classe `A`. Valgono quindi le usuali regole di (in)accessibilità. Vediamo un esempio (attenzione: la compilazione in `g++` di tale esempio potrebbe non seguire lo standard C++ — a tal proposito si veda la sezione 11.8 sulle nested classes del documento di standardizzazione 2003 — e potrebbe non segnalare le prime due illegalità).

```
class C {
private:
```

```
int x;
class B {};
class A {
private:
    B b; // Illegale: il tipo C::B è privato
    int y;
    void f(C* p, int i){
        p->x = i; // Illegale: il campo dati C::x è privato
    }
    int g(A* p) {
        return p->y; // Illegale: il campo dati A::y è privato
    }
};
```

3.2 Il problema dell'interferenza

Alcuni metodi della classe bolletta, ad esempio Aggiungi_Telefonata(), modificano l'oggetto di invocazione. Vedremo come in particolari circostanze questa caratteristica possa provocare degli inconvenienti e cosa si possa fare per evitarli. Supponiamo che per la classe bolletta sia stato ridefinito, come funzione esterna, l'operatore << che permette di stampare la lista delle telefonate. Consideriamo il seguente esempio di main():

```
int main() {
    bolletta b1; // costruttore senza argomenti
    telefonata t1(orario(9,23,12),orario(10,4,53),2121212);
    telefonata t2(orario(11,15,4),orario(11,22,1),3131313);
    b1.Aaggiungi_Telefonata(t1); b1.Aaggiungi_Telefonata(t2);
    cout << b1;
    bolletta b2;
    b2 = b1;
    b2.Togli_Telefonata(t1);
    cout << b1 << b2;
}
```

L'output sarà qualcosa del genere:

```
TELEFONATE IN BOLLETTA:
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:
```

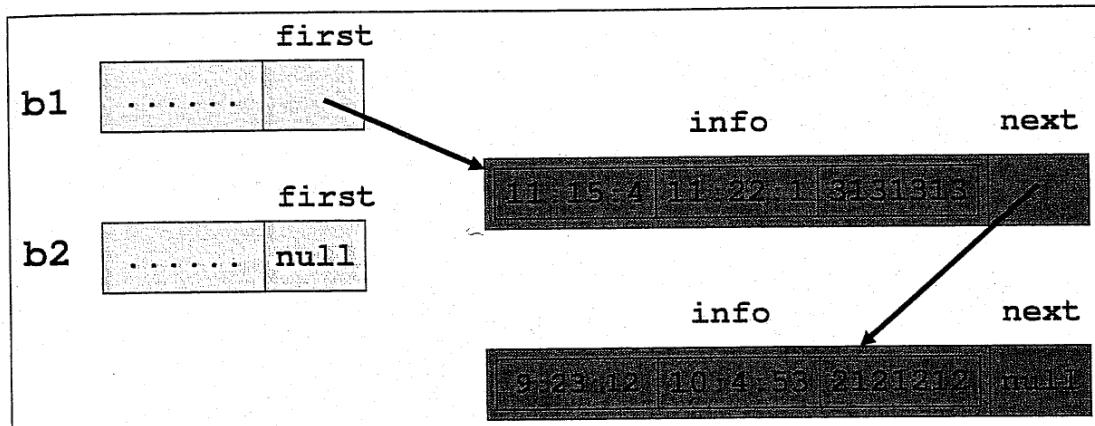
3.2 Il problema dell'interferenza

1) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313

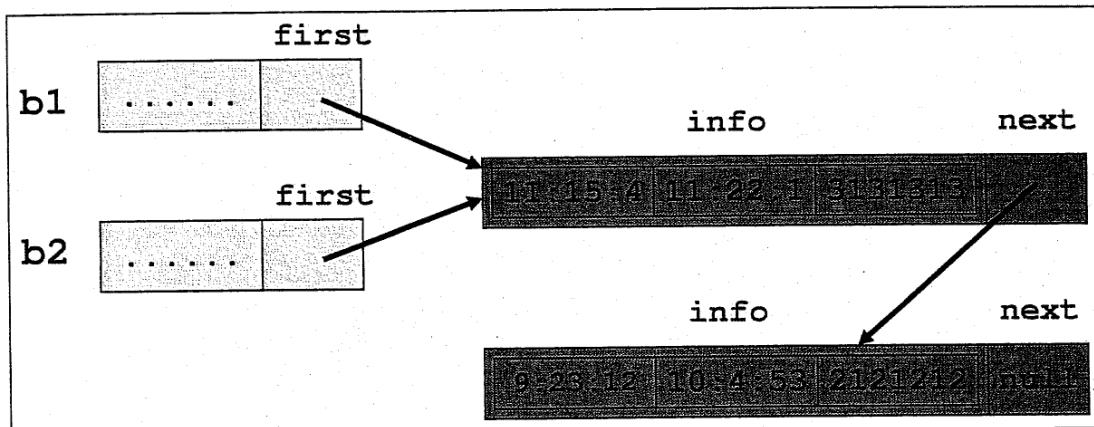
TELEFONATE IN BOLLETTA:

1) INIZIO 11:15:4 FINE 11:22:1 NUMERO 3131313

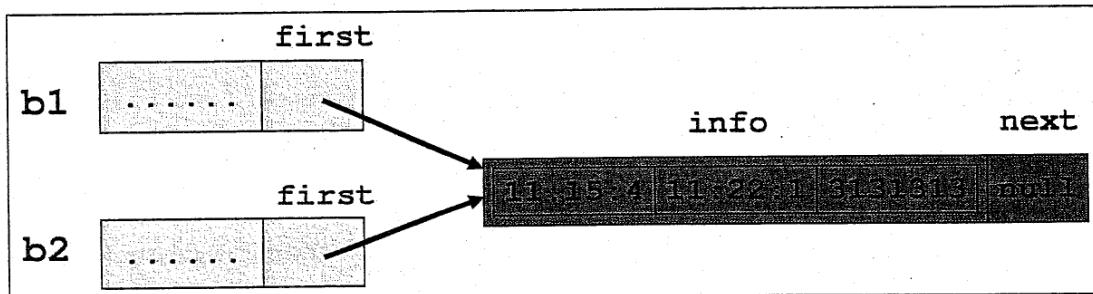
La situazione prima dell'assegnazione $b2 = b1$; è rappresentata dalla seguente figura.



La situazione dopo l'assegnazione $b2 = b1$; diventa quindi la seguente:

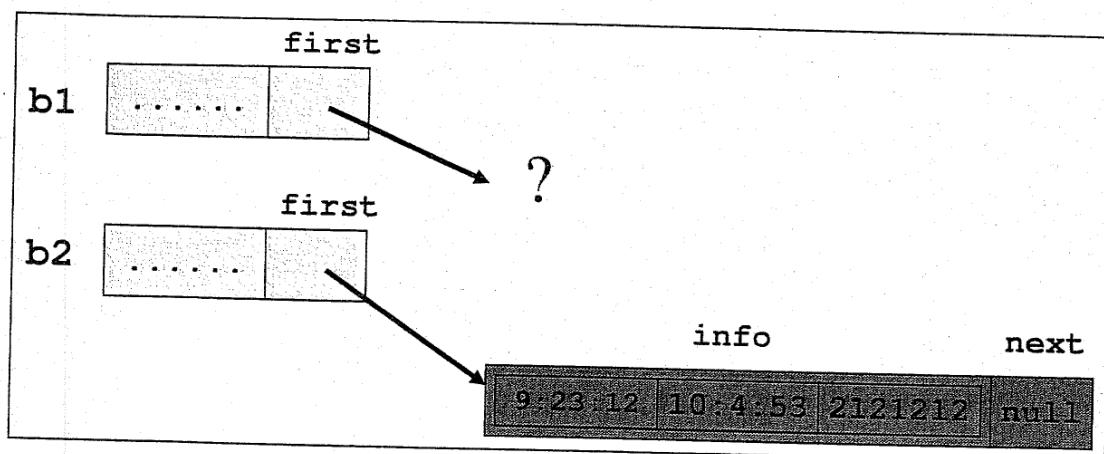


Infine, dopo $b2.\text{Togli_Telefonata}(t1)$; la situazione evolve nel seguente modo:



Se invece della telefonata t_1 avessimo rimosso dalla bolletta b_2 la telefonata t_2 la situazione sarebbe stata anche "peggiore":

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI



La spiegazione risiede nel fatto che l'assegnazione standard, naturalmente, copia i campi puntatore ma non gli oggetti a cui essi puntano. Avremmo avuto lo stesso problema anche con la costruzione di copia

bolletta b2 = b1;

al posto di

bolletta b2;

b2 = b1;

perché anche il costruttore di copia standard copia i campi puntatore ma non gli oggetti a cui essi puntano.

L'effetto sopra descritto si chiama *interferenza tra oggetti* o *aliasing*. Si può presentare anche in altre situazioni, ad esempio quando una funzione ha più parametri della stessa classe passati per valore e ad essi viene passato lo stesso oggetto. L'interferenza è dovuta a due cause concomitanti:

1. vi è condivisione di memoria tra gli oggetti;
2. vi sono funzioni che modificano gli oggetti.

Poiché generalmente non è possibile evitare che ci siano funzioni che modificano gli oggetti dobbiamo evitare o quanto meno gestire opportunamente la condivisione di memoria.

Per evitare la condivisione di memoria occorre ridefinire sia l'assegnazione che il costruttore di copia in modo tale che essi effettuino una cosiddetta *copia profonda*: ogni volta che copiano un puntatore devono eseguire una copia anche dell'oggetto puntato. Vediamo dapprima come si possano ridefinire assegnazione e costruttore di copia per evitare completamente la condivisione di memoria. Poiché una copia profonda può risultare molto costosa sia in termini di tempo di esecuzione che di memoria allocata (si pensi ad esempio al caso, non irrealistico, di liste con milioni di elementi!), vedremo in seguito una tecnica più raffinata mediante la quale è possibile avere una condivisione controllata della memoria che permette ugualmente di evitare le interferenze ma con un costo molto minore.

3.3 Copie profonde

3.3.1 Assegnazione profonda

L'operatore di assegnazione, che è binario ed infisso, si ridefinisce come ogni altro operatore dichiarando un metodo `operator=` che implementa come vogliamo venga eseguita l'assegnazione ridefinita. Occorre decidere il tipo dei parametri e il tipo restituito nella dichiarazione di `operator=`. Consideriamo l'assegnazione `x=y`, in cui `x` è un oggetto di qualche classe `C`. Supponiamo che l'assegnazione sia ridefinita internamente alla classe `C` come metodo. Come al solito, il primo operando `x` viene assunto come oggetto di invocazione dell'assegnazione, mentre avremo un parametro formale di tipo `C` per il secondo operando `y`. Dichiareremo tale parametro per riferimento al fine di evitare che venga effettuata inutilmente una copia del secondo operando `y` ed inoltre lo dichiareremo costante. Come al solito, l'assegnazione `x=y` restituisce l'`l`-valore di `x`. Ciò, in particolare, permette l'usuale composizione di assegnazioni come `x=y=z`. Pertanto il tipo del risultato sarà un riferimento ad un oggetto della stessa classe di `x`. La dichiarazione di `operator=` sarà quindi la seguente:

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

Aggiorniamo quindi la parte pubblica della classe `bolletta` come segue:

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

Vogliamo che l'assegnazione di `bolletta` effettui copie profonde. Per ottenere ciò aggiungiamo a `bolletta` due metodi statici di utilità che quindi dichiariamo privati: il metodo `copia()` che data una lista `L` restituisce una copia profonda di `L` e il metodo `distruggi()` che distrugge, cioè dealloca, una data lista. Aggiungiamo pertanto alla parte privata della classe `bolletta` le seguenti dichiarazioni:

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

D'altra parte, in `bolletta.cpp` aggiungiamo le seguenti definizioni.

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
bolletta::nodo* bolletta::copia(nodo* p) {
    if (!p) return 0;
    nodo* primo = new nodo;
    // invocazione del costruttore di default di nodo
    // primo punta al primo nodo della copia della lista
    primo->info = p->info;
    nodo* q = primo;
    // q punta all'ultimo nodo della lista finora copiata
    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) {
    // scorro tutta la lista deallocando ogni nodo
    nodo* q;
    while (p) {
        q = p;
        p = p->next;
        delete q;
    }
}
```

Oppure possiamo optare per le seguenti definizioni ricorsive di `copia` e `distruggi`.

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

void bolletta::distruggi(nodo* p) {
    // caso base: lista vuota, nulla da fare
    if (p) {
        // passo induttivo:
```

3.3 Copie profonde

```
// per induzione distruggi(p->next) dealloca la coda di p e
// quindi rimane da deallocare solamente il primo nodo di p
distruggi(p->next);
delete p;
}
```

Usando le funzioni private `copia` e `distruggi` possiamo allora fare un primo tentativo di definizione di `operator=`:

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

Tuttavia ci sono due problemi "sottili" con la precedente ridefinizione dell'assegnazione:

1. Quando si esegue l'assegnazione `y = x`; l'oggetto `y` naturalmente può già avere una sua lista di elementi e la memoria occupata da tale lista non viene deallocata.
2. Se si esegue una assegnazione `x = x`; viene effettuata inutilmente la copia della lista di elementi. Per chi ritiene molto improbabile che un programmatore esperto scriva esplicitamente `x = x`; in un programma... osserviamo che non è affatto improbabile che un'assegnazione tra due celle di un array come `a[i] = a[j]`; venga eseguita con `i == j`!

Per ovviare ad entrambi gli inconvenienti useremo la seguente definizione di `operator=`:

```
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

Chiaramente anche il costruttore di copia può creare condivisione di memoria e quindi interferenza. Se vogliamo evitare tale condivisione dobbiamo ridefinirlo. Aggiungiamo quindi alla parte pubblica della classe `bolletta` la dichiarazione:

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

mentre in `bolletta.cpp` inseriremo la definizione:

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

3.4 Oggetti come parametri di funzione

In questa sezione rivediamo i meccanismi per il passaggio dei parametri in C++, facendo attenzione al caso in cui gli argomenti siano degli oggetti.

Ogni volta che una funzione `F()` viene invocata, i suoi parametri formali vengono allocati in memoria (nello stack, più precisamente nel record di attivazione della funzione `F()`), e inizializzati con i corrispondenti parametri attuali. Come noto, il C++ prevede due meccanismi per il passaggio dei parametri, il *passaggio per valore* e il *passaggio per riferimento*.

Passaggio per valore. In questo caso i parametri formali vengono inizializzati tramite delle *copie* degli r-valori dei parametri attuali. Ad esempio, per una funzione con prototipo `T1 fun(T x)` ed una espressione `e` compatibile con il tipo `T` del parametro, nella chiamata `fun(e)` il parametro formale `x` viene inizializzato con il valore dell'espressione `e`. Quindi, quando `T` è un tipo classe l'inizializzazione del parametro `x` comporta la chiamata del costruttore di copia della classe `T`. È importante notare quindi che quando deve essere passato come argomento un oggetto che occupa parecchio spazio in memoria, il passaggio per valore potrebbe essere costoso sia in termini di tempo che di spazio. Inoltre, con questo meccanismo i valori manipolati dalla funzione sono copie dei parametri attuali locali alla funzione, quindi le modifiche fatte sui parametri formali non si riflettono sui parametri attuali.

Passaggio per riferimento. Si consideri il prototipo di funzione `T1 fun(T& x)`. Il parametro `x` è un riferimento ad una variabile di tipo `T` e quindi il parametro attuale `a` in una chiamata di funzione `fun(a)` deve essere un'espressione indirizzabile di tipo `T`, cioè `a` deve avere un l-valore. Ad esempio, `a` può essere una variabile di tipo `T` oppure il valore di tipo `T&` ritornato per riferimento da qualche funzione, mentre `a` non può essere un valore di tipo `T` ritornato per valore da qualche funzione, in quanto privo di l-valore (cioè non indirizzabile). Ciò che succede nell'invocazione `fun(a)` è che il parametro formale `x` diventa un alias del parametro attuale `a`, senza bisogno di creare alcuna copia locale alla funzione. Questo meccanismo permette quindi

3.4 Oggetti come parametri di funzione

alle funzioni di modificare i valori delle variabili passate per riferimento. Inoltre, questo meccanismo può essere utile quando vengono passati oggetti che occupano parecchio spazio in memoria.

Ad esempio, si considerino i prototipi di funzione T1 F(int& x) e int g(). Poiché il parametro formale x di F è un riferimento, le invocazioni F(2) e F(g()) provocano un errore in compilazione, in quanto i parametri attuali non sono indirizzabili. Per lo stesso motivo non è possibile definire una funzione con il seguente prototipo: T1 F(int& x=2).

Abbiamo osservato che la scelta del passaggio per riferimento di oggetti può essere dettata da ragioni di efficienza, quando la funzione non modifica il valore del parametro attuale. In questo caso è buona pratica dichiarare costante il parametro formale mediante l'attributo const, indicando quindi che la funzione non modifica il valore dell'argomento, come ad esempio nel prototipo T1 G(const T&). Si noti che il tipo const T& per il parametro formale permette alla funzione di essere invocata con un qualsiasi valore (anche non indirizzabile) di tipo T, incluso un oggetto anonimo restituito da qualche funzione, e non necessariamente con una variabile (ovvero una espressione indirizzabile). Ad esempio, per i prototipi di funzione T1 F(const int& x) e int g() si ha quindi che le invocazioni F(2) e F(g()) risultano ora corrette, così come il prototipo

T1 F(const int& x=2).

Notiamo inoltre che per un tipo classe T e un prototipo di funzione

T1 F(const T& x)

poichè il parametro formale x è un riferimento ad un oggetto costante, nel corpo della funzione si potranno invocare su x solamente i metodi costanti della classe T.

Dovrebbe essere quindi chiara l'importanza di aggiungere l'attributo const a quei parametri passati per riferimento che la funzione non modifica. Ad esempio, l'overloading dell'operatore di output per oggetti di una classe C come funzione esterna viene definito tipicamente con prototipo

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

Nel passaggio dei parametri per riferimento costante bisogna prestare attenzione alla seguente (in qualche modo sorprendente) regola del C++ standard che viene implementata a partire dalla versione 3.4 del compilatore g++ (sino alla versione 3.3 g++ era tollerante): se un riferimento costante viene inizializzato con un r-valore non indirizzabile (cioè senza valore) di un qualche tipo classe C allora il costruttore di copia di C deve essere accessibile. La regola è illustrata dal seguente esempio.

```
class C {  
public:  
    C() {...}  
private:  
    C(const C& c) {...} // costruttore di copia privato
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
};

C fun1() {...}           // ritorna C per valore

void fun2(const C& c) {...} // parametro c riferimento costante

void fun3(void) {
    fun2(C());          // Illegale, costruttore di copia inaccessibile
    fun2(fun1());        // Illegale, costruttore di copia inaccessibile
    C c;
    fun2(c);            // OK, c ha un l-valore
}
```

Vediamo ora due esempi relativi alla classe bolletta che illustrano il comportamento delle due diverse tipologie di passaggio dei parametri per gli oggetti di una classe. Vogliamo scrivere una funzione Somma_Durate esterna alla classe bolletta che data una bolletta b restituisca la somma delle durate delle telefonate in b. Definiamo Somma_Durate con un parametro di tipo bolletta passato per valore.

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

Se eseguiamo il seguente esempio di 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(t2); b1.Aggiungi_Telefonata(t1);
    cout << b1;
    cout << "LA SOMMA DELLE DURATE >> " << Somma_Durate(b1) << endl;
    cout << b1;
}
```

come ci si aspetta otteniamo correttamente un output analogo al seguente:

3.4 Oggetti come parametri di funzione

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

Ricordiamo che il metodo `Estrai_Una` provoca effetti collaterali sulla bolletta di invocazione. Tuttavia, il passaggio per valore dell'oggetto `b1` nell'invocazione `Somma_Durate(b1)` viene eseguito tramite una chiamata al costruttore di copia di bolletta che costruisce il parametro formale (e oggetto locale alla funzione `Somma_Durate`) `b`, che, per come lo abbiamo ridefinito, effettua una copia profonda della lista delle telefonate. Non vi è quindi condivisione di memoria tra l'oggetto `b1` e l'oggetto `b` locale alla funzione `Somma_Durate`. Quindi, la funzione `Somma_Durate` benché estragga dalla bolletta `b` tutte le telefonate svuotandola completamente, lascia invariato il parametro attuale `b1` nell'invocazione `Somma_Durate(b1)`.

Consideriamo ora la seguente funzione esterna `Chiamate_A` che rimuove dal parametro `b` di tipo `bolletta` passato per riferimento tutte le telefonate al numero `num` e restituisce una nuova bolletta contenente le telefonate rimosse.

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

Se eseguiamo il seguente esempio di `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);
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
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;  
}
```

come ci si aspetta, correttamente otteniamo un output analogo al seguente:

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

Tuttavia è importante notare il seguente problema nella precedente funzione `Chiamate_A`. Essa crea i due oggetti locali `selezionate` e `resto` che, come ogni oggetto locale, esistono solo durante l'esecuzione della funzione. La funzione memorizza nell'oggetto `resto` la lista delle telefonate non selezionate. Alla fine `resto` viene assegnato al parametro `b`. Per come abbiamo ridefinito l'assegnazione, essa effettua una copia profonda della lista. Quando la funzione termina non viene recuperata la memoria dinamica allocata per la lista associata a `resto`. Inoltre, `Chiamate_A` memorizza nell'oggetto `selezionate` la lista delle telefonate selezionate. Alla fine l'oggetto `selezionate` viene ritornato. L'istruzione `return` usa il costruttore di copia per costruire un oggetto anonimo che verrà distrutto non appena usato come valore di ritorno. Siccome anche il costruttore di copia effettua una copia profonda non viene recuperata la memoria dinamica di due liste: quella di `selezionate` e quella dell'oggetto anonimo. Quindi ci sono tre liste allocate dinamicamente che non vengono deallocate! Si tratta di uno spreco di memoria che un buon programma non può assolutamente permettersi. Vedremo come porre rimedio a ciò.

3.5 Tempo di vita delle variabili

Il *tempo di vita* di una variabile di qualsiasi tipo è l'intervallo di tempo in cui la variabile viene mantenuta in memoria durante l'esecuzione del programma.

- Le *variabili di classe automatica* sono quelle definite all'interno di qualche blocco, tipicamente il blocco che definisce una funzione: esse vengono allocate nello stack quando l'esecuzione raggiunge la loro definizione e vengono deallocate quando termina l'esecuzione del blocco in cui sono definite.
- Le *variabili di classe statica* sono le variabili globali (con "scope globale"), cioè definite all'esterno di ogni funzione e classe, i campi dati statici e le variabili statiche definite in qualche blocco: le prime due categorie di variabili di classe statica vengono allocate all'inizio dell'esecuzione del programma mentre le variabili statiche interne ad un blocco vengono allocate quando l'esecuzione raggiunge la loro definizione. Tutte le variabili di classe statica vengono deallocate al termine dell'esecuzione del programma.
- Le *variabili dinamiche* sono sempre allocate nello heap, l'area della memoria dinamica gestita dal programmatore: vengono allocate quando viene eseguito l'operatore `new` e vengono deallocate quando viene invocato l'operatore `delete`.

Notiamo che sono variabili di classe automatica i parametri formali passati per valore di una funzione e tutte le variabili (non statiche) locali di una funzione. Sono invece variabili di classe statica le variabili locali ad una funzione dichiarate statiche. Le variabili con lo stesso tempo di vita — tipicamente variabili definite nello stesso blocco oppure campi dati statici di una classe — vengono deallocate nell'ordine inverso a quello in cui sono state allocate. Per variabili che sono oggetti di qualche classe, allocazione significa invocazione di un costruttore mentre deallocazione corrisponde all'invocazione del distruttore.

3.6 Distruttore

Possiamo evitare gli sprechi di memoria osservati con la funzione `Chiamate_A` della Sezione 3.4 usando un particolare meccanismo del C++. Quando termina il tempo di vita di un oggetto di qualche classe viene richiamato automaticamente un particolare metodo detto *distruttore*. Come per i costruttori anche per i distruttori è previsto un *distruttore standard* per ogni classe. Il distruttore standard su un oggetto `x` semplicemente rilascia la memoria occupata da `x`. Bisogna quindi prestare molta attenzione al fatto che se l'oggetto `x` include dei campi dati puntatore allora il distruttore standard si limita a rilasciare la memoria occupata dai campi puntatore ma non dealloca la memoria a cui puntano i campi puntatore di `x` e questo comportamento del distruttore standard molto spesso non è corretto. È possibile però ridefinire il distruttore in modo che esso effettui una cosiddetta distruzione profonda degli oggetti, cioè in modo che il distruttore deallochi anche la memoria a cui puntano i campi puntatore. Ad esempio, se un oggetto `x` include un campo dati `p` che punta alla testa di una lista allora la distruzione profonda di `x` deallocherà tutta la lista puntata da `p`. Il concetto di distruttore standard è generalizzato anche ai tipi non classe, cioè i tipi primitivi o derivati: il distruttore standard per un tipo non classe si limita a rilasciare la memoria allocata.

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

Il distruttore è un metodo senza parametri e senza tipo di ritorno identificato dal nome della classe preceduto dal simbolo “~” (“ondina” oppure “tilde”, denotato nel codice dal carattere tipografico “~”). Il distruttore della classe bolletta si ridefinisce quindi inserendo nella parte pubblica della classe la dichiarazione:

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

La definizione di questo distruttore profondo sarà la seguente:

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

Abbiamo visto che il distruttore di oggetti viene invocato automaticamente al termine del tempo di vita di un oggetto. Di conseguenza, le regole di invocazione dei distruttori sono le seguenti:

1. Per gli oggetti di classe statica, al termine del programma (all'uscita dalla funzione principale `main()`).
2. Per gli oggetti di classe automatica definiti in un blocco (ad esempio una funzione), all'uscita dal blocco in cui sono definiti; in particolare, ciò vale per i parametri formali di una funzione.
3. Per gli oggetti dinamici (allocati sullo heap), quando viene eseguito l'operatore `delete` sui corrispondenti puntatori, altrimenti al termine del programma.
4. Per gli oggetti che sono campi dati di qualche oggetto `x`, quando `x` viene distrutto.
5. Gli oggetti con lo stesso tempo di vita, tipicamente oggetti definiti nello stesso blocco oppure oggetti statici di una classe, vengono distrutti nell'ordine inverso a quello in cui sono stati creati.

In particolare, il distruttore viene invocato nei seguenti casi:

1. sulle variabili locali di una funzione al termine dell'esecuzione della funzione;
2. sui parametri di una funzione passati per valore al termine dell'esecuzione della funzione;

3.6 Distruttore

3. sull'oggetto anonimo ritornato come risultato da una funzione non appena esso sia stato usato (se qualche ottimizzazione prolunga il tempo di vita dell'oggetto anonimo ritornato significa che non viene invocato il distruttore).

In particolare, la distruzione al ritorno di una chiamata di funzione `F()` segue questo ordine:

1. vengono distrutte le variabili locali a `F()`;
2. viene distrutto l'oggetto anonimo ritornato per valore da `F()` non appena sia stato usato;
3. vengono distrutti i parametri di `F()` passati per valore.

Menzioniamo inoltre il seguente fatto: per una funzione `F(T1 f1, ..., Tn fn)` con più di un parametro formale, in ogni invocazione `F(a1, ..., an)` l'ordine seguito per passare i parametri attuali `a1, ..., an` è da destra verso sinistra nella lista dei parametri di `F()`, cioè il passaggio dei parametri comporta la seguente lista di inizializzazione per i parametri di `F()`:

`Tn fn = an; ...; T1 f1 = a1;`

Conseguentemente, poiché al momento della distruzione dei parametri passati per valore l'ordine è inverso all'ordine del passaggio dei parametri, l'ordine di distruzione dei parametri passati per valore è da sinistra verso destra nella lista dei parametri di `F()`. Illustriamo questi casi tramite il seguente esempio.

```
class A {
    int x;
public:
    ~A() { cout << "~A(" << x << ") " ; }
    A(int z): x(z) { cout << "A(" << x << ") " ; }
    A(const A& a): x(a.x+5) { cout << "Ac(" << x << ") " ; }
};

A Fun(A f1, A f2) { A loc(3); return f1; }

int main() {
    A a1(1), a2(2); // stampa: A(1) A(2)
    Fun(a1,a2);
    /* Stampa:
     * Ac(7)  costruzione di copia del parametro f2 da a2
     * Ac(6)  costruzione di copia del parametro f1 da a1
     * A(3)   costruzione oggetto loc locale a Fun
     * Ac(11) costruzione di copia dell'oggetto anonimo ritornato
     * ~A(3)  distruzione oggetto loc locale a Fun
    */
}
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
~A(11)    distruzione dell'oggetto anonimo ritornato
~A(6)     distruzione parametro f1
~A(7)     distruzione parametro f2
*/
cout << "**** ";
/* Stampa:
~A(2)    distruzione oggetto a2 del main()
~A(1)    distruzione oggetto a1 del main()
*/
}
```

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

1. innanzitutto viene eseguito il corpo del distruttore della classe C, se questo esiste;
2. vengono quindi richiamati i distruttori per i campi dati nell'ordine inverso alla loro lista di dichiarazione, cioè nell'ordine x_n, \dots, x_1 . Per un campo dati di tipo non classe (cioè tipo primitivo o derivato) viene semplicemente rilasciata la memoria (come abbiamo visto, possiamo pensare che questo sia il comportamento del “distruttore standard” per i tipi non classe) mentre per i tipi classe viene invocato il distruttore, standard oppure ridefinito.

Il distruttore standard semplicemente ha il corpo vuoto. Quindi il distruttore standard di una classe C si limita a richiamare i distruttori per i campi dati di C nell'ordine inverso a quello di dichiarazione.

Il distruttore di bolletta da noi ridefinito viene quindi richiamato automaticamente al termine dell'esecuzione della funzione Chiamate_A per distruggere gli oggetti locali selezionate e resto e l'oggetto anonimo ritornato da Chiamate_A non appena esso sia stato usato. Il distruttore richiama il metodo statico privato distruggi che a sua volta esegue l'istruzione delete su tutti gli elementi della lista. Analizziamo dettagliatamente ciò che succede.

1. L'istruzione delete p; provoca l'invocazione del distruttore della classe nodo e poiché esso non è stato ridefinito viene richiamato il distruttore standard di nodo.
2. Il distruttore standard della classe nodo richiama a sua volta per ogni campo dati di nodo, cioè info e next, i rispettivi distruttori. Il campo puntatore next viene semplicemente deallocato, mentre per il campo info viene invocato il distruttore della classe telefonata, e siccome neanch'esso è stato ridefinito verrà richiamato il distruttore standard di telefonata.
3. Il distruttore standard della classe telefonata dealloca l'intero numero e richiama il distruttore della classe orario per i campi inizio e fine, che sarà quindi il distruttore standard di orario.

3.6 Distruttore

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

Poiché le classi telefonata e orario non hanno campi dati puntatore mentre la classe nodo ha il campo dati next puntatore al nodo successivo che è stato distrutto con la chiamata distruggi(*p->next*) ; prima che venga richiamata la delete su tale nodo, il comportamento dei distruttori standard risulta adeguato. Un modo più elegante di procedere sarebbe stato quello di ridefinire anche il distruttore per la classe nodo in modo tale che esso stesso provveda a distruggere l'oggetto puntato da next. In questo caso, la definizione del distruttore della classe nodo diventa quindi la seguente:

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

In questo modo il distruttore di bolletta non richiede più la chiamata alla funzione distruggi, ma è sufficiente definirlo nel seguente modo:

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

Attenzione: la precedente definizione del distruttore della classe nodo modifica il comportamento del metodo Estrai_Una(). Infatti l'istruzione delete *p*; causa la distruzione profonda della lista di nodi puntata dal campo dati first dell'oggetto di invocazione. Quindi una invocazione di Estrai_Una() provoca la distruzione profonda della bolletta di invocazione! Una possibile soluzione consiste nella seguente modifica del metodo Estrai_Una().

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

Esercizio 3.6.1. I seguenti programmi compilano correttamente e sono compilati con l'opzione -fno-elide-constructors. Quali stampe produce la loro esecuzione? Attenzione: senza l'opzione -fno-elide-constructors potrebbero produrre un output diverso.

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
// PROGRAMMA UNO

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

// funzione esterna, passaggio del parametro per valore
C F(C p) { return p; }

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

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

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

```
// PROGRAMMA DUE

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

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

C w("3");

class D {
public:
    static C c;
```

3.6 Distruttore

```
};

C D::c("4");

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

```
// PROGRAMMA TRE

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

// passaggio del parametro e valore ritornato per riferimento
C& F(C& p) { return p; }

C w("3");

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

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

```
// PROGRAMMA QUATTRO

class A {
private:
    int z;
public:
    ~A() {cout << "Ad ";}
};
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
class B {
public:
    A* p;
    A a;
    ~B() {cout << "Bd ";}
};

class C {
public:
    static B s;
    int k;
    A a;
    ~C() {cout << "Cd ";}
};
B C::s=B();

int main() {
    C c1, c2;
}
```

3.7 Nascondere la parte privata di una classe

Supponiamo di aver definito una qualsiasi classe C e di voler nascondere la parte privata della dichiarazione di C all'utente finale della classe.

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

Dichiariamo una classe C_privata internamente e privatamente alla classe C_handle che conterrà la parte privata della classe C, e quindi dichiariamo nella parte privata di C_handle un puntatore ad una tale classe, mentre la parte pubblica di C rimane invariata in C_handle.

```
// file "C_handle.h"
class C_handle {
public:
    // parte pubblica
private:
    class C_privata;
    C_privata* punt;
};
```

3.7 Nascondere la parte privata di una classe

Quindi, nell'implementazione separata di `C_handle` definiremo la classe `C_privata` contenente la parte privata di `C`.

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

Naturalmente occorre modificare le definizioni dei metodi di `C` tenendo conto che per accedere alla parte privata dobbiamo usare il puntatore. Il file header `C_handle.h` fornito all'utente finale conterrà soltanto la definizione della classe `C_handle` mentre la definizione della classe `C_privata` che contiene la parte privata di `C` e la definizione dei metodi risiedono nel file `C_handle.cpp` che viene compilato separatamente fornendo all'utente finale soltanto il file oggetto `C_handle.o` prodotto dalla compilazione. Illustriamo questa tecnica sulla classe `orario`.

```
// file "orario_handle.h"
#ifndef ORARIO_HANDLE_H
#define ORARIO_HANDLE_H
#include <iostream>
using std::ostream;

class orario_handle {
public:
    orario_handle(int = 0, int = 0, int = 0);
    int Ore() const;
    int Minuti() const;
    int Secondi() const;
    void AvanzaUnOra();
private:
    class orario_rappr;
    orario_rappr* punt;
};

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

#endif
```

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

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

```

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

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

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

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

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

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

```

Attenzione: con questa tecnica si possono verificare problemi di interferenza, visto che le classi hanno un puntatore come campo dati. Il seguente frammento di codice mostra questo fenomeno.

```

orario_handle o1(17,11,27), o2;
cout << o1 << ' ' << o2 << endl;
// stampa: 17:11:27 0:0:0
o2=o1;           // stesso puntatore punt in o2 e o1
o1.AvanzaUnOra(); // interferenza
cout << o1 << ' ' << o2 << endl;
// stampa: 18:11:27 18:11:27

```

Sarà quindi necessario ridefinire adeguatamente l'assegnazione e il costruttore di copia di `orario_handle`, con la tecnica di copia profonda già illustrata, per evitare interferenze indesiderate.

3.8 Array di oggetti

Naturalmente è possibile definire array statici o dinamici di oggetti. Come vengono costruiti e distrutti? Per la costruzione viene richiamato implicitamente il costruttore di default,

3.9 Cast

standard o ridefinito, per ogni oggetto dell'array (che quindi deve essere disponibile). Per la distruzione viene richiamato implicitamente il distruttore (standard o ridefinito) per ogni oggetto dell'array.

Esempio 3.8.1.

```
class C {
public:
    int i;
    C(int x=3): i(x) {cout << "C01 ";}
};

int main() {
    C a[4]; cout << endl;           // stampa: C01 C01 C01 C01
    C* ad = new C[2]; cout << endl; // stampa: C01 C01
    cout << a[0].i << a[1].i << a[2].i << a[3].i << endl;
                                         // stampa: 3333
    cout << ad->i << (ad+1)->i << endl; // stampa: 33
}
```

Per un array a di tipo C è possibile costruire mediante il costruttore di copia gli oggetti di a a partire da una lista di inizializzazione di oggetti di C, analogamente a quanto accade per gli array di tipo non classe. Se la lista di inizializzazione non è completa allora per gli elementi dell'array a per cui non viene fornito un oggetto di C per la costruzione di copia, viene richiamato il costruttore di default (che quindi deve essere disponibile). Si consideri il seguente esempio.

```
class C {
public:
    int i;
    C(int x=3): i(x) {}
};

int main() {
    C x; C y(5);
    C a[4] = {x, C(), y};
    cout << a[0].i << a[1].i << a[2].i << a[3].i << endl;
    // stampa: 3353
}
```

3.9 Cast

La notazione “()” ereditata dal C per le *conversioni esplicite (explicit cast)* è mantenuta per scopi di compatibilità in C++ ed è illustrata dal seguente esempio:

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
float x; char c; void* p;  
int i = (int) x;  
float y = (float) c;  
nodo* q = (nodo*) p;
```

Poiché le conversioni di tipo sono operazioni potenzialmente “pericolose” (ad esempio possono portare a perdite di informazione), il C++ standard ha introdotto una notazione esplicita più visibile e differenziata tra le varie tipologie di cast. La sintassi del cast del C++ standard è la seguente:

```
nome_cast <Tipo> (Expr)
```

in cui nome indica la tipologia di cast, che può essere `const`, `static`, `reinterpret` o `dynamic`, mentre `Tipo` denota il tipo in cui deve essere convertito il valore dell'espressione `Expr`. `Tipo` viene anche detto *target type* del cast dell'espressione `Expr`. La sintassi richiede che l'espressione da convertire `Expr` compaia sempre tra parentesi tonde.

Static cast.

```
static_cast <Tipo> (Expr)
```

Lo static cast permette di rendere esplicito l'uso di tutte le conversioni, implicite o meno, previste e/o permesse dal linguaggio o definite dal programmatore. Tutte le conversioni oggetto dello static cast si basano su informazione di tipo statica, cioè disponibile a compile-time. Lo static cast rende visibile l'uso della conversione in situazioni potenzialmente pericolose quali conversioni tra tipi predefiniti con perdita di informazioni (dette anche “narrowing conversion”, ad esempio da `long` a `int`) che usualmente provocano solo un avviso (warning) da parte del compilatore. Usando lo `static_cast` tali segnalazioni di warning vengono evitate. La conversione esplicita `static_cast` permette anche il cast (sempre molto pericoloso) tra puntatori a tipi qualsiasi (sempre che non venga rimosso l'attributo `const`) e in particolare quello dal tipo puntatore generico `void*` a `Tipo*`. Le conversioni implicite “safe”, cioè senza perdita di informazione, sono riassunte nella seguente tabella, dove la notazione $T_1 \Rightarrow T_2$ significa che vi è una conversione implicita dal tipo T_1 al tipo T_2 .

$T\& \Rightarrow T$
$T[] \Rightarrow T^*$
$T \Rightarrow \text{const } T$
$T^* \Rightarrow \text{const } T^*$
$\text{float} \Rightarrow \text{double} \Rightarrow \text{long double}$
$\text{char} \Rightarrow \text{short} \Rightarrow \text{int} \Rightarrow \text{long}$
$\text{unsigned char} \Rightarrow \dots \Rightarrow \text{unsigned long}$

Esempio 3.9.1.

3.9 Cast

```
// Esempio di narrowing conversion
double d = 3.14;
int x = static_cast<int>(d);
// Esempio di castless conversion
char c = 'a';
int x = static_cast<int>(c);
// Esempio di conversione void* => T*
void* p; p=&d;
double* q = static_cast<double*>(p);
// Esempio di castless conversion
int* r = static_cast<int*>(q);
```

Const cast.

```
const_cast <T*> (puntatore costante)
const_cast <T&> (riferimento costante)
```

Il `const_cast` permette di convertire un puntatore/riferimento ad un tipo `const T` ad un puntatore/riferimento a `T` (rimuovendo quindi l'attributo `const`). Il compilatore segnala un errore se si tenta di usare uno degli altri operatori di conversione del C++ per rimuovere l'attributo `const`.

Esempio 3.9.2.

```
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, cast implicito
```

Reinterpret cast.

```
reinterpret_cast <T*> (puntatore)
reinterpret_cast <T&> (riferimento)
```

`reinterpret_cast` si limita a reinterpretare a basso livello la sequenza di bit con cui è rappresentato il valore puntato da puntatore come fosse un valore di tipo `T`. Quindi, `reinterpret_cast` permette ogni tipologia di conversione tra puntatori. Inoltre, questa conversione vale anche per i riferimenti. Come è facile intuire questo tipo di cast è particolarmente pericoloso e va usato con molta cautela ed in situazioni di sicurezza. Permette

situazioni estreme e con poco senso, ad esempio quelle evidenziate nel seguente frammento di codice:

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

Dynamic cast.

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

Nel caso del `dynamic_cast` il cosiddetto “tipo dinamico” di puntatore o riferimento non è noto a tempo di compilazione ma soltanto a tempo di esecuzione. Si tratta quindi di una conversione che avviene dinamicamente a tempo di esecuzione. Questa tipologia di conversione verrà approfonditamente analizzata nel Capitolo 6 nel contesto dell’ereditarietà fra classi.

3.10 Funzioni amiche

Supponiamo di voler definire l’operatore di output `operator<<` per la classe `bolletta`. Abbiamo visto, nel caso dalla classe `orario`, l’opportunità di dichiarare tale operatore come funzione esterna alla classe. Procederemo quindi allo stesso modo per la classe `bolletta`.

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

Questa soluzione è certamente corretta in quanto usa soltanto i metodi pubblici della classe `bolletta` e l’operatore `operator<<` della classe `telefonata`. Inoltre, benché questa definizione svuoti la bolletta `b`, tale bolletta è una copia profonda della bolletta passata come parametro attuale che non viene quindi modificata. Analizziamo un po’ più a fondo questa definizione di `operator<<`. Notiamo che dal punto di vista dell’efficienza questa soluzione non è certamente soddisfacente: effettuare una copia profonda di una lista

soltanto allo scopo di percorrerla per stamparne gli elementi è certamente uno spreco. Il problema è che una funzione esterna non ha accesso ai campi privati di una classe e quindi non può percorrere la lista. Nel caso di `bolletta`, non abbiamo accesso al puntatore `first`. Impedire l'accesso a tali campi agli utenti della classe riservandolo soltanto al progettista della classe è certamente ragionevole. Ma la funzione `operator<<`, benché sia stata dichiarata come funzione esterna alla classe, è da considerarsi a tutti gli effetti un membro della classe la cui definizione è a carico del progettista della classe. Il C++ permette al progettista della classe di dichiarare una funzione esterna quale `operator<<` come *funzione amica* della classe, permettendo quindi l'accesso alla parte privata della classe e allo stesso tempo preservando quelle caratteristiche vantaggiose osservate in precedenza di cui godono le funzioni esterne. Per fare questo basta mettere all'interno della classe `bolletta` una dichiarazione della funzione esterna `operator<<` preceduta dalla keyword `friend`. La definizione della funzione `operator<<` risiede invece nel file `bolletta.cpp` assieme alle definizioni dei metodi propri di `bolletta`.

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

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

3.11 Classi amiche e iteratori

Nella manipolazione degli oggetti di una classe collezione si avverte ben presto la necessità di poter accedere agli elementi della collezione, ad esempio per poter scorrere tutti gli elementi della collezione. È quindi opportuno che il progettista della classe fornisca all'utente gli strumenti per accedere agli elementi della collezione. Ovviamente tale accesso non deve permettere all'utente di accedere direttamente alla parte privata della classe collezione.

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

Il progettista della classe collezione può allo scopo definire una nuova *classe iteratore* i cui oggetti sono "indici" di elementi della classe contenitore. Si tratta della stessa idea degli indici interi che permettono di scorrere un array e degli iteratori sui contenitori della libreria STL.

Consideriamo la seguente semplice classe collezione di interi.

```
class contenitore {  
private:  
    class nodo {  
        public: // public per convenienza, nessuna influenza esterna  
        int info;  
        nodo* next;  
        nodo(int x, nodo* p): info(x), next(p) {}  
    };  
  
    nodo* first; // puntatore al primo nodo della lista  
  
public:  
    contenitore(): first(0) {}  
    void aggiungi_in_testa(int x) {first = new nodo(x,first);}  
};
```

Vogliamo definire una classe *iteratore* i cui oggetti rappresentano degli indici ai nodi degli oggetti della classe *contenitore*. Quindi, *iteratore* avrà come unico campo dati un puntatore alla classe *nodo* interna alla classe *contenitore*. Verranno inoltre ridefiniti gli operatori di uguaglianza e di incremento prefisso. Una prima versione di *iteratore* sarà quindi la seguente.

```
class iteratore {  
private:  
    contenitore::nodo* punt; // nodo puntato dall'iteratore  
public:  
    bool operator==(iteratore i) const {  
        return punt == i.punt;  
    }  
    iteratore& operator++() { // operator++ prefisso  
        if (punt) punt = punt->next;  
        return *this;  
    }  
};
```

Tuttavia, sia il campo dati *punt* che i metodi di questa definizione della classe *iteratore* accedono alla parte privata della classe *contenitore*, ovvero alla classe interna *nodo* ed alla sua rappresentazione. Useremo pertanto lo strumento dell'*amicizia tra classi*. Il C++

3.11 Classi amiche e iteratori

mette a disposizione la dichiarazione di amicizia a livello di classi: dichiarando che una classe C è amica di una classe D, si garantisce a tutti i membri della classe C l'accesso alla parte privata di D, in particolare si ottiene che tutti i metodi della classe C possano accedere alla parte privata di D. *Attenzione:* la relazione di amicizia tra classi non è né simmetrica né transitiva. Inoltre, definiremo la classe iteratore come classe pubblica interna alla classe contenitore: stiamo progettando una classe di iteratori per oggetti della classe contenitore, quindi la classe iteratore progettualmente fa parte delle funzionalità offerte da contenitore. Si tratta di un approccio standard, a cui aderiscono anche i contenitori della libreria STL.

Lo schema della classe contenitore sarà pertanto il seguente.

```
class contenitore {  
    friend class iteratore; // dichiarazione di amicizia  
private:  
    class nodo {  
        ...  
    };  
    nodo* first;  
public:  
    class iteratore {  
        private:  
            contenitore::nodo* punt; // per amicizia  
        ...  
    };  
    contenitore();  
    void aggiungi_nodo(int x);  
};
```

La classe contenitore metterà a disposizione dei metodi pubblici per definire gli iteratori “iniziali” e “finali” su un certo oggetto e ridefinirà l’operatore di indicizzazione operator[] con un parametro it di tipo iteratore per accedere all’informazione memorizzata nell’elemento indicizzato da it. Questi metodi naturalmente dovranno accedere al puntatore privato punt della classe iteratore: garantiremo tale accesso dichiarando la classe contenitore amica della classe iteratore.

```
class contenitore {  
    friend class iteratore; // dichiarazione di amicizia  
private:  
    class nodo { ... };  
    nodo* first;  
public:  
    class iteratore {  
        friend class contenitore; // dichiarazione di amicizia  
        ...  
    }; // Attenzione: la dichiarazione della classe iteratore
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
// deve precedere le dichiarazioni metodi che usano tale  
// tipo nella loro segnatura  
...  
iteratore begin() const;  
iteratore end() const;  
int& operator[](iteratore) const;  
};
```

Questi metodi d'utilizzo degli iteratori nella classe contenitore avranno quindi le seguenti semplici definizioni.

```
contenitore::iteratore contenitore::begin() const {  
    iteratore aux;  
    aux.punt = first; // per amicizia  
    return aux;  
}  
  
contenitore::iteratore contenitore::end() const {  
    iteratore aux;  
    aux.punt = 0; // per amicizia  
    return aux;  
}  
  
int& contenitore::operator[](contenitore::iteratore it) const {  
    return it.punt->info; // per amicizia  
}
```

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

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

Applichiamo ora l'idea della classe iteratore alla nostra classe bolletta.

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

3.11 Classi amiche e iteratori

```
friend class iteratore;
private:
    class nodo {
public:
    nodo();
    nodo(const telefonata& x, nodo* p): info(x), next(p) {}
    telefonata info;
    nodo* next;
    ~nodo();
};
nodo* first; // puntatore al primo nodo della lista
static nodo* copia(nodo* );
static void distruggi(nodo* );
public:
    class iteratore {
        friend class bolletta;
private:
    bolletta::nodo* punt; // nodo puntato dall'iteratore
public:
    bool operator==(iteratore) const;
    bool operator!=(iteratore) const;
    iteratore& operator++(); // operator++ prefisso
    iteratore operator++(int); // operator++ postfisso
}; // end classe iteratore
bolletta(): first(0) {}
~bolletta(); // distruttore profondo
bolletta(const bolletta&); // copia profonda
bolletta& operator=(const bolletta&); // assegnazione profonda
bool Vuota() const;
void Aggiungi_Telefonata(telefonata t);
void Togli_Telefonata(telefonata t)
telefonata Estrai_Una();
// metodi che usano iteratore
iteratore begin() const;
iteratore end() const;
telefonata& operator[](iteratore) const;
};
#endif
```

Si noti che abbiamo ridefinito l'operatore di incremento `operator++` sia prefisso che postfisso. L'argomento fittizio `int` della versione postfissa è un mero simbolo per distinguere i due operatori. Le definizioni mancanti dei metodi sono le seguenti.

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

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

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

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

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

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

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

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

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

A questo punto l'utente della classe può calcolare la somma delle durate delle telefonate di una bolletta con la seguente semplice funzione, in cui il parametro è passato per riferimento.

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

3.12 Dichiarazioni incomplete di classi

```
durata = durata + b[it].Fine() - b[it].Inizio();  
return durata;  
}
```

Esercizio 3.11.1. Ridefinire l'operatore “telefonata& operator*()” come metodo della classe iteratore cosicché la funzione Somma_Durate possa essere scritta nel seguente modo:

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

3.12 Dichiarazioni incomplete di classi

Una classe C può usare puntatori e riferimenti ad una classe D che non è definita ma che è solamente dichiarata tramite una cosiddetta *dichiarazione incompleta*, ovvero una mera dichiarazione del nome della classe D. Ad esempio, nella classe C possono essere definiti un campo dati di tipo D*, un metodo con un argomento di tipo D* o D& o che restituisce un D*. Il motivo per cui in questi casi basta una dichiarazione incompleta dovrebbe essere chiaro: la dichiarazione incompleta di D è sufficiente per la compilazione della classe C perché la memoria da allocare per puntatori e riferimenti è indipendente dal tipo puntato o a cui ci si riferisce e quindi al compilatore basta solamente conoscere il nome di tale tipo. Naturalmente, la dichiarazione incompleta di D dovrà essere visibile alla classe C, cioè dovrà precedere la dichiarazione della classe C.

```
class D; // dichiarazione incompleta di D  
  
class C {  
    D* p;  
    D* m() { ... }  
    void n(D*) { ... }  
    D& f() { ... }  
};  
  
class D {  
    // dichiarazione completa di D  
};
```

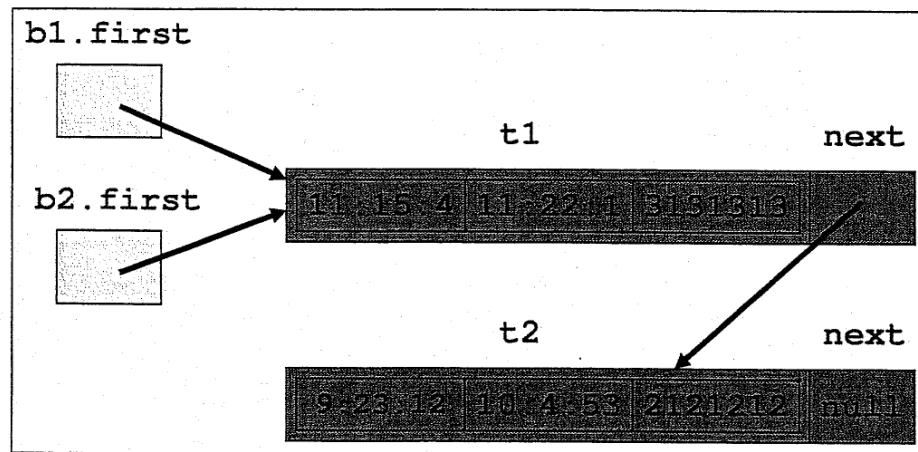
Le dichiarazioni di amicizia all'interno di una classe giocano anche il ruolo di dichiarazioni incomplete, cioè in una classe C una dichiarazione che la classe D è amica di C funge anche

da dichiarazione incompleta di D per tutti i membri della classe C (ma non all'esterno di C). Vedremo nel seguito vari esempi dell'utilità delle dichiarazioni incomplete di classi.

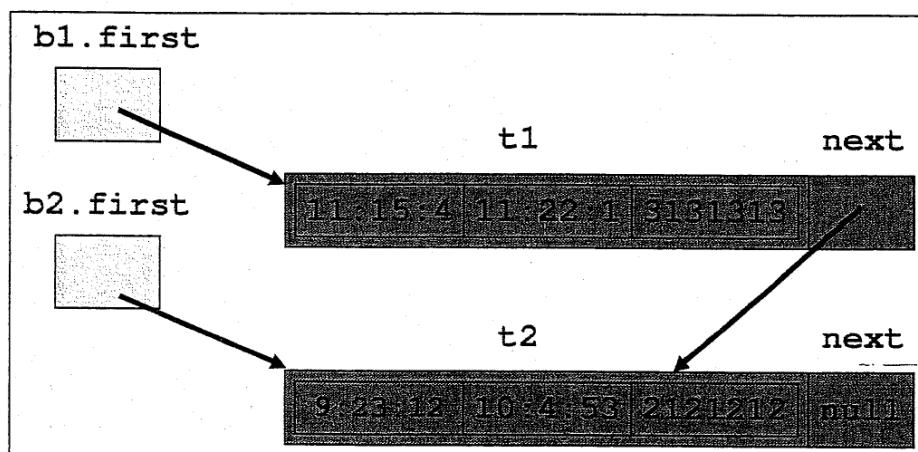
3.13 Condivisione controllata della memoria

Le ridefinizioni del costruttore di copia e dell'assegnazione per la classe `bolletta` evitano completamente la condivisione di memoria e quindi ogni possibile interferenza tra oggetti diversi. Tuttavia, effettuare copie profonde, seguite da corrispondenti distruzioni profonde può risultare inutilmente costoso in alcuni casi. Ad esempio, la copia profonda risulta inutile quando passiamo un parametro per valore ad una funzione che non lo modifica. Possiamo ovviare a questi sprechi adottando la cosiddetta *tecnica pigra* (lazy in inglese) che consiste nell'effettuare la copia profonda solamente quando la condivisione di memoria farà sorgere problemi di interferenza.

Consideriamo la seguente situazione di condivisione di memoria.

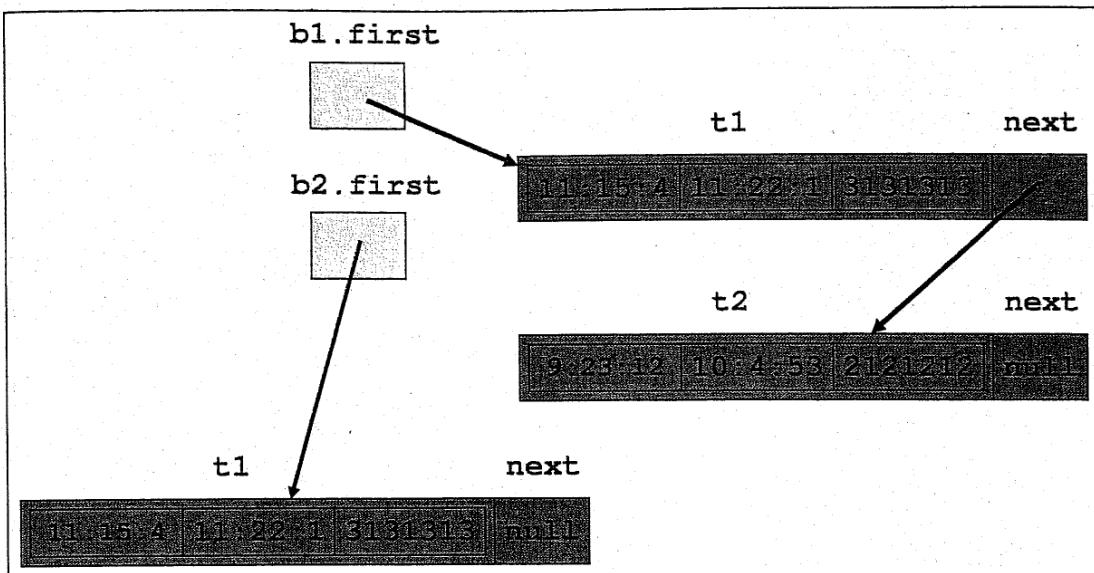


Dopo l'invocazione `b2.Togli_Telefonata(t1)`; vorremmo arrivare ad avere la seguente situazione.



Invece dopo l'invocazione `b2.Togli_Telefonata(t2)`; vorremmo ottenere la seguente situazione.

3.13 Condivisione controllata della memoria



Per ottenere ciò utilizziamo la cosiddetta tecnica del “*reference counting*”, cioè aggiungiamo alla classe interna nodo un campo dati riferimenti di tipo intero che conta il numero di puntatori che puntano ad un dato oggetto di nodo.

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

Dobbiamo ridefinire l'operatore `delete` della classe `nodo` in modo tale che esso si limiti a decrementare di una unità il campo `riferimenti` e soltanto quando tale campo `riferimenti` diventa 0 rilasci effettivamente la memoria utilizzata dal nodo. La dichiarazione dell'overloading dell'operatore `delete` da inserire internamente alla classe `nodo` ha segnatura

```
void operator delete(void*);
```

dove il parametro formale di tipo `void*` è il puntatore all'oggetto di classe `nodo` da deallo- care. Tale ridefinizione è implicitamente statica, ovvero non ha un oggetto di invocazione.

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

L'overloading dell'operatore `delete` per una classe `C` naturalmente non ha effetti sull'operatore `delete` di altri classi o di tipi non classe: per i tipi non classe ovviamente la `delete` rimane quella standard, mentre per una classe `D` viene invocata o la `delete` standard oppure quella ridefinita internamente a `D`. È comunque possibile invocare la `delete` standard per `C` usando l'operatore di risoluzione dello scope globale in questo modo:

```
::delete p;
```

È possibile ridefinire anche l'operatore `new`. Per i nostri scopi tuttavia non sarà necessario. La dichiarazione dell'overloading dell'operatore `new`, anche in questo caso implicitamente statico, internamente ad una classe `C` è la seguente:

```
class C {  
...  
public:  
    void* operator new(size_t s);  
...  
};
```

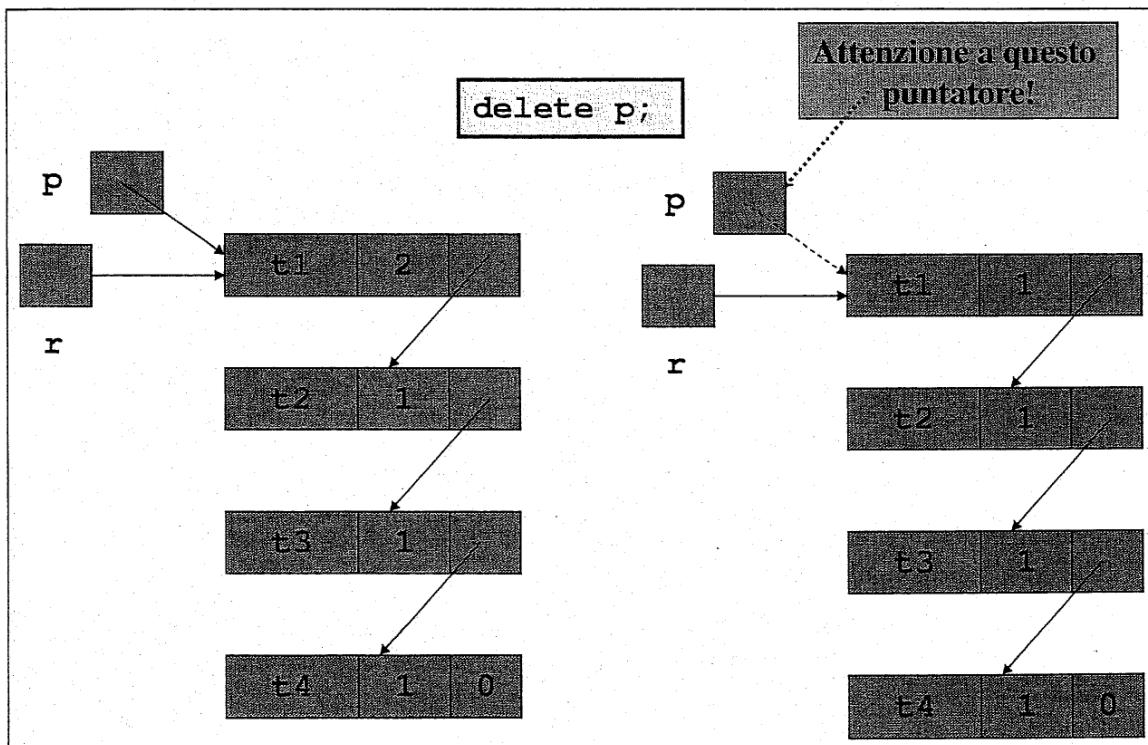
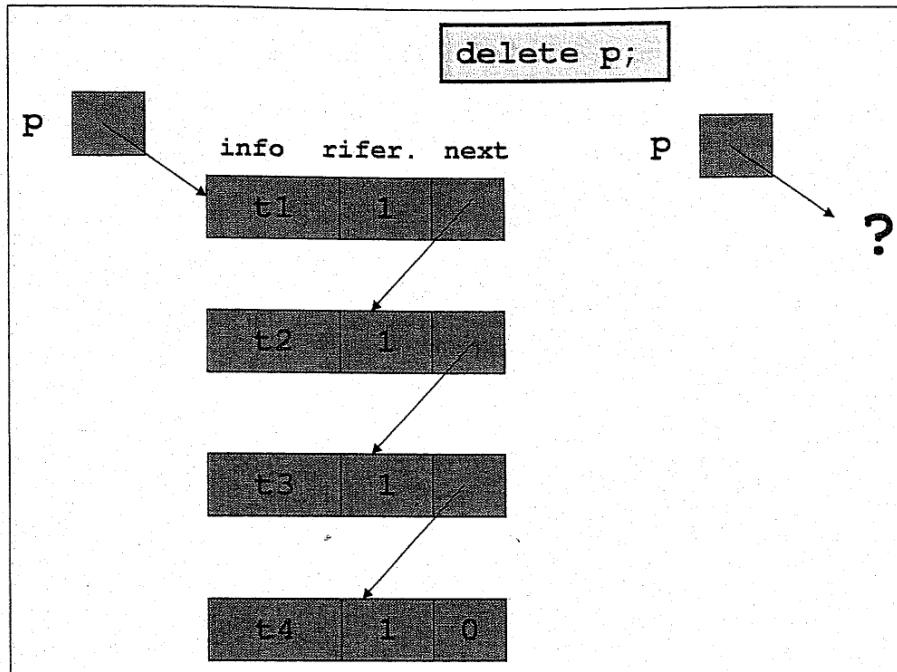
che ha quindi un parametro `s` di tipo `size_t`, dove `size_t` è un tipo definito tramite un `typedef` nel file header di sistema `<cstddef>` (la definizione effettiva di `size_t` dipende dall'architettura sottostante) e dove `s` è automaticamente inizializzato al numero di byte da allocare. Usualmente, la ridefinizione della `new` contiene una chiamata `malloc(s)` (si tratta di una funzione di basso livello di memory allocation usata in C ed ereditata in C++) dove `s` è il parametro di tipo `size_t`.

Torniamo alla ridefinizione di `delete` per la classe `nodo`.

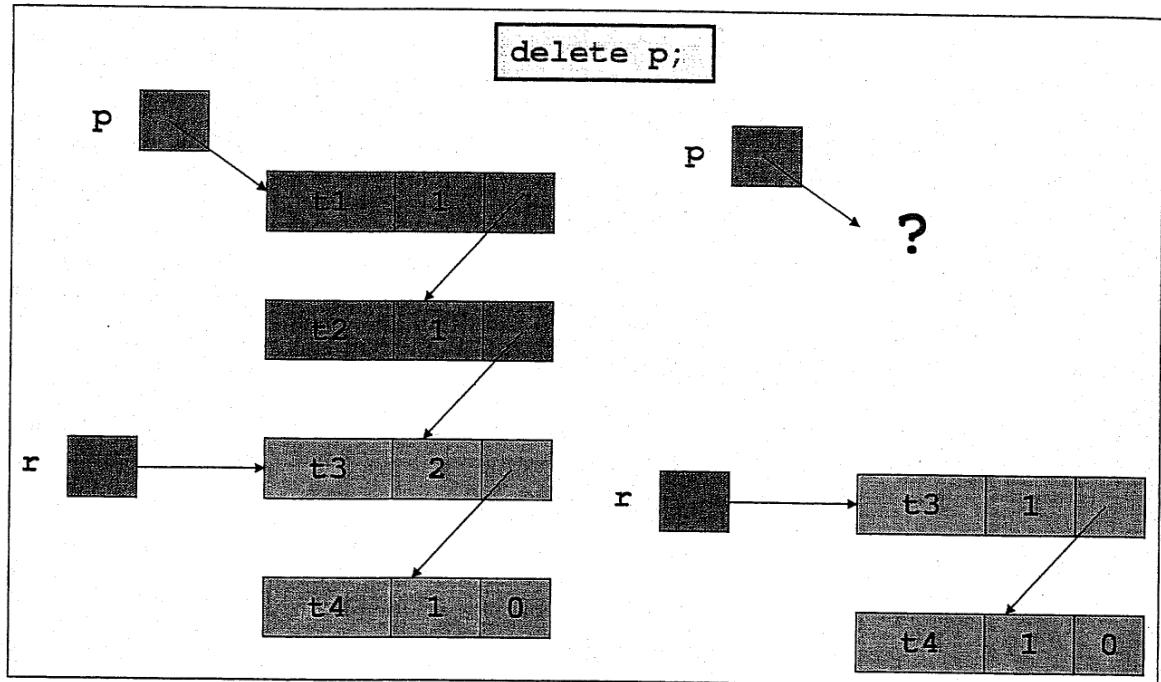
```
void bolletta::nodo::operator delete(void* p) {  
    if(p){ // se c'è qualcosa da deallocare  
        nodo* q = static_cast<nodo*> (p); // cast esplicito  
        // sappiamo che q->riferimenti > 0  
        q->riferimenti--;  
        if (q->riferimenti == 0) { // devo deallocare "effettivamente"  
            delete q->next; // invoco ricorsivamente la delete ridefinita  
            // su q->next poiché sto per eliminare un  
            // puntatore al nodo puntato da q->next  
            ::delete q; // delete standard su q  
        }  
    }  
}
```

Rappresentiamo graficamente la situazione per varie possibilità di `delete p;`

3.13 Condivisione controllata della memoria



3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI



La dichiarazione completa della classe nodo da inserire nella parte privata della classe bolletta è quindi la seguente.

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

Poiché la classe nodo è stata dichiarata nella parte privata della classe bolletta le uniche funzioni che possono accedere al campo **riferimenti** e gestire i puntatori a oggetti di tipo **nodo** sono i metodi della classe **bolletta**. Quindi la gestione del campo **riferimenti** è completamente sotto il controllo del progettista della classe **bolletta**. Esternamente alla classe **bolletta** non è nemmeno visibile la presenza del campo dati **riferimenti**. Naturalmente, la tecnica **lazy** impone l'opportuna gestione del campo dati **riferimenti** per

3.13 Condivisione controllata della memoria

tutti i metodi della classe `bolletta`. Tali modifiche ai metodi di `bolletta` non avranno alcuna conseguenza sui programmi degli utenti della classe. Si tratta di una modifica alla rappresentazione interna della classe `bolletta` e quindi come tale non visibile all'utente esterno. Vediamo quindi la definizione completa della classe `bolletta` con la gestione pigrigra della condivisione di memoria. Il principale problema per una implementazione corretta della memoria condivisa è mantenere aggiornato il campo dati `riferimenti` degli oggetti di tipo `nodo`. Dimenticare di incrementare `x.riferimenti` quando si assegna l'indirizzo di `x` ad un puntatore `p` di tipo `nodo*` o dimenticare di decrementare `x.riferimenti` quando si distrugge un puntatore `p` che puntava ad `x` oppure quando gli si assegna un nuovo valore può portare a situazioni inconsistenti con conseguenze potenzialmente disastrose.

La parte pubblica della dichiarazione contenuta nel file header `bolletta.h` rimane invariata. La parte privata di `bolletta` contiene la nuova versione della dichiarazione della classe interna `nodo` ed il puntatore `first` alla testa della lista. Non sono invece più necessari i metodi statici privati `copia` e `distruggi` che servivano per effettuare copie e distruzioni profonde.

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

class bolletta {
public:
    bolletta(): first(0) {}
    bolletta(const bolletta&);
    ~bolletta();
    bool Vuota() const;
    void Aggiungi_Telefonata(const telefonata&);
    void Togli_Telefonata(const telefonata&);
    telefonata Estrai_Una();
    bolletta& operator=(const bolletta&);
private:
    class nodo {
        public:
            telefonata info;
            nodo* next;
            int riferimenti;
            nodo();
            nodo(const telefonata&, nodo*);
            void operator delete(void*);
    };
    nodo* first;
};

#endif
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

La definizione dei costruttori della classe nodo diventa la seguente.

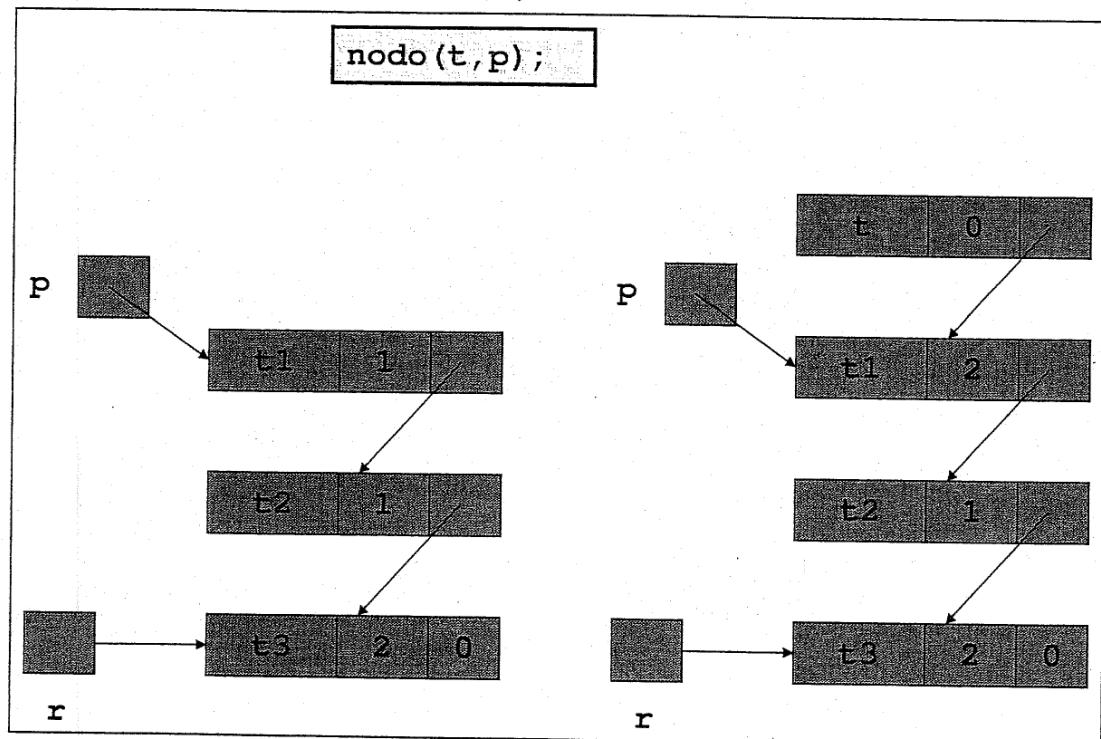
```
// file "bolletta.cpp"
#include <iostream>
#include "bolletta.h"

bolletta::nodo::nodo(): // per il campo dati info costr. standard
    next(0), riferimenti(0) {}

// nessun puntatore al nodo quindi riferimenti(0)

bolletta::nodo::nodo(const telefonata& t, nodo* p)
    : info(t), next(p), riferimenti(0) {
    // p var. locale che punta al nodo puntato dal parametro attuale
    // quindi dovrei fare: if (p) p->riferimenti++;
    if (next) next->riferimenti++; // aggiornamento
    // p var. locale cessa di vivere
    // quindi dovrei fare: if (p) p->riferimenti--;
}
```

Graficamente:



Occorre inoltre aggiungere la ridefinizione di `delete` per `nodo` che già abbiamo visto.

```
void bolletta::nodo::operator delete(void* p) {
    if(p) {
```

3.13 Condivisione controllata della memoria

```
nodo* q = static_cast<nodo*> (p);
// sappiamo che q->riferimenti è > 0
q->riferimenti--;
if(q->riferimenti == 0) {
    delete q->next; // chiamata ricorsiva della
                      // delete ridefinita su q->next
    ::delete q; // delete standard su q
}
}
```

Con ciò abbiamo completato la definizione della classe interna nodo, cioè dei “metodi privati” di bolletta. Vediamo ora la definizione dei metodi pubblici.

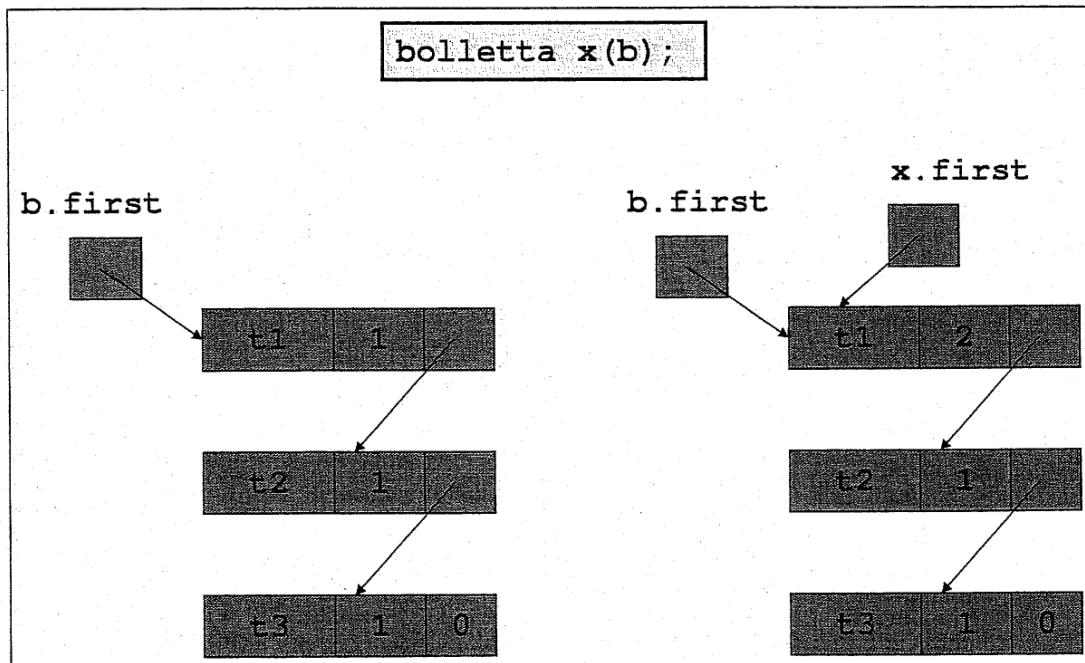
Il costruttore di copia di bolletta ora provoca condivisione controllata di memoria in quanto si limita a copiare il puntatore first aggiornando il campo riferimenti dell’oggetto puntato. Il codice diventa semplicemente il seguente.

```
bolletta::bolletta(const bolletta& b): first(b.first) {
    if (first) first->riferimenti++;
}
```

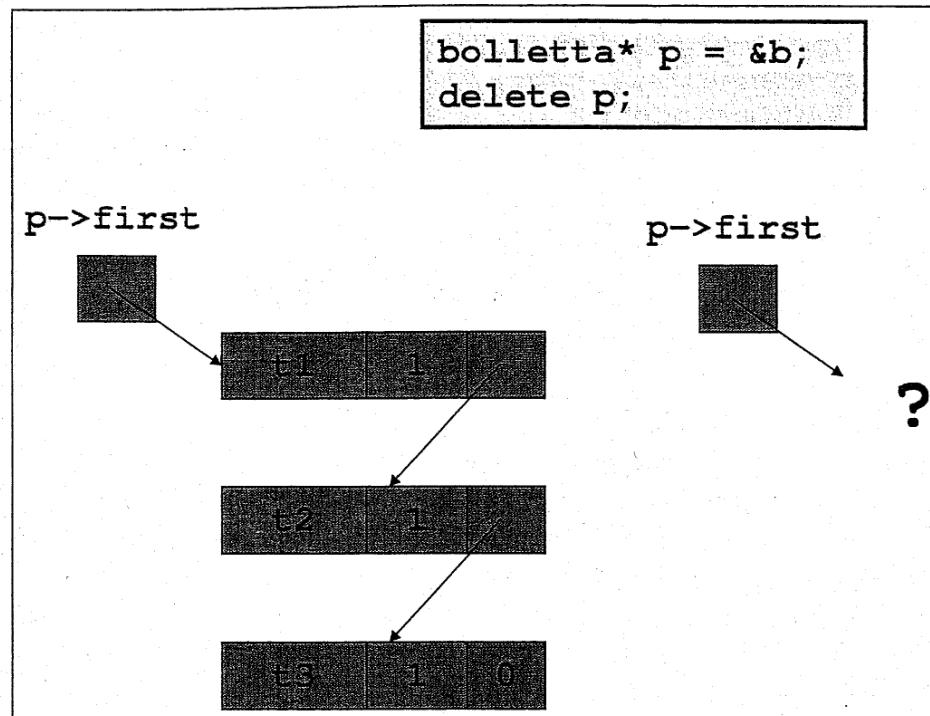
Il distruttore semplicemente richama la versione ridefinita di delete sul puntatore first.

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

Analizziamo il comportamento graficamente.



3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI



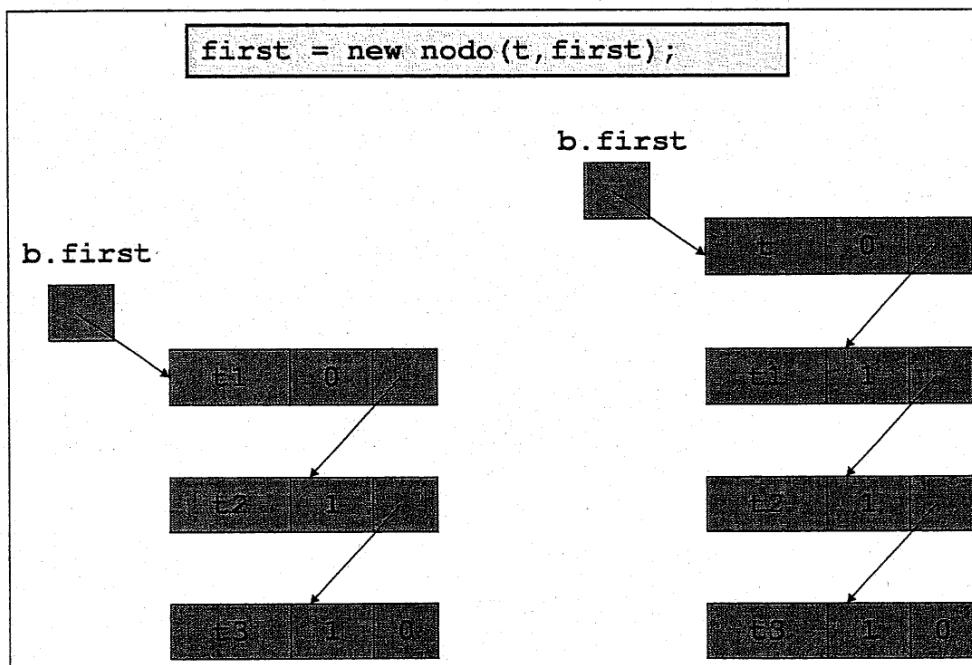
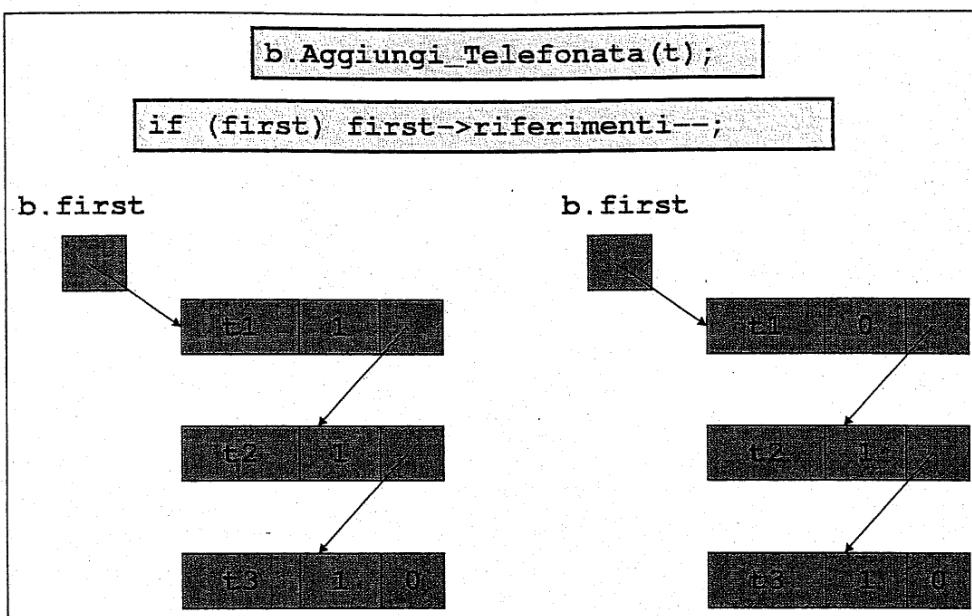
La definizione del test Vuota () non cambia:

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

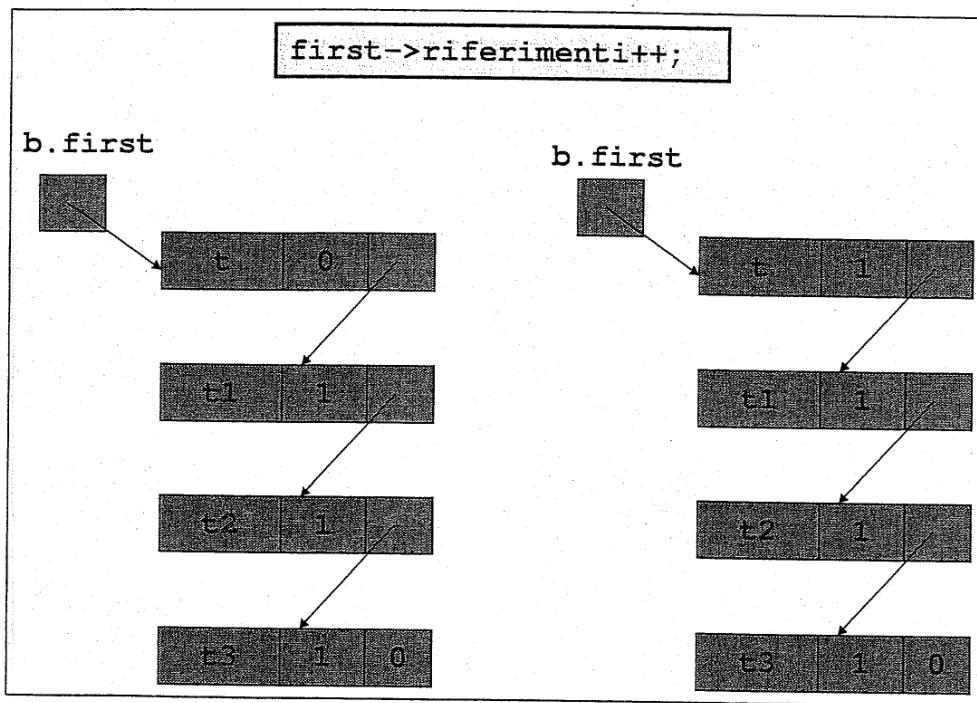
Cambia invece il metodo Aggiungi_Telefonata.

```
void bolletta::Aggiungi_Telefonata(const telefonata& t){  
    if (first) first->riferimenti--;  
    // NOTARE MOLTO BENE:  
    // a questo punto vi è una inconsistenza temporanea  
    // è necessario il first->riferimenti-- perché la new  
    // seguente, che invoca il costruttore di nodo, comporta  
    // l'incremento del campo first->riferimenti  
    first = new nodo(t,first);  
    first->riferimenti++;  
}
```

3.13 Condivisione controllata della memoria



3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI



Si noti attentamente che senza `if(first)` `first->riferimenti--;`, avremmo ottenuto la seguente situazione inconsistente.

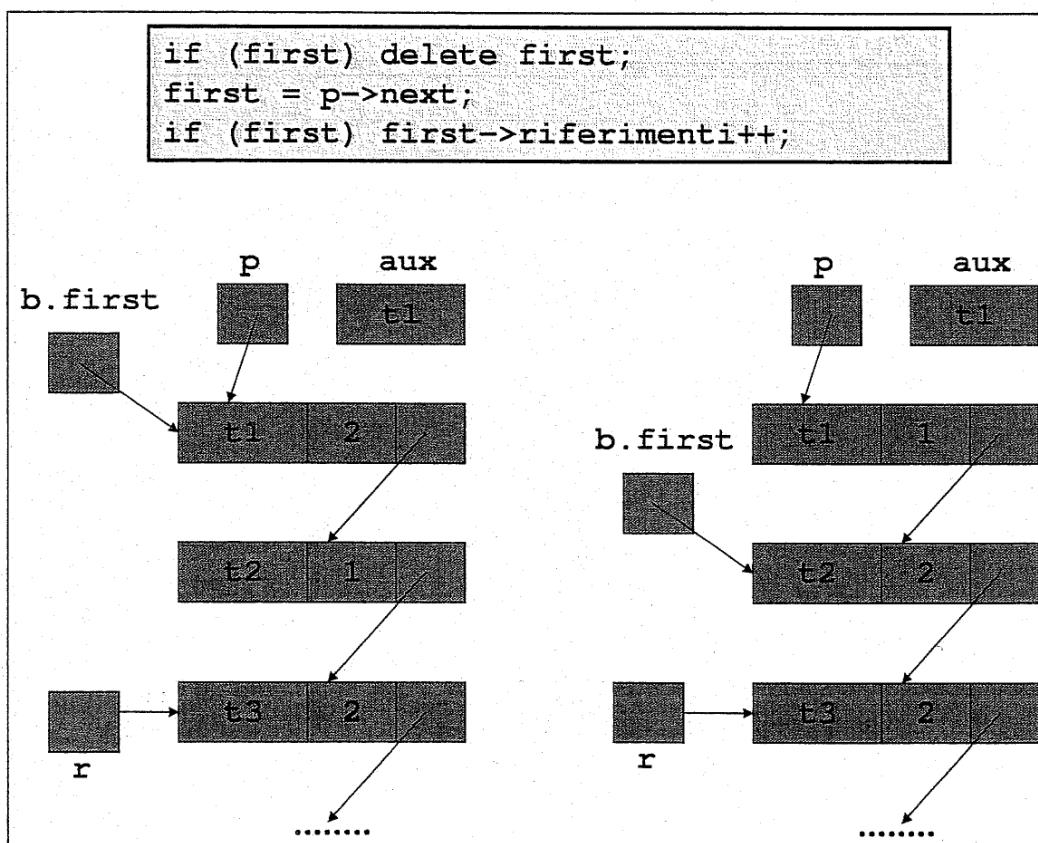
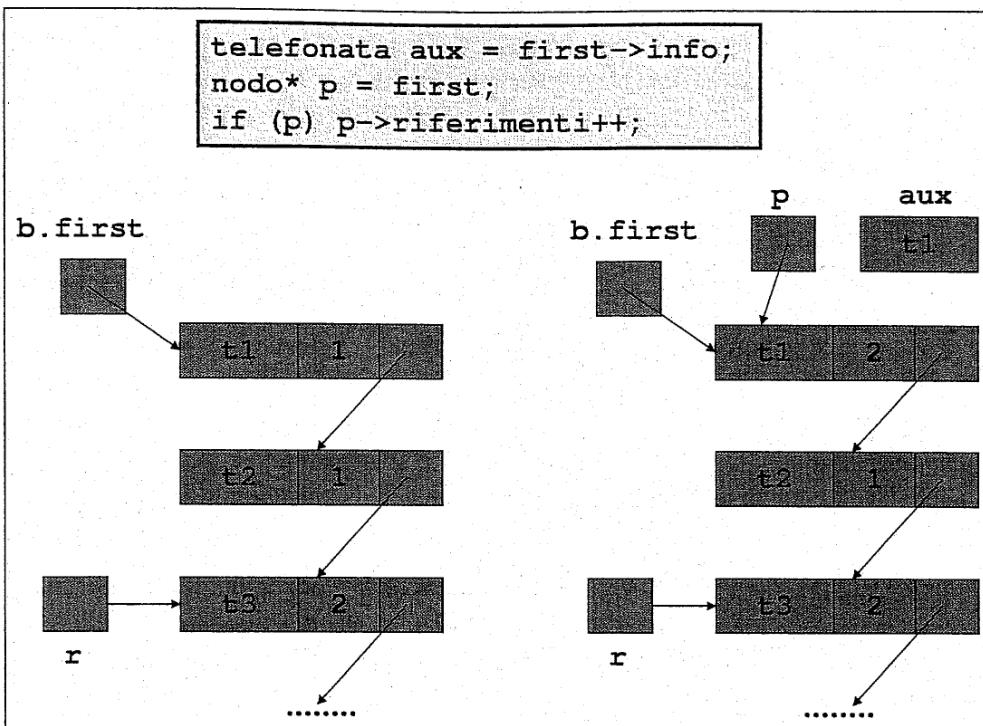


Un po più delicata è la modifica del metodo `Estrai_Una()`.

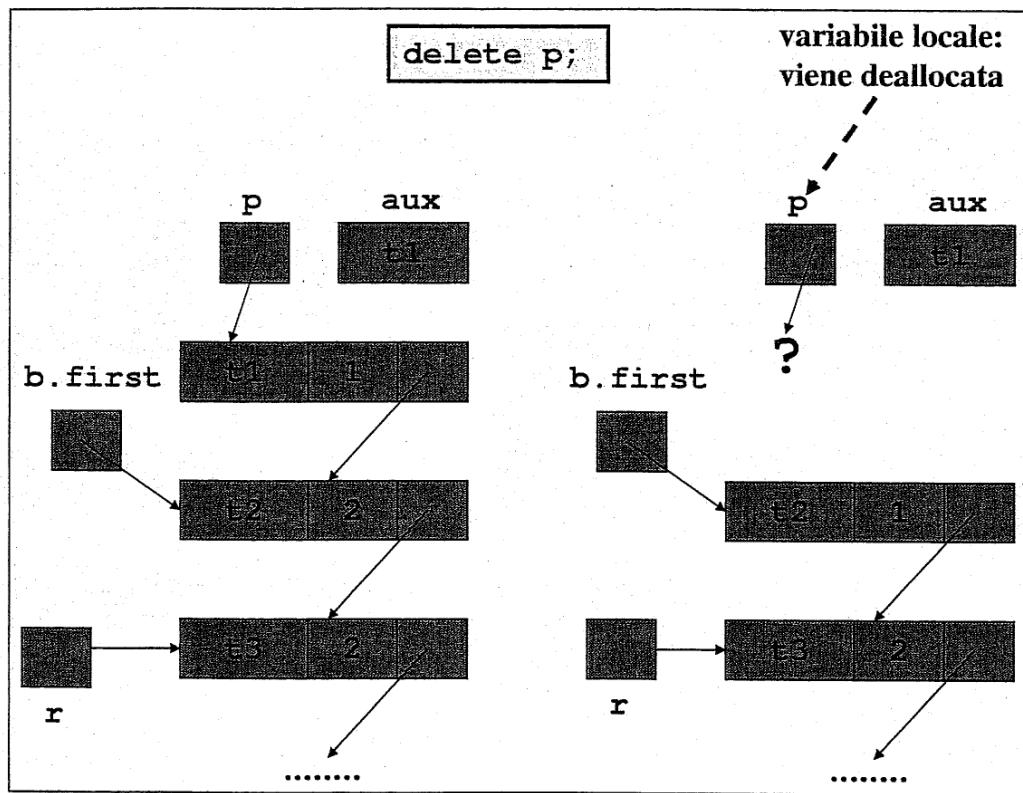
```
telefonata bolletta::Estrai_Una() {
    // Precondizione: bolletta di invocazione non vuota, first != 0
    telefonata aux = first->info;
    nodo* p = first;
    // p->riferimenti++;
    // first->riferimenti--;
    first = first->next;
    if (first) first->riferimenti++;
    delete p;
    return aux;
}
```

Analizziamo graficamente ciò che succede invocando `t=b.Estrai_Una();`.

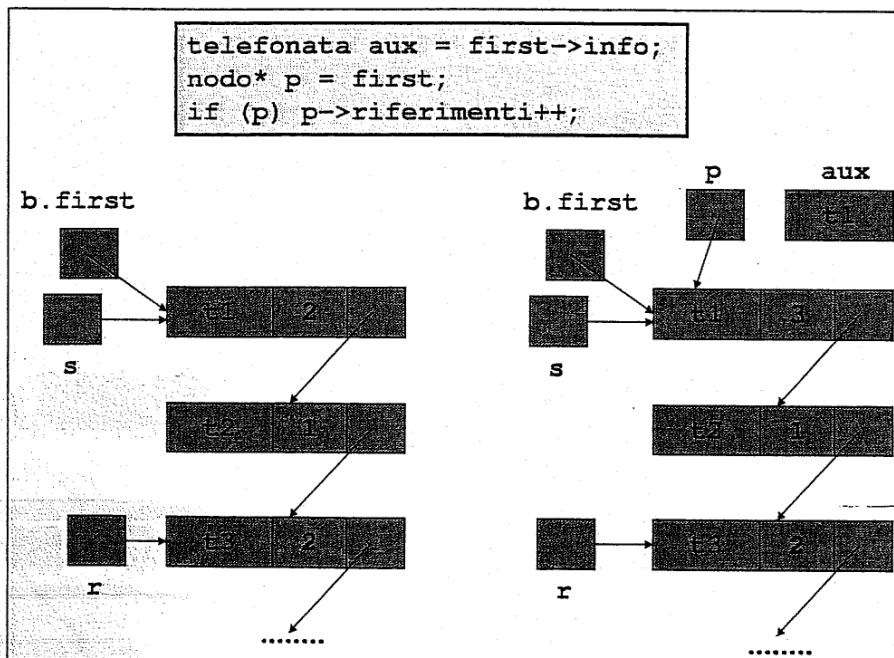
3.13 Condivisione controllata della memoria



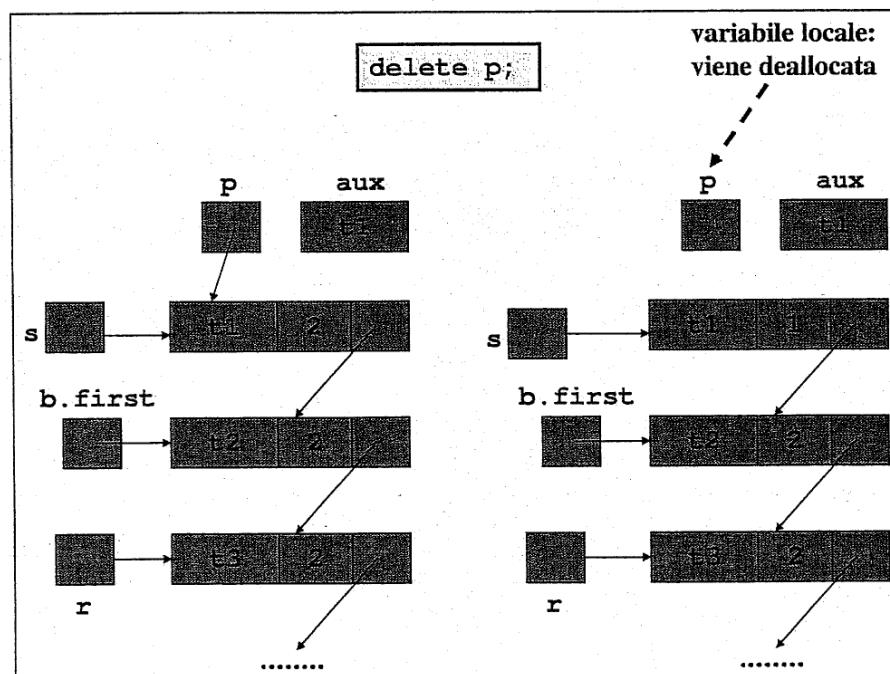
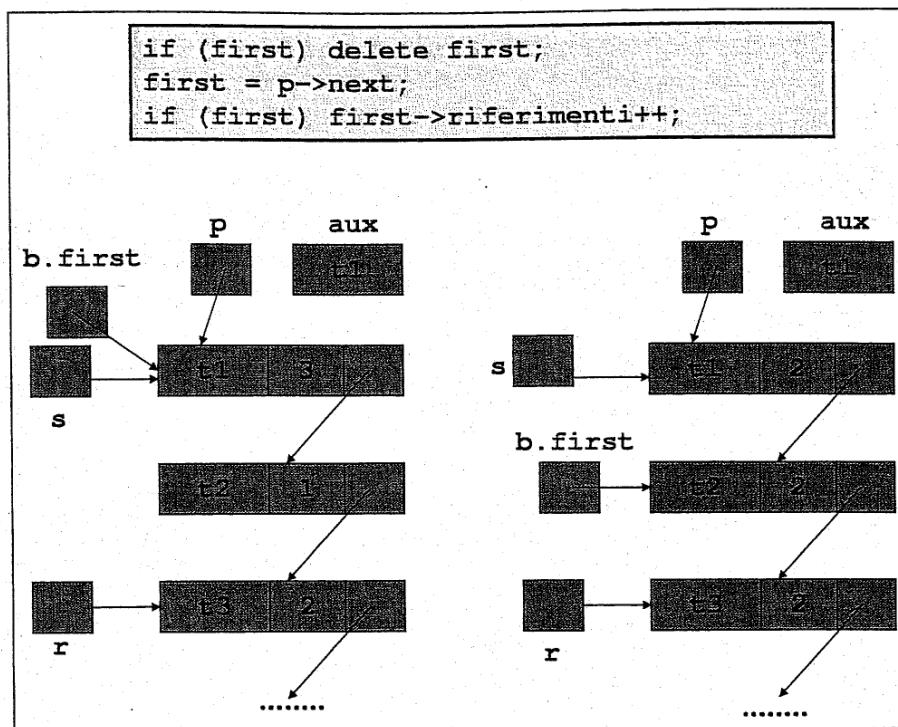
3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI



Consideriamo ora la seguente diversa situazione di invocazione di `t=b.Estrai_Una();`.



3.13 Condivisione controllata della memoria



Ricordiamo la precedente definizione di `Togli_Telefonata`.

```

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

```

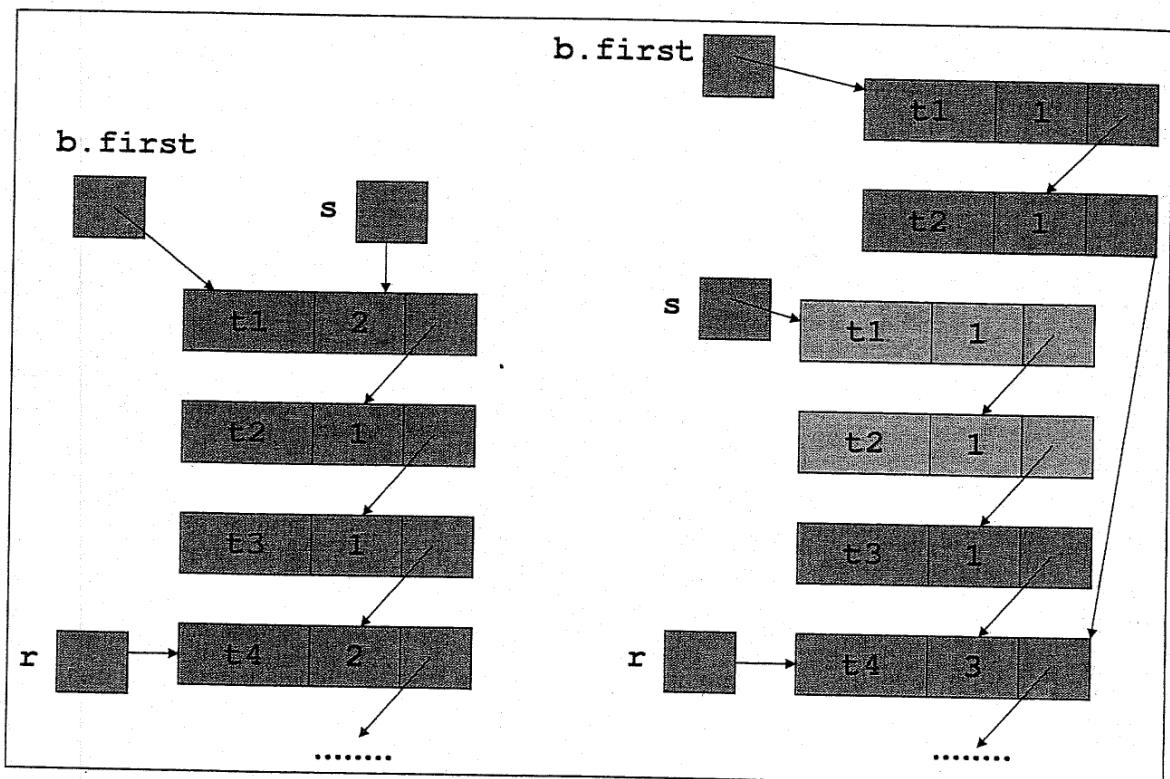
3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```

    p = p->next;
}
if (p) { // ho trovato t
    if (!prec) // t è in testa alla lista
        first = p->next;
    else // t non è in testa e prec punta al nodo precedente
        prec->next = p->next;
    delete p;
}
}

```

Risulta parecchio delicata la modifica di `Togli_Telefonata()`. L'idea è quella di fare una copia profonda parziale della lista fino al nodo contenente la telefonata da togliere. Costruiremo la copia scorrendo la lista. Quindi la copia risulterà inutile qualora non si trovasse la telefonata da togliere. Si consideri ad esempio la seguente situazione per l'invocazione `b.Togli_Telefonata(t3);`



La definizione completa del metodo è la seguente.

```

void bolletta::Togli_Telefonata(const telefonata& t) {
    nodo *p = first, *prec = 0, *q = 0;
    // if (p) p->riferimenti++;
    // if (first) delete first;
    first = 0;
}

```

3.13 Condivisione controllata della memoria

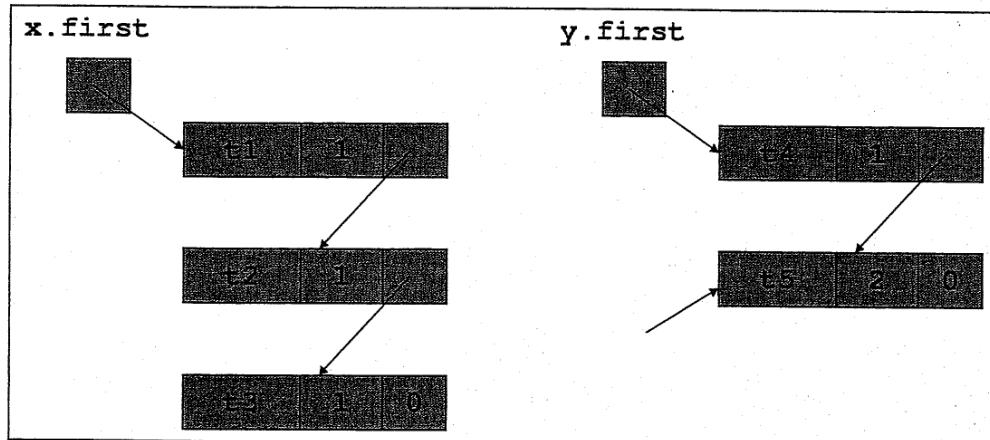
```
while (p && !(p->info == t)) {
    // fa una copia degli elementi che precedono quello da togliere
    if (q) delete q;
    q = new nodo(p->info, p->next);
    q->riferimenti++;
    // e li inserisce nella nuova lista sempre puntata da first
    if (prec == 0) {
        // if (first) delete first;
        first = q;
        first->riferimenti++;
    }
    else {
        if (prec->next) delete prec->next;
        prec->next = q;
        prec->next->riferimenti++;
    }
    // aggiorna prec e p per il ciclo while
    if (prec) delete prec;
    prec = q;
    prec->riferimenti++;
    if (p) delete p;
    p = q->next;
    if (p) p->riferimenti++;
}
} // fine ciclo while
// finito il ciclo while, se p == 0 la telefonata
// non è stata trovata
if (p)
    // tolgo il nodo puntato da p
    if (prec == 0) { // t era in testa
        // if (first != 0) delete first;
        first = p->next;
        if (first) first->riferimenti++;
    }
    else {
        if (prec->next) delete prec->next;
        prec->next = p->next;
        if (prec->next)
            prec->next->riferimenti++;
    }
// p, prec e q sono puntatori locali
if (p) delete p;
if (prec) delete prec;
if (q) delete q;
}
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

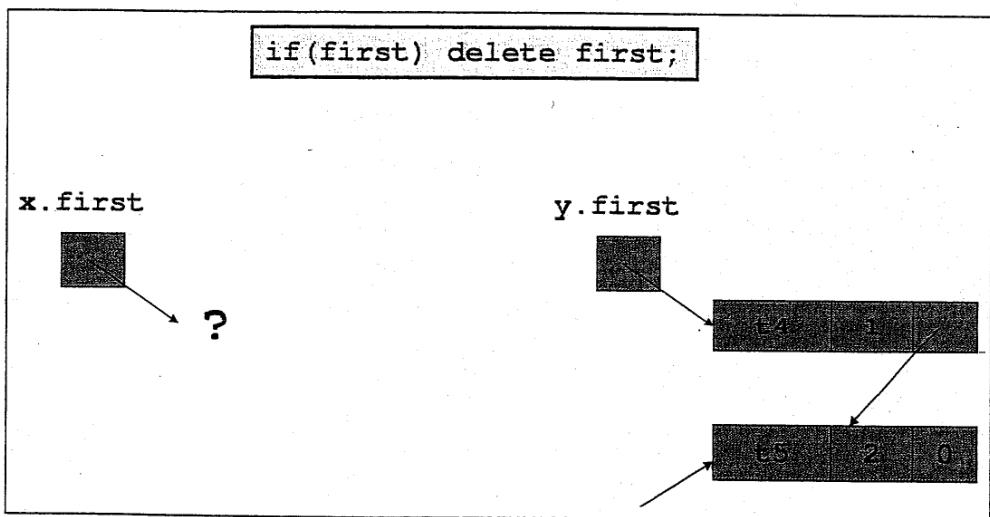
Anche la ridefinizione dell'operatore di assegnazione richiede delle modifiche:

```
bolletta& bolletta::operator=(const bolletta& b) {
    if (this != &b) {
        if (first) delete first;
        first = b.first;
        if (first) first->riferimenti++;
    }
    return *this;
}
```

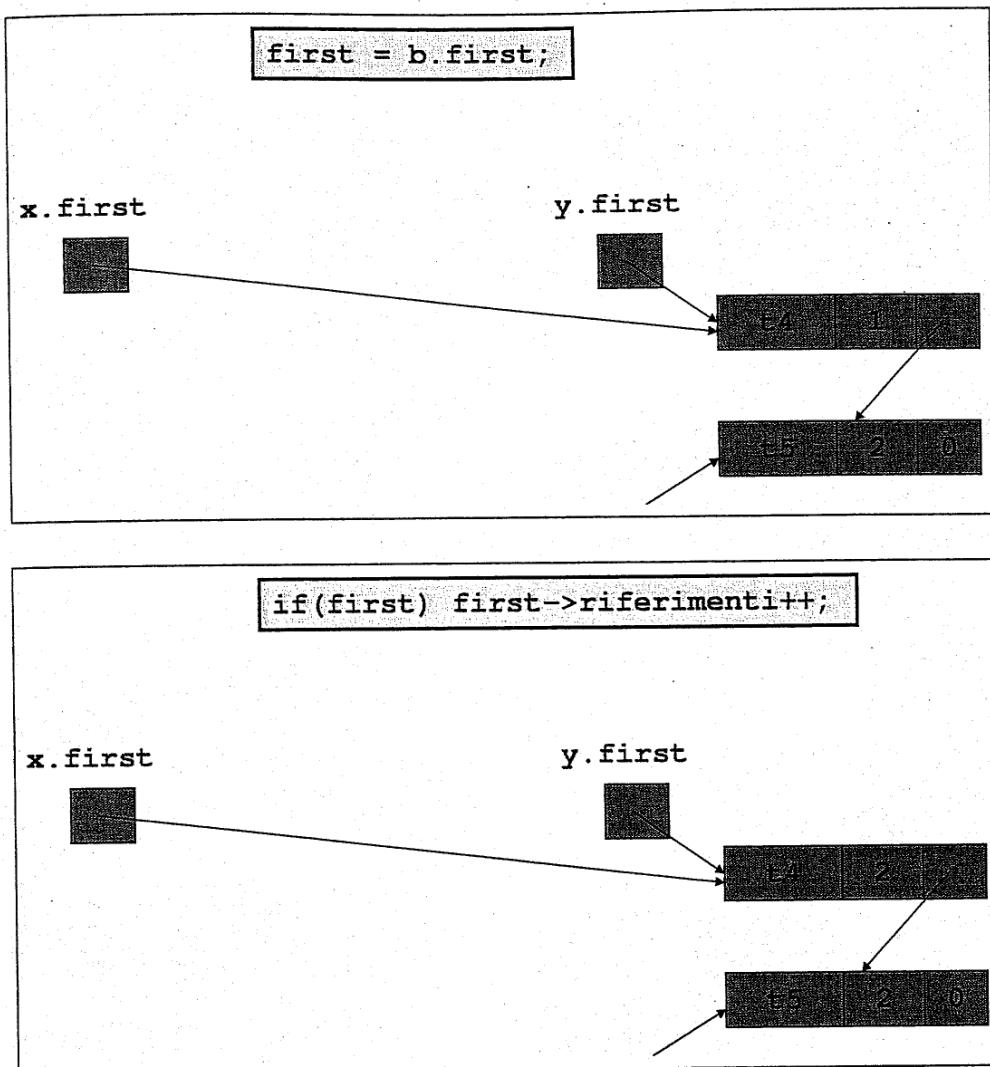
Ad esempio consideriamo la situazione iniziale rappresentata nella seguente figura.



In questa situazione vediamo graficamente ciò che succede eseguendo l'assegnazione `x=y;`.



3.13 Condivisione controllata della memoria



3.13.1 Puntatori smart

Come abbiamo notato, la difficoltà principale nella tecnica del reference counting risiede nel mantenere aggiornato il campo `riferimenti` degli oggetti di tipo `nodo`. Quando si assegna ad un puntatore `p` di tipo `nodo*` l'indirizzo di un oggetto `x` di tipo `nodo` non sempre è possibile inserire l'istruzione che aggiorna `x.riferimenti` nel punto esatto in cui si assegna l'indirizzo di `x` a `p`. Analogamente quando si distrugge un puntatore `p` che puntava ad `x` oppure si assegna un nuovo valore al puntatore `p` che puntava ad `x`. Di conseguenza ci saranno alcune sezioni "critiche" del programma in cui la condizione che ci siano esattamente `x.riferimenti` puntatori che puntano ad `x` non è soddisfatta.

Per rendere automatica la gestione di tale campo `riferimenti` basterebbe poter ridefinire assegnazione, costruttore di copia e distruttore per i puntatori a `nodo`. Purtroppo questo non è possibile in quanto i puntatori sono tipi derivati e non oggetti di una classe definita dal programmatore. La soluzione è definire una classe da usare quale *puntatore*

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

smart, cioè "furbo/intelligente" (in inglese "smart pointer"), a nodo che contiene un unico campo dati di tipo *nodo** ed in cui ridefinisco assegnazione, costruttore di copia e distruttore che aggiornano opportunamente il campo riferimenti di nodo. Questa idea porta alla seguente nuova versione di bolletta.

```
// file "bolletta.h"
#ifndef BOLLETTA_H
#define BOLLETTA_H
#include <iostream>
#include "telefonata.h"
using std::ostream;

class bolletta {
public:
    // la parte pubblica non cambia
    bool Vuota() const;
    void Aggiungi_Telefonata(const telefonata&);
    void Togli_Telefonata(const telefonata&);
    telefonata Estrai_Una();
    friend ostream& operator<<(ostream&, const bolletta&);
private:
    // la parte privata contiene la definizione della classe nodo
    // e di una classe smartp puntatore "furbo" a nodo.
    class nodo; // dichiarazione incompleta di nodo
    class smartp {
public:
        nodo* punt;           // unico campo dati di smartp
        smartp(nodo* p = 0);   // COSTRUTTORE
        // NB1: agisce da convertitore implicito da nodo* a smartp
        // NB2: funziona anche da costruttore senza argomenti
        smartp(const smartp&);           // COSTRUTTORE DI COPIA
        ~smartp();                  // DISTRUTTORE
        smartp& operator=(const smartp&); // ASSEGNAZIONE
        nodo& operator*() const;        // DEREFERENZIAZIONE
        nodo* operator->() const;      // ACCESSO A MEMBRO
        bool operator==(const smartp&) const; // UGUAGLIANZA
        bool operator!=(const smartp&) const; // DISUGUAGLIANZA
    }; // fine classe smartp

    class nodo { // dichiarazione completa di nodo
public:
        nodo();
        nodo(const telefonata&, const smartp&);
        telefonata info;
        smartp next; // smart pointer al nodo successivo
        int riferimenti;
```

3.13 Condivisione controllata della memoria

```
// NB: non ridefinisco la delete di nodo
}; // fine classe nodo

smartp first; // unico campo dati di bolletta
};

#endif
```

Overloading dell'operatore di accesso a membro. L'operatore “->” di accesso a membro tramite puntatore può essere ridefinito internamente come operatore unario postfisso. Tipicamente ritorna un puntatore ad una classe oppure un oggetto di una classe per cui è stato ridefinito ->. Si consideri il seguente semplice esempio:

```
class C {
public:
    int x;
    C(int k=0): x(k) {}
};

class SmartPointer {
private:
    C* p;
public:
    SmartPointer(C* q = 0): p(q) {}
    C* operator->() const { return p; }
};

int main() {
    C a(5); SmartPointer ptr(&a);
    cout << ptr->x; // stampa 5
}
```

Passiamo ora ad esaminare la definizione delle funzioni della classe `bolletta` e delle classi interne a `bolletta`.

```
// file "bolletta.cpp"
#include "bolletta.h"
using std::ostream; using std::endl;

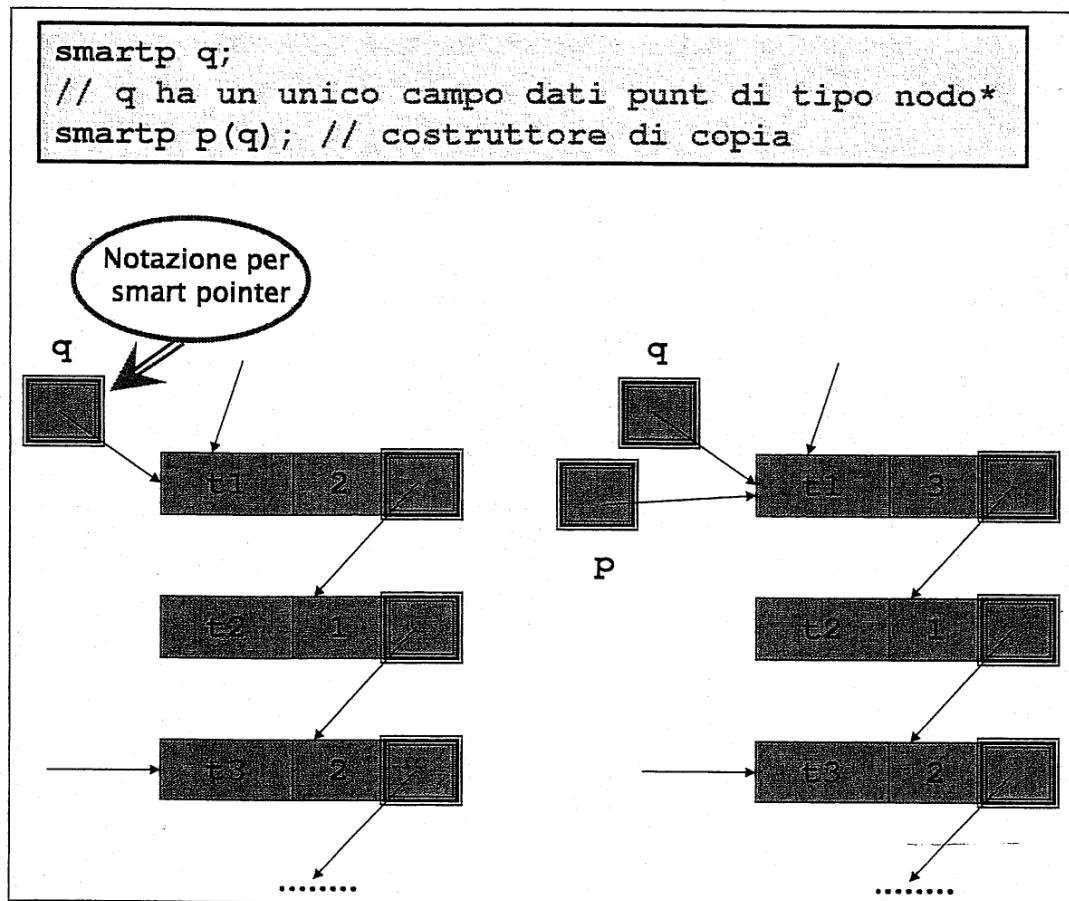
// METODI DI smartp

// COSTRUTTORE
bolletta::smartp::smartp(nodo* p): punt(p)
{ if (punt) punt->riferimenti++; }
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
// COSTRUTTORE DI COPIA  
bolletta::smartp::smartp(const smartp& s): punt(s.punt)  
{ if (punt) punt->riferimenti++; }  
  
// DISTRUTTORE  
bolletta::smartp::~smartp()  
{ if (punt) {  
    punt->riferimenti--;  
    if (punt->riferimenti == 0)  
        delete punt;  
    // NB: è la delete standard di nodo che richiama  
    // (ricorsivamente) il distruttore di smartp  
}
```

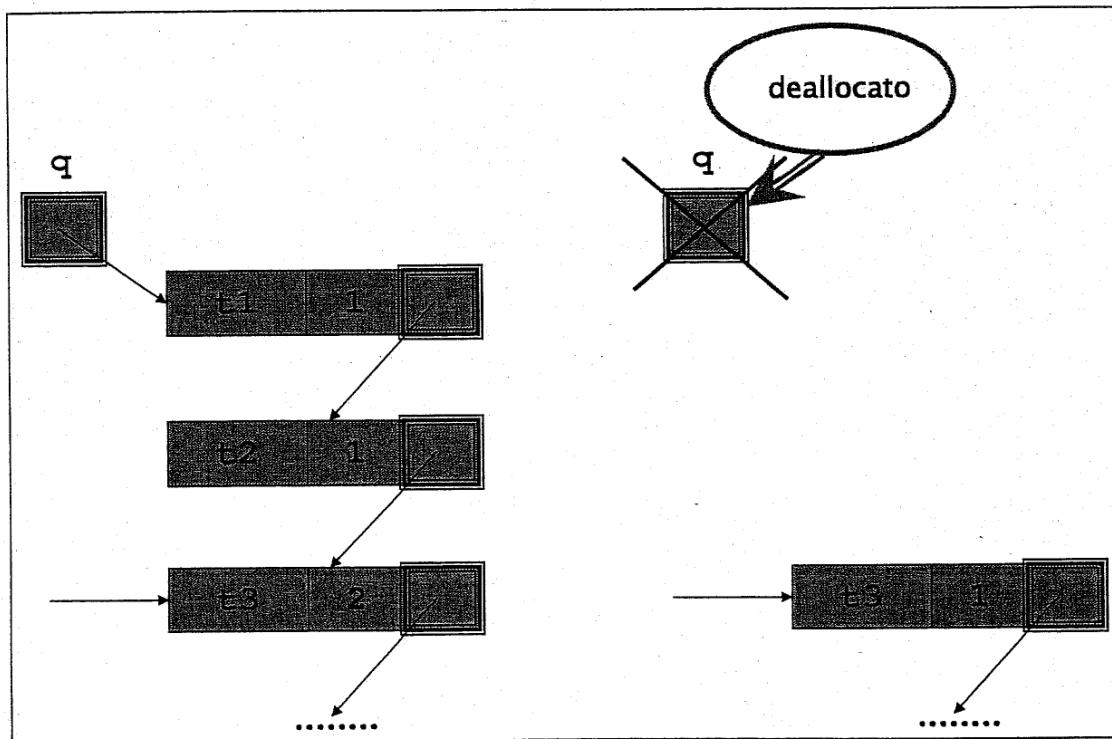
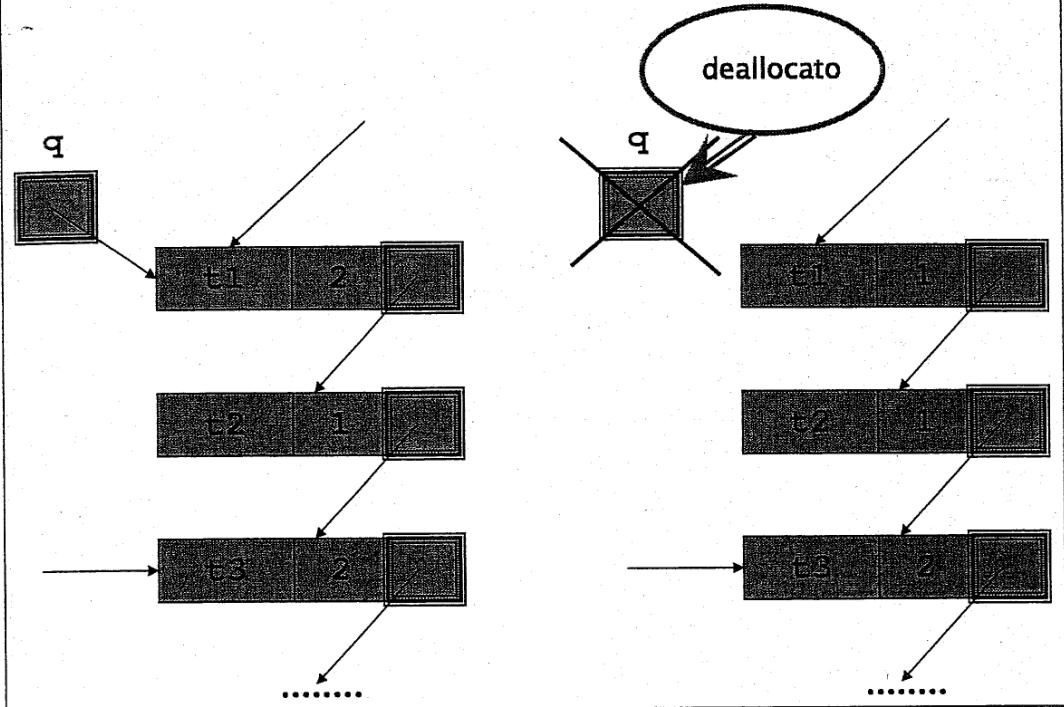
Analizziamo graficamente il comportamento del costruttore di copia di smartp.



Per il distruttore di smartp la situazione si può rappresentare graficamente nel seguente modo.

3.13 Condivisione controllata della memoria

```
// distruttore  
smartp q;  
smartp* pp=&q;  
delete pp; // ovvero: "~smartp(q);"
```

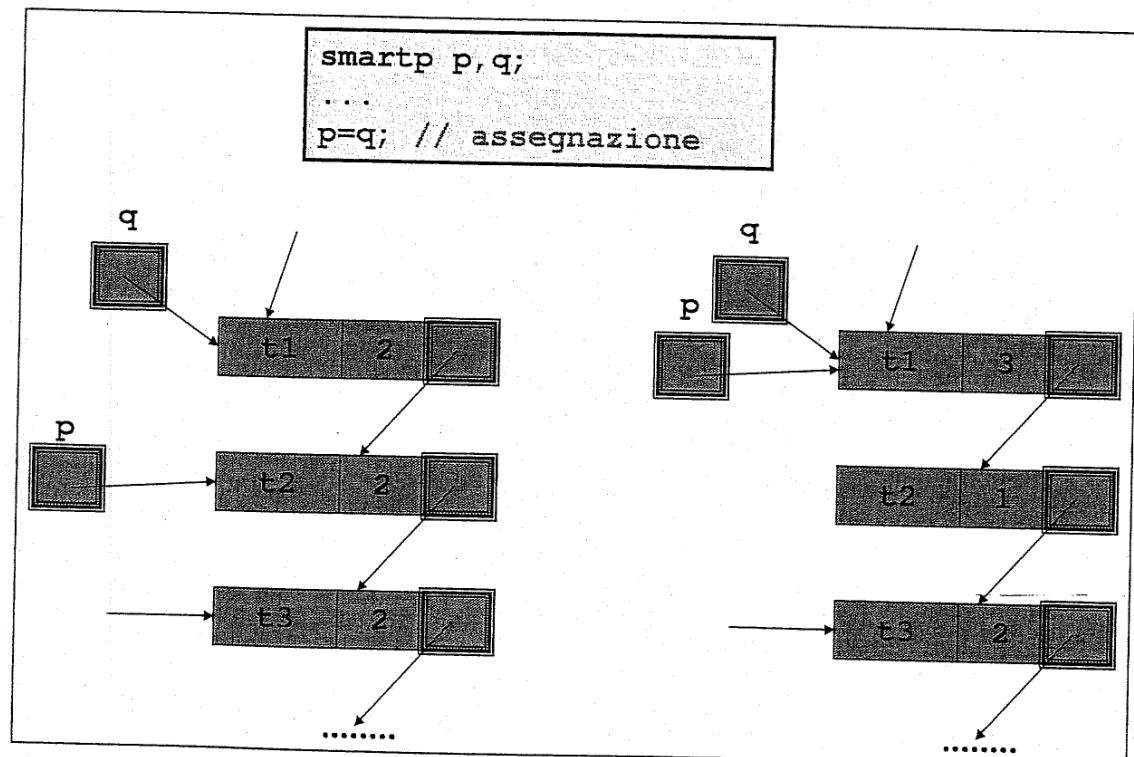


3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

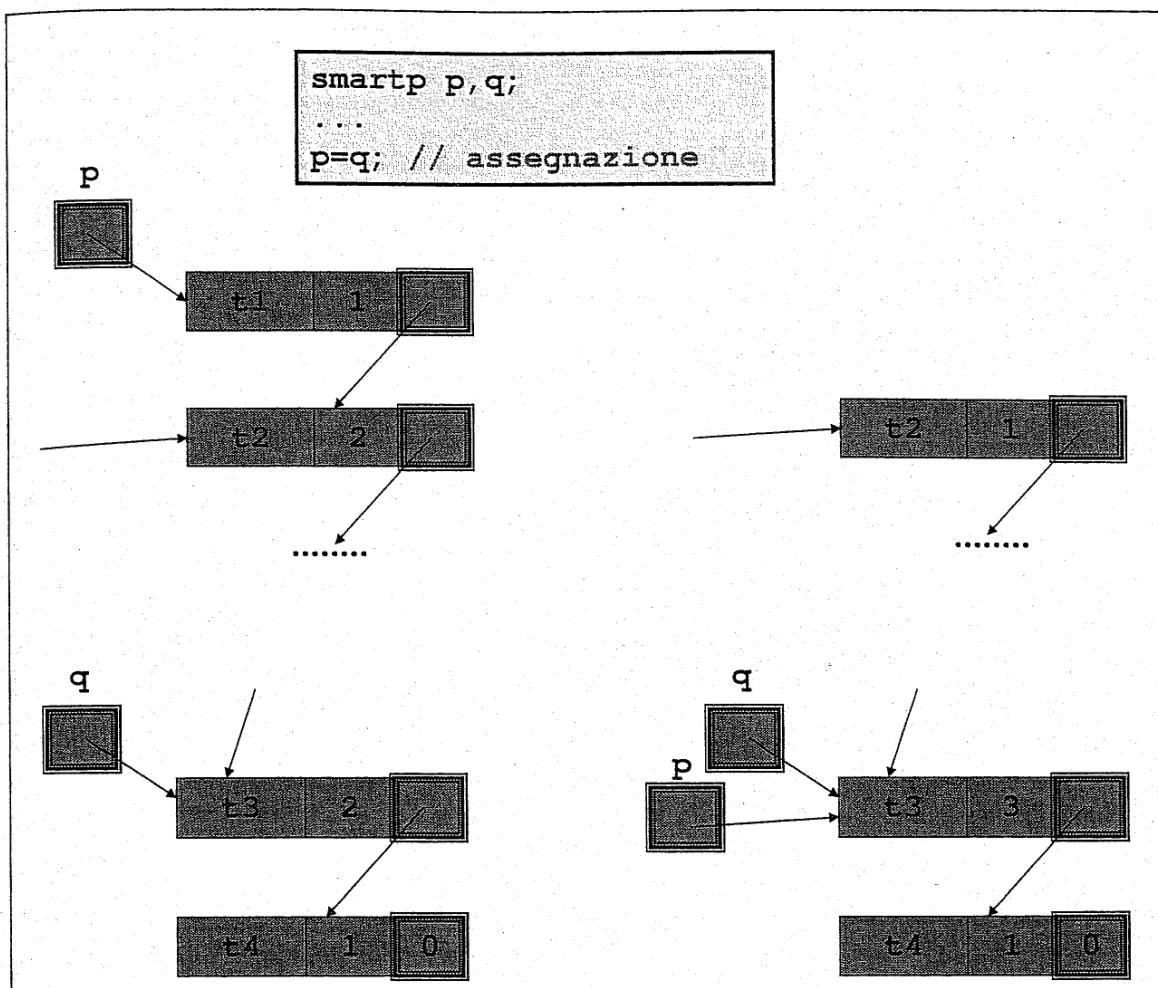
L'assegnazione tra puntatori smart sarà così definita.

```
// ASSEGNAZIONE
bolletta::smartp& bolletta::smartp::operator=(const smartp& s) {
    if (this != &s) {
        nodo* t = punt;
        // if (t) t->riferimenti++;
        // if (punt) punt->riferimenti--;
        punt = s.punt;
        if (punt) punt->riferimenti++;
        if (t) {
            t->riferimenti--;
            if (t->riferimenti == 0) delete t;
            // delete standard di nodo che quindi richiama
            // il distruttore ridefinito di smartp
        }
    }
    return *this;
}
```

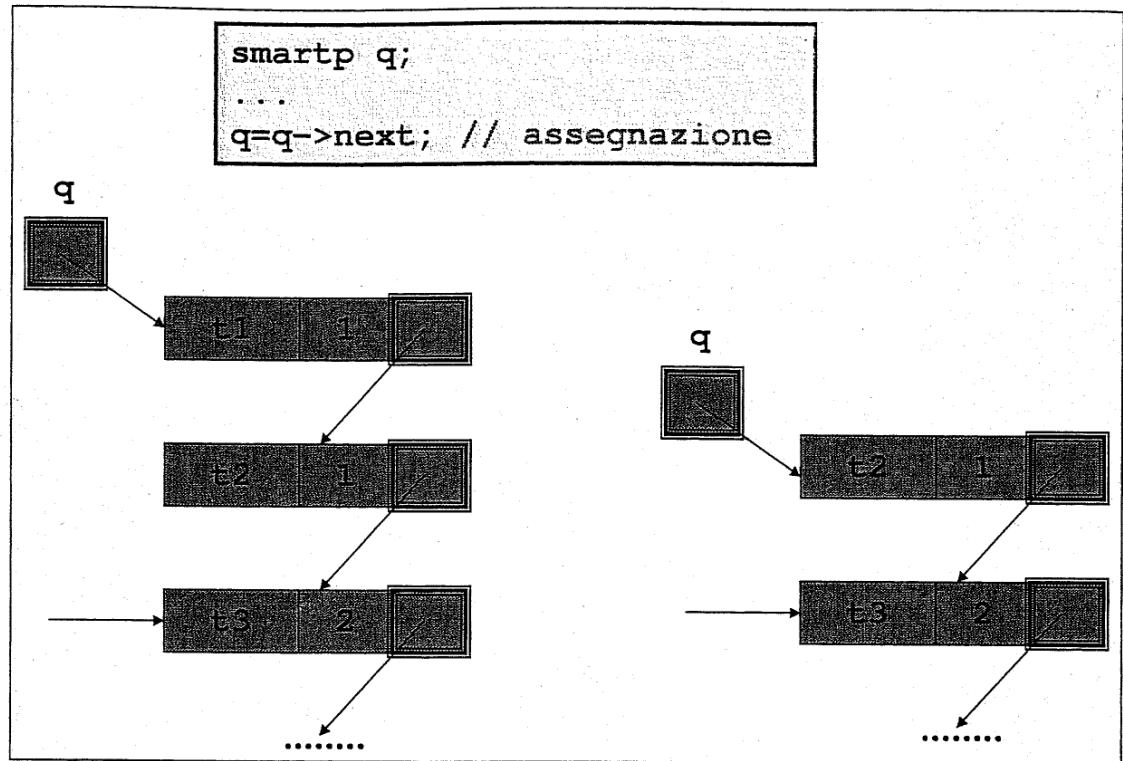
Vediamo graficamente i seguenti esempi di assegnazione $p=q$; tra puntatori smart.



3.13 Condivisione controllata della memoria



Consideriamo anche il successivo esempio di assegnazione `q=q->next`.



Vediamo ora la definizione dell'overloading degli operatori di dereferenziazione `*`, di selezione di membro `->`, di uguaglianza e disuguaglianza.

```

// OPERATORE DI DEREFERENZIAZIONE
// "trasforma" un oggetto smartp s nello l-valore di nodo puntato
// da s.punt. Posso fare: *s.info *s.next *s.riferimenti
bolletta::nodo& bolletta::smartp::operator*() const {
    return *punt;
}

// OPERATORE DI SELEZIONE DI MEMBRO
// "trasforma" un oggetto s di smartp nel puntatore ordinario a
// nodo contenuto in s.
// Posso fare: s->info s->next s->riferimenti
bolletta::nodo* bolletta::smartp::operator->() const {
    return punt;
}

// OPERATORE DI UGUAGLIANZA
bool bolletta::smartp::operator==(const smartp& p) const
{ return punt == p.punt; }

// OPERATORE DI DISUGUAGLIANZA
  
```

3.13 Condivisione controllata della memoria

```
bool bolletta::smartp::operator!=(const smartp& p) const  
{ return punt != p.punt; }
```

Per la classe interna nodo abbiamo solamente le seguenti due definizioni dei costruttori.

```
// Metodi di nodo: 2 costruttori  
  
bolletta::nodo::nodo(): riferimenti(0) {}  
// NB: invocazioni info() e next() implicite  
  
bolletta::nodo::nodo(const telefonata& t, const smartp& p)  
: info(t), next(p), riferimenti(0) {}  
// NB: next(p) è una invocazione al  
// costruttore di copia ridefinito di smartp
```

Passiamo quindi ai metodi di bolletta. Grazie ai puntatori smart che gestiscono "automaticamente" il campo dati riferimenti, le definizioni ritornano ad essere semplici.

```
bool bolletta::Vuota() const { return first == 0; }  
// NB: cast implicito da 0 a smartp  
  
void bolletta::Aggiungi_Telefonata(const telefonata& t){  
    first = new nodo(t,first);  
    // NB: cast implicito da nodo* a smartp  
}  
  
telefonata bolletta::Estrai_Una(){  
    // Precondizione: bolletta non vuota  
    // Attenzione: non va fatta la delete sul primo nodo  
    telefonata aux = first->info;  
    first = first->next;  
    return aux;  
}  
  
void bolletta::Togli_Telefonata(const telefonata& t) {  
    smartp p = first, prec, q;  
    smartp original = first; // first originale  
    first = 0;  
    // p scorre la lista, prec è il nodo che precede  
    // quello puntato da p, q punta alla copia del nodo  
    while (p!=0 && !(p->info == t)) { // p è uno smartp  
        // quindi notare il "p!=0"  
        // fa una copia dei nodi che precedono  
        // (eventualmente) quello da togliere
```

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```
q = new nodo(p->info, p->next);
// e li inserisce nella nuova lista puntata da first
if (prec == 0) first = q;
else prec->next = q;
// aggiorna prec e p per il ciclo while
prec = q; p = p->next;
}

// se p == 0 la telefonata non è stata trovata, ho inutilmente
// copiato la lista, quindi ripristino la situazione originale
if (p==0) { first = original; }
else if (prec == 0) first = p->next; // t era in testa
else prec->next = p->next;
// le variabili locali p, q, prec vengono distrutte
}

ostream& operator<<(ostream& os, const bolletta& b) {
if(b.Vuota()) os << "BOLLETTA VUOTA" << endl;
else {
    os << "TELEFONATE IN BOLLETTA" << endl;
    bolletta::smartp p = b.first; // amicizia
    int i = 1;
    while (p!=0) {
        os << i++ << ") " << p->info << endl;
        p = p->next;
    }
}
return os;
}
```

Esercizio 3.13.1. Aggiungere a bolletta un metodo pubblico

```
void Sostituisci(const telefonata& t1, const telefonata& t2);
```

che modifica la bolletta di invocazione sostituendo la prima (eventuale) occorrenza della telefonata t_1 con la telefonata t_2 . Tale definizione deve gestire la memoria in modo controllato. (*Suggerimento:* fare la copia della bolletta sino a t_1)

Un esercizio più difficile consiste nel definire un metodo

```
void Sostituisci_Tutte(const telefonata& t1, const telefonata& t2);
```

che modifica la bolletta di invocazione sostituendo tutte le (eventuali) occorrenze della telefonata t_1 con la telefonata t_2 . Naturalmente, la memoria deve essere gestita in modo controllato.

3.14 Progetto di metà corso: Polinomi in una variabile

Un polinomio a coefficienti interi (ad esempio $2x^4 - 3x^2 + x + 6$ è un polinomio nella variabile x a coefficienti interi) è rappresentato come una lista dinamica di monomi ordinati per esponente decrescente. Il polinomio nullo è rappresentato con la lista vuota.

Vogliamo gestire la memoria tramite la tecnica della condivisione controllata. Quindi, un monomio è rappresentato tramite una classe `Monomio` avente i seguenti campi dati.

`coefficiente`: il coefficiente intero del monomio;

`esponente`: l'esponente del monomio;

`riferimenti`: numero di puntatori che puntano al monomio;

`next`: puntatore smart al monomio successivo.

Come abbiamo visto, un puntatore smart ad un `Monomio` è rappresentato tramite una classe con un unico campo dati di tipo puntatore a `Monomio` (nella classe `Monomio` è presente il campo dati `riferimenti`). Per rendere smart tale classe si ridefiniscono il costruttore di copia, il distruttore e l'assegnazione in modo tale da controllare la condivisione di memoria. Inoltre si ridefiniranno gli operatori `*`, `->`, `==`, `!=`, etc (tutto ciò che risulterà necessario).

Possiamo quindi impostare la classe `Polinomio` nel seguente modo.

```
class Polinomio {
private: // parte privata di Polinomio
    class Monomio; // dichiarazione incompleta
    class SmartP {
public:
    Monomio* punt;
    ...
};
class Monomio {
public:
    int coefficiente, esponente, riferimenti;
    SmartP next; // puntatore smart
    ...
};
SmartP first; // smart pointer al primo monomio
public: // parte pubblica di Polinomio
    ...
};
```

La parte pubblica di `Polinomio` dovrà contenere costruttori, metodi e ridefinizioni di operatori in modo tale che il seguente esempio di `main()` compili ed esegui provocando le stampe riportate a commento.

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

```

int main() {
    Polinomio X(1,1);
    cout << "X = " << X << endl; // stampa: X = x
    Polinomio Z(3, 4);
    cout << "Z = " << Z << endl; // stampa: Z = 3x^4
    Polinomio T(X);
    cout << "T = " << T << endl; // stampa: T = x
    T = Z;
    cout << "T = " << T << endl; // stampa: T = 3x^4
    cout << "X = " << X << endl; // stampa: X = x
    Polinomio Q=3*X + 2*X*X - X*X*X + 7*(X^4);
    cout << "Q = " << Q << endl; // stampa: Q = 7x^4-x^3-2x^2+3x
    Polinomio P = 2*(X^2);
    cout << "P = " << P << endl; // stampa: P = 2x^2
    cout << "Q*Q = " << Q*Q << endl;
        // stampa: Q*Q = 49x^8-14x^7+29x^6+38x^5-2x^4+12x^3+9x^2
    cout << "Q/P = " << Q/P << endl; // stampa: Q/P = 3x^2+1
    cout << "Q^3 = " << (Q^3) << endl;
        // stampa: Q^3 = 2401x^32-1372x^27+2744x^26+...
    cout << "Q%P = " << Q%P << endl; // stampa: Q/P = x^4-x^3+3x
    cout << "Q(3) = " << Q(3) << endl; // stampa: Q(3) = 567
    if (P == 2*(X^2)) cout << "P(3) = " << P(3) << endl;
        // stampa: P(3) = 18
    if (P != Q) cout << "Q(P(3)) = " << Q(P(3)) << endl;
        // stampa: Q(P(3)) = 729702
}

```

Per realizzare i metodi ricorsivamente è comodo pensare i polinomi $P^n(x)$ definiti ricorsivamente sul grado $n \geq -1$ come segue:

- Il polinomio nullo 0 è l'unico polinomio $P^{-1}(x)$ di grado -1:

$$P^{-1}(x) = 0;$$

- Un polinomio $P^n(x)$ di grado $n \geq 0$ è la somma di un monomio ax^n di grado n e coefficiente $a \neq 0$ con un polinomio $R^k(x)$ di grado $k < n$:

$$P^n(x) = ax^n + R^k(x).$$

Ad esempio la somma $P^n(x) + Q^m(x)$ di due polinomi $P^n(x)$ e $Q^m(x)$ può essere definita (e calcolata) ricorsivamente nel modo seguente:

- Se $P^n(x) = 0$ allora $P^n(x) + Q^m(x) = Q^m(x)$.
- Se $Q^m(x) = 0$ allora $P^n(x) + Q^m(x) = P^n(x)$.

3.14 Progetto di metà corso: Polinomi in una variabile

- Altrimenti, cioè quando $n \geq 0$ e $m \geq 0$,

$$P^n(x) = ax^n + R^k(x) \quad \text{e} \quad Q^m(x) = bx^m + T^h(x)$$

dove:

- se $n > m$ allora $P^n(x) + Q^m(x) = ax^n + (R^k(x) + Q^m(x));$
- se $n < m$ allora $P^n(x) + Q^m(x) = bx^m + (P^n(x) + T^h(x));$
- se $m = n$ e
 - * $a + b = 0$ allora $P^n(x) + Q^m(x) = R^k(x) + T^h(x);$
 - * $a + b \neq 0$ allora $P^n(x) + Q^m(x) = (a + b)x^m + (R^k(x) + T^h(x)).$

Volendo usare queste definizioni sarà opportuno definire due metodi privati che estraggono da un polinomio non nullo il primo monomio e il resto del polinomio.

3 CLASSI COLLEZIONE E ARGOMENTI CORRELATI

Capitolo 4

Template

Il C++ è un linguaggio strongly typed: il programmatore è tenuto a specificare il tipo di ogni elemento sintattico che durante l'esecuzione denota un valore (ad esempio una variabile o una espressione) ed il linguaggio garantisce che tale valore sia utilizzato in modo coerente con il tipo specificato (ad esempio, non è possibile eseguire una moltiplicazione tra un intero ed un puntatore). Questo permette un controllo accurato del programma da parte del compilatore ed evita molti errori di programmazione. D'altra parte in certe situazioni la tipizzazione forte può risultare troppo vincolante. In qualche caso le conversioni implicite ed esplicite, se usate in modo corretto, possono alleviare tale vincolo senza diminuire sensibilmente i vantaggi della tipizzazione forte. Vi sono però varie situazioni rilevanti che non possono essere risolte mediante conversioni di tipo. Ad esempio, se volessemo definire una funzione che calcoli il minimo tra due valori dobbiamo necessariamente definire una funzione diversa per ogni tipo di valori: int, float, orario, string, etc.

```
int min (int a, int b) { return a < b ? a : b; }

float min (float a, float b) { return a < b ? a : b; }

orario min (orario a, orario b) { return a < b ? a : b; }

string min (string a, string b) { return a < b ? a : b; }
```

Una soluzione attraente ma subdolamente pericolosa potrebbe essere la definizione di *macro* istruzioni che vengono espanso sintatticamente nella fase di precompilazione. Ad esempio, usando la direttiva:

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

il preprocessore effettua le sostituzioni:

```
min(10,20)    con    10 < 20 ? 10 : 20
min(3.5,2.1)  con    3.5 < 2.1 ? 3.5 : 2.1
```

Questo funziona nei casi "semplici". Ma non abbiamo ottenuto una "vera" funzione: ad esempio l'istruzione `min(++i,--j)`; verrebbe sostituita dal preprocessore con:

```
++i < --j ? ++i : --j;
```

e provocherebbe una doppia applicazione dell'operatore `++` o dell'operatore `--`. Questo non è quello che si voleva, come dimostra il seguente esempio:

```
int i=3, j=6; cout << min(++i,--j); // stampa 5 !
```

La soluzione consiste nell'usare un *template di funzione*, che assieme ai *template di classe* implementano in C++ l'idea del cosiddetto *polimorfismo parametrico*.

4.1 Template di funzione

Un template di funzione è essenzialmente la descrizione di un metodo (o modello, traduzione letterale del termine inglese *template*) che il compilatore può usare per generare automaticamente istanze particolari di una funzione che differiscono per il tipo degli argomenti. Un template di funzione può essere sia una funzione globale (esterna ad una classe) che un metodo di una classe.

La definizione della funzione `min` come template è la seguente:

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

Dopo una tale definizione di template di funzione possiamo scrivere le seguenti istruzioni.

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

4.1 Template di funzione

La parola chiave `template` appare sia all'inizio delle dichiarazioni che delle definizioni di un template di funzione. Essa è seguita dalla lista (non vuota) dei *parametri del template* separati da virgolette e racchiusi tra i simboli “`<`” e “`>`”. I parametri possono essere *parametri di tipo* che sono preceduti dalla parola chiave `class` (oppure `typename`) e che dovranno essere istanziati con un tipo qualsiasi. Oppure *parametri valore* che sono parametri preceduti dal tipo di appartenenza del valore e devono essere istanziati con un valore costante del tipo indicato (quindi noto a tempo di compilazione). Il nome del parametro valore può essere usato solamente come un valore costante nella definizione del template.

La “costruzione” di una specifica funzione per alcuni parametri attuali di tipo e valore dallo schema del template viene detta *istanziazione del template*. Un template di funzione può essere istanziato implicitamente oppure esplicitamente. Nell’istanziazione *implicita* di un template i tipi e i valori effettivi dei parametri di tipo e valore del template sono dedotti automaticamente dai parametri attuali che sono passati come argomenti al template. Si tratta del cosiddetto processo di deduzione degli argomenti del template. Ad esempio:

```
int main() {  
    int i,j,k; orario r,s,t;  
    ...  
    // istanziazione implicita del template  
    k = min(i,j); // istanza: int min(int,int)  
    t = min(r,s); // istanza: orario min(orario,orario)  
    ...  
}
```

Nell’istanziazione implicita il tipo di ritorno dell’istanza del template non viene considerato nella deduzione degli argomenti. Ad esempio:

```
int main() {  
    double d; int i,j;  
    ...  
    d = min(i,j);  
    // istanza: int min(int,int) e quindi usa  
    // la conversione int => double  
}
```

Nella deduzione degli argomenti sono ammesse essenzialmente quattro sole tipologie di conversioni dal tipo dell’argomento attuale al tipo dei parametri del template:

1. conversione da l-valore in r-valore (in particolare da `T&` a `T`);
2. da array a puntatore, cioè da `T[]` a `T*`;
3. conversione di qualificazione costante, i.e. da `T` a `const T`;
4. conversione da r-valore a riferimento costante, cioè da r-valore di tipo `T` a `const T&`.

4 TEMPLATE

Ad esempio:

```
template <class T> void F(const T x) {...}
template <class T> void G(T* p) {...}
template <class T> void H(const T& y) {...}

int main() {
    int i = 6;
    int a[3] = {4,2,9};
    F(i);           // istanzia: void F(const int)
    G(a);           // istanzia: void G(int*)
    H(7);           // istanzia: void H(const int&)
}
```

L'algoritmo di deduzione degli argomenti di un template procede esaminando tutti gli argomenti attuali nella chiamata del template di funzione da sinistra verso destra. Se si trova uno stesso parametro *T* del template che appare più volte come parametro di tipo, l'argomento del template dedotto per *T* da ogni argomento attuale deve essere *esattamente* lo stesso. Nel seguente esempio l'algoritmo di deduzione degli argomenti del template non riesce ad ottenere un unico argomento e pertanto il compilatore segnala un errore in compilazione.

```
int main() {
    int i; double d, e;
    ...
    e = min(i,d);
    // Non compila:
    // si deducono due diversi argomenti del template: int e double
}
```

Quindi, se il processo di deduzione degli argomenti del template non porta ad una istanziazione univoca si ottiene un errore di compilazione.

È possibile specificare esplicitamente gli argomenti di un template: si tratta dell'istanziazione esplicita degli argomenti dei parametri del template di funzione. Naturalmente, nell'invocazione di un template di funzione *F* istanziato esplicitamente è possibile applicare qualsiasi conversione implicita di tipo per convertire un parametro attuale per *F* nel tipo del corrispondente parametro dell'istanza esplicita di *F*. Ad esempio:

```
int main() {
    int i; double d, e; ...
    e = min<double>(i,d);
    // Compila correttamente e istanzia: double min(double,double)
    // e quindi converte implicitamente i da int a double
}
```

4.2 Modelli di compilazione dei template di funzione

Soltamente si usa l'istanziazione esplicita solamente quando risulti necessaria per risolvere ambiguità o per usare particolari istanze di template in contesti in cui la deduzione degli argomenti non è possibile.

Esempio 4.1.1. Ricordiamo che un parametro di una funzione può anche essere un *riferimento ad un array statico*. In questo caso, la dimensione costante dell'array è parte integrante del tipo del parametro e il compilatore controlla che la dimensione dell'array passato come parametro attuale coincida con quella specificata nel tipo del parametro. Consideriamo ad esempio la seguente funzione.

```
int min(int (&a) [7]) { // array di 7 int
    int m = a[0];
    for (int i = 1; i < 7 ; i++)
        if (a[i] < m) m = a[i];
    return m;
}
```

È immediato generalizzare questa funzione ad un template con un parametro di tipo ed un parametro valore.

```
template <class T, int size>
T min(T (&a) [size]) {
    T vmin = a[0];
    for (int i = 1; i < size ; i++)
        if (a[i] < vmin) vmin = a[i];
    return vmin;
}

int main() {
    int ia[20]; orario oa[50];
    ...
    cout << min(ia);
    cout << min(oa);
    // oppure
    cout << min<int,20>(ia);
    cout << min<orario,50>(oa);
}
```

4.2 Modelli di compilazione dei template di funzione

Come abbiamo già detto, la definizione di un template di funzione è solamente uno schema per la definizione di un numero potenzialmente infinito di funzioni. Di per sé una definizione di template di funzione non può essere compilata in qualche forma di codice macchina che possa essere effettivamente eseguito. Il compilatore, quando incontra una definizione

di template di funzione, si limita a memorizzare una rappresentazione interna di tale definizione senza effettuare alcuna tipologia di vera compilazione. Ad esempio, quando il compilatore incontra la definizione:

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

sostanzialmente si limita a memorizzarla, senza effettuare traduzioni in linguaggio macchina. Soltanto quando incontrerà un uso effettivo del template di funzione genererà il codice macchina corrispondente alla particolare istanza di funzione utilizzata.

Questo pone alcuni interrogativi.

1. Affinché il compilatore possa generare l'istanza della funzione, la definizione del template deve essere visibile nel punto in cui compare la chiamata di funzione?
2. Occorre mettere le definizioni dei template di funzioni in un file header da includere in ogni file che richiede una istanziazione dei template?
3. Oppure nei file header si mettono soltanto le dichiarazioni dei template di funzione mentre le definizioni dei template si possono mettere in un file da "compilare" separatamente?

Per rispondere a queste domande occorre analizzare come il compilatore tratta i template. Il C++ standard prevede due modelli di compilazione dei template: la *compilazione per inclusione* e la *compilazione per separazione*.

4.2.1 Compilazione per inclusione

Con la compilazione per inclusione le definizioni dei template vengono messe in file header. Tali file header devono quindi essere inclusi in ogni file in cui vengono istanziati i template. Il compilatore usa quindi queste definizioni per generare il codice di tutte le istanze del template utilizzate nel file che sta compilando.

Questo semplice modello di compilazione presenta due sostanziali inconvenienti:

- (1) Il file header contiene tutti i dettagli della definizione dei template. Tali dettagli risultano quindi visibili all'utente e ciò va quindi contro il principio dell'information hiding.
- (2) Se la stessa istanza di un template viene usata in più file compilati separatamente il codice per tale istanza viene generato più volte dal compilatore.

Il problema (2) non crea inconvenienti a tempo di esecuzione in quanto al momento del collegamento dei file compilati separatamente il linker usa soltanto una delle istanze compilate mentre le altre vengono semplicemente ignorate. Tuttavia la compilazione ripetuta delle stesse istanze e l'inclusione di file header molto lunghi può rallentare il processo di

4.2 Modelli di compilazione dei template di funzione

compilazione in modo significativo. In alcuni compilatori il problema di efficienza si può alleviare selezionando una particolare opzione del compilatore che impedisce la generazione automatica del codice delle istanze dei template. In questo caso occorre forzare il compilatore a generare le istanze del template che vengono usate nel programma con delle cosiddette *dichiarazioni esplicite di istanziazione*.

Per il template di funzione

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

una dichiarazione esplicita di istanziazione al tipo int assume la seguente forma:

```
template int min(int,int);
```

e forza il compilatore a generare il codice dell'istanza del template relativa al tipo int. La definizione del template di funzione deve essere fornita nel file in cui compare la dichiarazione esplicita di istanziazione. Per quanto riguarda gli altri file che usano la stessa istanza del template, un parametro fornito al compilatore comunica che una dichiarazione esplicita di istanziazione compare in un altro file e che il template di funzione non deve essere istanziato. Nel caso del compilatore g++ dobbiamo invocare il compilatore con la seguente opzione:

```
g++ -fno-implicit-templates
```

4.2.2 Compilazione per separazione

Con la compilazione per separazione nel file header vengono messe soltanto le dichiarazioni dei template mentre le definizioni vengono messe in un file che può essere "compilato separatamente". Per avvisare il compilatore che la definizione del template di funzione potrebbe essere necessaria per generare istanze del template usate in altri file occorre farla precedere dalla parola chiave export. Si consideri il seguente esempio.

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

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

// file "main.cpp"
#include "min.h"
int main() {
    int i,j,k; orario r,s,t;
    ...
    k = min(i,j);
```

```

...
t = min(r,s);
...
}

```

Il modello di compilazione per separazione è conveniente. Tuttavia non tutti i compilatori supportano questo modello e anche quelli che lo supportano non sempre lo fanno bene, perché sono necessari ambienti di programmazione sofisticati, non disponibili in tutte le implementazioni del C++. Il compilatore g++ ha iniziato a supportare alcune forme di compilazione per separazione dei template.

4.3 Template di classe

Un *template di classe* è essenzialmente la descrizione di un metodo (modello) che il compilatore può usare per generare automaticamente istanze particolari di una classe che differiscono per il tipo di alcuni membri, cioè campi dati, metodi o classi interne, propri della classe. Supponiamo ad esempio di voler definire una classe Queue che implementa una coda con politica di inserimento/rimozione First In First Out (FIFO). Tale classe dovrà implementare una coda in cui gli elementi (o item) vengono estratti nello stesso ordine in cui sono inseriti. La dichiarazione della classe Queue potrebbe essere la seguente, dove *Tipo* è il tipo degli elementi della coda.

```

class Queue {
public:
    Queue();
    ~Queue();
    bool is_empty() const; // testa se la coda è vuota
    void add(const Tipo&); // aggiunge un item alla coda
    Tipo remove();         // rimuove un item alla coda
private:
    ...
};

```

Naturalmente, il compilatore quando compila la classe Queue ed i suoi metodi deve conoscere a quale tipo effettivo ci si vuole riferire. Ad esempio, se vogliamo definire una coda di interi dobbiamo sostituire *int* al posto di *Tipo* in tutti i punti della definizione della classe in cui compare. Un modo più comodo potrebbe essere quello di usare una definizione di tipo mediante un *typedef*:

```
typedef int Tipo;
```

In entrambi i casi, se nello stesso programma dobbiamo usare sia code di interi che code di stringhe, dobbiamo scrivere due definizioni distinte della classe e con due nomi diversi. Ad esempio:

4.3 Template di classe

```
class QueueInt {  
public:  
    Queue();  
    ~Queue();  
    bool is_empty() const;  
    void add(const int&);  
    int remove();  
private:  
    ...  
};
```

```
class QueueString {  
public:  
    Queue();  
    ~Queue();  
    bool is_empty() const;  
    void add(const string&);  
    string remove();  
private:  
    ...  
};
```

Duplicare le classi in questo modo è noioso, allunga inutilmente le righe di programma da scrivere e soprattutto rende difficile la modifica ed il riutilizzo del software. Immaginiamo di dover propagare a tutte le copie della classe una modifica o una correzione di uno dei metodi propri della classe! Possiamo evitare questi problemi usando i template di classe, ovvero parametrizzando la classe rispetto ai tipi. La definizione di un template per la classe Queue può essere la seguente:

```
template <class T>  
class Queue {  
public:  
    Queue();  
    ~Queue();  
    bool is_empty() const;  
    void add(const T&);  
    T remove();  
private:  
    ...  
};
```

Quindi, per definire, rispettivamente, una coda `qi` di interi, una coda `qb` di bollette e una coda `qs` di stringhe, scriveremo:

```
Queue<int> qi;  
Queue<bolletta> qb;  
Queue<string> qs;
```

Come per i template di funzione, anche nei template di classe vi sono due tipologie di parametri: *parametri di tipo* e *parametri valore*. I primi sono preceduti dalla keyword `class` (oppure `typename`) mentre i secondi sono preceduti dal tipo del valore. A differenza dei template di funzione dove, in mancanza di una istanziazione esplicita dei parametri del template di funzione il compilatore cerca di dedurre automaticamente l'istanziazione da un esame dei tipi dei parametri attuali della funzione, con i template di classe occorre sempre istanziare in modo esplicito i parametri del template. I parametri, sia di tipo che valore, in un template di classe possono avere un valore di default. Le regole per l'uso dei

valori di default nei template di classe sono analoghe a quelle già note per l'uso dei valori di default nelle funzioni. Ad esempio:

```
template <class T = int, int size = 1024>
class Buffer {
    ...
};

int main() {
    Buffer<> ib;           // Buffer<int,1024>
    Buffer<string> sb;     // Buffer<string,1024>
    Buffer<string,500> sbs; // Buffer<string,500>
}
```

Illustreremo le caratteristiche dei template di classe esaminando nel dettaglio una definizione completa della classe template Queue. Per ora consideriamo sola la dichiarazione di Queue. Queue usa un template di classe esterna QueueItem per gli oggetti che sono elementi della coda. Implementeremo la coda come una lista puntata da due puntatori, uno al primo nodo ed uno all'ultimo nodo.

```
template <class T>
class QueueItem {
public:           // per ora tutto public
    QueueItem(const T&);
    T info;
    QueueItem* next;
};

template <class T>
class Queue {
public:
    Queue();      // notare la sintassi: Queue e non Queue<T>
    ~Queue();
    bool is_empty() const;
    void add(const T&);
    T remove();
private:
    // notare la sintassi: QueueItem<T> e non QueueItem
    QueueItem<T>* primo;
    QueueItem<T>* ultimo;
};
```

4.3.1 Istanziazione di un template di classe

La definizione di un template di classe specifica come si possono costruire delle istanze di classe qualora vengano forniti i valori dei parametri del template. Ad esempio, soltanto quando il compilatore incontra una istruzione quale `Queue<int> qi;` costruisce effettivamente una classe "coda di interi" di nome `Queue<int>`, a meno che non lo abbia già fatto precedentemente, dopo di che costruisce anche l'oggetto `qi` appartenente a tale classe. Se a causa di una istruzione quale: `Queue<string> qs;` il compilatore costruisce anche la classe "coda di stringhe" `Queue<string>` allora le due classi, benché costruite usando lo stesso template di classe, sono due classi *completamente distinte*. In particolare, `Queue<int>` non ha accesso alla parte privata di `Queue<string>` e viceversa. Il nome di una istanza di template di classe come `Queue<int>` o `Queue<string>` si può usare in qualsiasi punto del programma in cui si può usare il nome di una classe normale. Nella dichiarazione o definizione di un template (di classe o di funzione) possono comparire sia nomi di istanze di template di classe sia nomi di template di classe. Ad esempio:

```
template <class T>
int f(Queue<T>& qt, Queue<string> qs);
// Queue<T> template di classe
// Queue<string> istanza di template di classe
```

Invece, al di fuori di definizioni o dichiarazioni di template (di classe o funzione) possono comparire solo nomi di istanze di template di classe. L'occorrenza in un programma del nome di una istanza di un template di classe non è sufficiente perché il compilatore generi tale istanza. Occorre che il nome compaia in un contesto che richieda un utilizzo effettivo di tale definizione. Ad esempio, il compilatore non genera l'istanza `Queue<int>` quando incontra le due seguenti occorrenze del nome dell'istanza:

```
template <class T> class Queue;

void Stampa(const Queue<int>& q) {
    Queue<int>* pqi = &q;
    ...
}
```

perché non è necessaria l'istanza della classe `Queue` per copiare un riferimento o un puntatore a `Queue`. Non è quindi necessaria la definizione del template ma basta la sua dichiarazione incompleta. Questo vale per le definizioni di riferimenti e puntatori ad una classe che infatti non richiedono un utilizzo effettivo della classe. Come noto, questo in effetti vale per ogni classe e non solo per le istanze di un template di classe. Infatti la definizione

```
class Queue; // dichiarazione incompleta

void Stampa(const Queue& q) {
    Queue* pqi = &q;
    ...
}
```

funziona perché la dichiarazione incompleta della classe è sufficiente per definire puntatori o riferimenti ad oggetti della classe.

Quando viene effettivamente istanziato un template di classe? Ovviamamente la definizione del template è necessaria quando si deve operare su oggetti della classe. Il compilatore è costretto a generare l'istanza del template di classe con

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

void Stampa(Queue<int> q) {
    Queue<int> qi;
    ...
}
```

perché l'istanza `Queue<int>` del template serve per allocare lo spazio per i due oggetti `q` e `qi`. Deve quindi essere visibile la definizione del template di classe `Queue`.

Consideriamo il seguente esempio più sottile: il compilatore è pure costretto a generare l'istanza `Queue<int>` con

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

void Stampa(const Queue<int>& q) {
    Queue<int>* pqi = &q;
    pqi++;
    ...
}
```

perché questa istanza della classe serve per calcolare la quantità `sizeof(Queue<int>)` di cui occorre incrementare il puntatore per eseguire `pqi++`.

4.3.2 Metodi di template di classe

Come per le normali classi, in un template di classe la definizione di un metodo può comparire sia all'interno (come metodo inline) sia all'esterno della classe. Ad esempio, si consideri la seguente definizione inline del costruttore della classe `Queue`.

```
template <class T>
class Queue {
    ...
public:
    Queue() : primo(0), ultimo(0) {}
    ...
};
```

4.3 Template di classe

La definizione esterna di un metodo di un template di classe richiede la seguente sintassi:

```
template <class T>
class Queue {
    ...
public:
    Queue();
    ...
};

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

Un metodo di un template di classe non viene istanziato quando viene istanziata la classe ma se e soltanto quando il programma usa effettivamente quel metodo. Per avere un esempio di definizione di metodi completiamo la definizione dei template di classe QueueItem e Queue. Per completare la definizione del template QueueItem basta aggiungere la definizione del costruttore ad un argomento che mettiamo inline.

```
// file "Queue.h"
#ifndef QUEUE_H
#define QUEUE_H

template <class T>
class QueueItem {
public:
    // per gli scopi di Queue basta questo costruttore
    QueueItem(const T& val) : info(val), next(0) {}
    T info;
    QueueItem* next;
};

template <class T>
class Queue {
public:
    Queue() : primo(0), ultimo(0) {}
    bool is_empty() const;
    void add(const T&);
    T remove();
    ~Queue();
private:
    QueueItem<T>* primo;
```

4 TEMPLATE

```
    QueueItem<T>* ultimo;  
};
```

Seguendo il modello di compilazione per inclusione dei template di funzione, le definizioni dei metodi andranno nello stesso file header `Queue.h`.

```
// sempre nel file "Queue.h"  
template <class T>  
bool Queue<T>::is_empty() const { return (primo == 0); }  
  
template <class T>  
void Queue<T>::add(const T& val) {  
    QueueItem<T>* p = new QueueItem<T>(val);  
    if(is_empty())  
        primo = ultimo = p;  
    else { // aggiunge in coda  
        ultimo->next = p; ultimo = p;  
    }  
}  
  
#include <iostream>  
using std::cerr; using std::endl;  
  
template <class T>  
T Queue<T>::remove() {  
    if (is_empty()) {  
        cerr << "remove() su coda vuota" << endl;  
        exit(1);  
    }  
    QueueItem<T>* p = primo;  
    primo = primo->next;  
    T aux = p->info;  
    delete p;  
    return aux;  
}  
  
template <class T>  
Queue<T>::~Queue() {  
    while (!is_empty()) remove();  
}  
#endif
```

Analogamente ai template di funzione, anche per i template di classe il C++ standard prevede i modelli di compilazione per inclusione e per separazione. Come si è già notato, purtroppo molti compilatori non supportano il modello di compilazione per separazione, ed

4.3 Template di classe

il compilatore g++ lo supporta in forma limitata. Non tratteremo ulteriormente il modello per separazione. Useremo invece il semplice modello di compilazione per inclusione: la dichiarazione e la definizione di un template di classe sono entrambe poste in un file header che deve essere incluso in ogni file che voglia usare il template di classe. Questo file header, quindi, conterrà anche le definizioni esterne dei metodi del template di classe. Ad esempio, nella libreria del compilatore g++, si vedano i file header delle classi contenitore (che sono dei template) della STL.

Vediamo un esempio che mostra quando vengono create le istanze di template di classe e di metodi.

```
#include<iostream>
using std::cout; using std::endl;
#include "Queue.h"

int main() {
    Queue<int>* pi = new Queue<int>;
    // vengono istanziati la classe Queue<int> ed il suo costruttore
    // Queue<int>() perché new deve costruire un oggetto della classe
    int i;
    for (i = 0; i < 10; i++) pi->add(i);
    // vengono istanziati i metodi add<int> e is_empty<int>, la
    // classe QueueItem<int> e il suo costruttore QueueItem<int>()
    for (i = 0; i < 10; i++) cout << pi->remove() << endl;
    // viene istanziato il metodo remove<int>
}
```

4.3.3 Dichiarazioni friend in template di classe

Analizziamo le possibili dichiarazioni di amicizia in un template di classe. In un template di classe possono apparire tre tipologie di dichiarazioni friend.

1. Dichiarazione nel template di classe C di una classe o funzione friend non template.
Vediamo un esempio.

```
class A { ... int fun(); ... };

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

In questo caso, la classe B, la funzione test () e il metodo A::fun () della classe A sono friend di *tutte le istanze* del template di classe C. Si noti che non serve dichiarare o definire B e test () prima del template C mentre occorre che la classe A sia stata dichiarata prima del template C affinché sia visibile a C che la classe A include un metodo fun () .

- Dichiarazione nel template di classe C di un template di classe o di un template di funzione friend associato, cioè avente tra i suoi parametri alcuni dei parametri del template C. In questo caso tutte le istanze della classe template C hanno come amica *una ed una sola corrispondente istanza* del template di classe friend associato o del template di funzione friend associato. Vediamo un esempio.

```
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 funzione 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);
};
```

Si tratta del caso più comune e significativo. Ad ogni istanza del template di classe C rimane associata come amica una ed una sola istanza del template di classe B e dei template di funzione A<T>::fun () e test. Le dichiarazioni dei template di classe A e B e del template di funzione test devono essere visibili quando viene definito il template di classe C. Si noti che per poter fare la dichiarazione del template di funzione test è necessario che sia visibile una dichiarazione incompleta del template di classe C.

- Dichiarazione nel template di classe C di un template di classe o di un template di funzione friend non associato, cioè aventi i parametri disgiunti dai parametri di C (vale a dire che i due insiemi di parametri sono disgiunti). In questo caso, i template di classe o di funzione dichiarati friend sono friend di ogni istanza del template di classe C. Vediamo anche in questo caso un esempio.

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

4.3 Template di classe

```
T t;

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

template<class Tp>
friend class B;

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

};
```

Alcuni compilatori pre-standard ancora non supportano quest'ultima tipologia di dichiarazioni friend. Il compilatore g++ supporta i template di classe e di funzione friend non associati a partire dalla versione 3.x.

Esempio 4.3.1. Il seguente programma compila e la sua esecuzione provoca le stampe riportate.

```
// dichiarazione incompleta del template di classe C
template<class T> class C;

// dichiarazione del template di funzione f_friend
template<class T>
void f_friend(C<T>);

template<class T>
class C {
friend void f_friend<T>(C<T>);
private:
    T t;
public:
    C(T x) : t(x) {}
};

template<class T>
void f_friend(C<T> c) {
    cout << c.t << endl; // per amicizia
}

int main() {
    C<int> c1(1); C<double> c2(2.5);
    f_friend(c1); // stampa: 1
    f_friend(c2); // stampa: 2.5
}
```

4 TEMPLATE

Esempio 4.3.2. Il seguente programma compila e la sua esecuzione provoca le stampe riportate.

```
template <class T>
class C {
    template <class V>
    friend void fun(C<V>);
private:
    T x;
public:
    C(T y) : x(y) {}
};

template <class T>
void fun(C<T> t) {
    cout << t.x << " ";
    C<double> c(3.1);
    cout << c.x << endl; // ok grazie all'amicizia non associata
}

int main() {
    C<int> c(4);
    C<string> s("blob");
    fun(c); // stampa: 4 3.1, istanziazione implicita fun<int>
    fun(s); // stampa: blob 3.1, istanziazione implicita fun<string>
}
```

Vediamo quindi le definizioni friend per le classi QueueItem e Queue. Dal momento che la classe QueueItem non deve essere usata al di fuori della classe Queue dichiariamo privati tutti i suoi membri (costruttore incluso), e quindi dobbiamo dichiarare in QueueItem che Queue è classe amica di QueueItem.

```
template <class T>
class QueueItem {
private:
    T info;
    QueueItem* next;
    QueueItem(const T& val) : info(val), next(0) {}
};
```

Potremmo dichiarare tutte le istanze di Queue classi amiche di ogni istanza di QueueItem, cioè Queue friend non associato di QueueItem come segue:

```
template <class T>
class QueueItem {
```

4.3 Template di classe

```
template<class Tp> friend class Queue;
private:
    T info;
    QueueItem* next;
    QueueItem(const T& val) : info(val), next(0) {}
};
```

Tuttavia, ciò non sarebbe soddisfacente per i nostri scopi: infatti, non avrebbe molto senso che la classe `Queue<int>` fosse amica della classe `QueueItem<string>`. Conviene quindi associare ad ogni istanza di `QueueItem` una sola istanza amica della classe `Queue`, ovvero quella associata. Optiamo quindi per la seguente dichiarazione di amicizia associata:

```
template <class T>
class QueueItem {
    friend class Queue<T>;
private:
    T info;
    QueueItem* next;
    QueueItem(const T& val) : info(val), next(0) {}
};
```

Vogliamo ora definire l'overloading dell'operatore di output per una coda. La funzione `operator<<` deve essere dichiarata come funzione amica del template di classe `Queue` perché deve accedere ai suoi campi privati. Il suo prototipo dovrebbe essere uno od entrambi tra i due seguenti?

```
ostream& operator<<(ostream&, const Queue<int>&);
```

```
ostream& operator<<(ostream&, const Queue<string>&);
```

Naturalmente, né l'uno né l'altro. La soluzione migliore consiste nell'usare un template di funzione esterna:

```
template <class T>
ostream& operator<<(ostream&, const Queue<T>&);

template <class T>
ostream& operator<<(ostream& os, const Queue<T>& q) {
    os << "(";
    QueueItem<T>* p = q.primo; // per amicizia con Queue
    for (; p != 0; p = p->next) // per amicizia con QueueItem
        os << *p << " ";           // operator<< per QueueItem
    os << ")" << endl;
    return os;
}
```

4 TEMPLATE

Dobbiamo quindi dichiarare `operator<<` come funzione amica associata sia del template `Queue` che del template `QueueItem`.

```
// dichiarazione incompleta
template <class T> class Queue;

// dichiarazione incompleta
template <class T> ostream& operator<<(ostream&, const Queue<T>&);

template <class T>
class Queue {
    friend ostream& operator<< <T>(ostream&, const Queue<T>&);
    ...
};

template <class T>
class QueueItem {
    friend ostream& operator<< <T>(ostream&, const Queue<T>&);
    ...
};
```

Inoltre, dobbiamo definire `operator<<` per il template di classe `QueueItem`.

```
template <class T>
ostream& operator<<(ostream& os, const QueueItem<T>& qi) {
    os <<qi.info;// amicizia con QueueItem
        // Notare che si richiede operator<< per il tipo T
    return os;
}
```

Dobbiamo quindi inoltre dichiarare la funzione esterna `operator<<` di `QueueItem` come amica associata della classe `QueueItem`.

```
template <class T>
class QueueItem {
    friend ostream& operator<< <T>(ostream&, const QueueItem<T>&);
    ...
};
```

Si noti che il fatto che `operator<<` di `QueueItem` utilizza `operator<<` per il parametro di tipo `T` introduce una sottile dipendenza di tipo dall'istanza di `Queue`. In effetti, se vogliamo poter stampare una coda appartenente a qualche istanza `Queue<Type>`, bisogna accertarsi che l'operatore di output sia definito sul tipo `Type` usato per istanziare il template. In caso contrario il compilatore segnala un errore quando deve istanziare il template.

4.3 Template di classe

dell'operatore di output di `Queue<Type>`, ossia dove esso viene effettivamente usato. Ad ogni modo, il template di classe `Queue` può essere istanziato con un tipo per il quale non sia definito `operator<<` purché non si cerchi di stampare un oggetto di tale istanza.

4.3.4 Membri statici in template di classe

Naturalmente anche in un template di classe possono essere dichiarati campi dati e metodi statici. In tal caso ogni istanza del template di classe ha dei propri campi dati e metodi statici, distinti da quelli delle altre istanze della classe. Consideriamo il seguente esempio. Vogliamo aggiungere al template di classe `Queue` un campo dati statico intero che funzioni da contatore globale degli oggetti `QueueItem` presenti nelle liste di tutti gli oggetti di una certa istanza di `Queue`.

```
template <class T>
class Queue {
    private:
        static int contatore;
    ...
};
```

Pertanto, ad ogni istanza di `Queue` è associato un campo statico `contatore` diverso. L'inizializzazione di un campo dati statico di un template di classe naturalmente deve essere esterna alla classe e deve rispettare la seguente sintassi:

```
template <class T>
int Queue<T>::contatore = 0;
```

Un campo dati statico è istanziato e quindi inizializzato dalla definizione del template soltanto se viene effettivamente usato. La mera definizione di un campo dati statico non provoca allocazione di memoria. Ad esempio, l'inizializzazione precedente specifica al compilatore come inizializzare il campo dati statico `contatore` associato ad un'istanza del template di classe `Queue` se tale istanza verrà creata e se il campo `contatore` verrà usato.

Esempio 4.3.3. Il seguente programma compila e la sua esecuzione provoca le stampe riportate.

```
template<int I> // parametro valore
class C {
    static int numero;
public:
    C();
    void stampa_numero();
};
```

```
// inizializzazione parametrica del campo dati statico
template<int I>
int C<I>::numero = I;

template<int I>
C<I>::C() { numero++; }

template<int I>
void C<I>::stampa_numero()
{ cout << "Valore statico: " << numero << endl; }

int main()
{
    C<1> uno; C<2> due_a, due_b;
    uno.stampa_numero(); // stampa: 2
    due_a.stampa_numero(); // stampa: 4
    due_b.stampa_numero(); // stampa: 4
}
```

Esercizio 4.3.4. Il seguente programma compila. Quali stampe provoca la sua esecuzione?

```
class A {
public:
    A(int x=0) { cout << x << "A() "; }
};

template<class T>
class C {
public:
    static A s;
};

template<class T>
A C<T>::s=A();

int main() {
    C<double> c;
    C<int> d;
    C<int>::s = A(2);
}
```

4.3.5 Template di classe annidati

Sappiamo che in una classe possono essere dichiarate altre classi annidate, sia pubbliche che private. All'interno di un template di classe possono essere dichiarati altri template

4.3 Template di classe

di classe annidati, sia associati che non associati. Ad esempio, per impedire all'utente del template di classe Queue di utilizzare direttamente il template di classe QueueItem noi abbiamo dichiarato privati tutti i membri di QueueItem incluso il costruttore e quindi abbiamo dichiarato Queue classe amica associata di QueueItem. Un modo migliore per permettere l'uso di QueueItem soltanto alla classe Queue è quello di annidare nella parte privata della definizione del template di classe Queue la definizione del template di classe QueueItem.

```
template <class T>
class Queue {
private:
    // template implicito 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

Il C++ standard prevede che l'uso di tipi e di template di classe o di funzione che dipendono da un parametro di tipo debba essere disambiguato tramite le parole chiave typename (per i tipi) e template (per i template sia di classe che di funzione). In altri termini non è permesso l'uso implicito di tipi e template che dipendono da parametri di tipo.

Supponiamo ad esempio che un template di classe contenga classi annidate, template di classi annidate e template di funzioni come segue:

```
template <class T> class C {
public:

    class D {                                // classe annidata
public:
    T x;
};

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

4 TEMPLATE

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

Allora l'uso dei nomi D, E e fun2 che dipendono da un parametro di tipo, ad esempio nel seguente template di funzione templateFun, deve essere disambiguato come segue:

```
template <class T>
void templateFun(typename C<T>::D d) {
    // C<T>::D è un uso di un tipo che dipende dal parametro T
    typename C<T>::D d2 = d;

    // (1) E<int> è un uso del template di classe annidata
    //      che dipende dal parametro T
    // (2) C<T>::E<int> è un uso di un tipo
    //      che dipende dal parametro T
    typename C<T>::template E<int> e;
    e.fun1();

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

Aderendo allo standard C++, gli usi impliciti di tipi e template non sono permessi dalla versione 3.4 del compilatore g++ (nella versione 3.3 g++ genera dei warning, mentre sino alla versione 3.2 g++ era tollerante).

4.4 Esempio di template di classe: alberi binari di ricerca

Come noto, gli *alberi binari di ricerca* sono strutture dati dinamiche che analogamente alle liste permettono di aggiungere e togliere un elemento da una collezione. Rispetto alle liste hanno il vantaggio di permettere la ricerca efficiente di un elemento. Essi permettono inoltre di trovare efficientemente il minimo, il massimo, il precedente e il successivo.

Iniziamo con un breve richiamo delle notazioni e della terminologia usati per gli alberi binari.

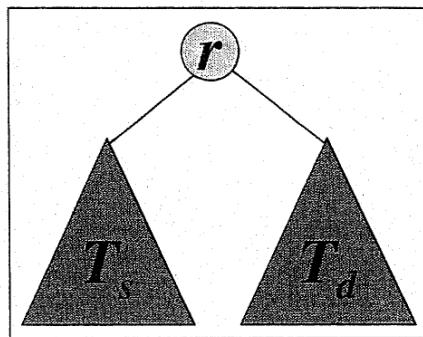
L'insieme $\text{Tree}(N)$ degli alberi binari sull'insieme N dei valori contenuti nei nodi è ricorsivamente definito nel seguente modo:

Base: L'insieme vuoto \emptyset è un albero binario, cioè $\emptyset \in \text{Tree}(N)$;

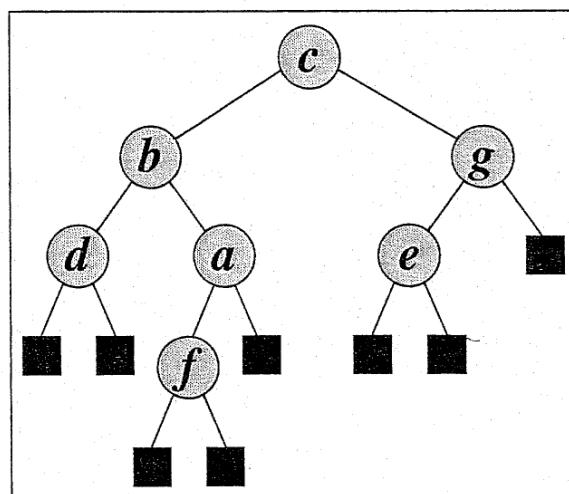
4.4 Esempio di template di classe: alberi binari di ricerca

Induzione: Se $T_s, T_d \in \text{Tree}(N)$ sono alberi binari e $r \in N$ allora la terna ordinata (r, T_s, T_d) è un albero binario, cioè $(r, T_s, T_d) \in \text{Tree}(N)$.

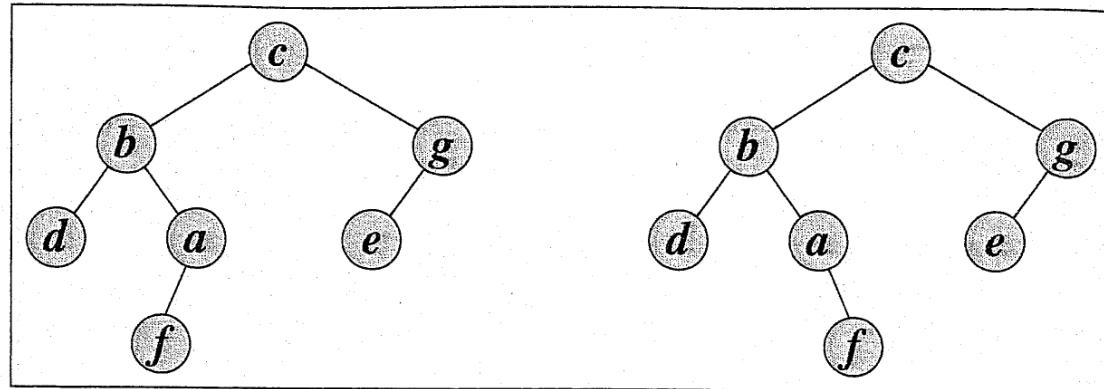
Quando T è l'insieme vuoto diciamo che esso è l'*albero vuoto*. Nell'albero binario $T = (r, T_s, T_d)$ il primo elemento r della terna è la *radice* dell'albero T , il secondo elemento T_s è il *sottoalbero sinistro* di T ed il terzo elemento T_d è il *sottoalbero destro* di T . Rappresentiamo graficamente l'albero vuoto \emptyset con un quadratino nero ■. Invece l'albero non vuoto $T = (r, T_s, T_d)$ si rappresenta graficamente con un nodo etichettato r e sotto di esso, ricorsivamente, le due rappresentazioni dei sottoalberi T_s e T_d , con T_s alla sinistra di T_d .



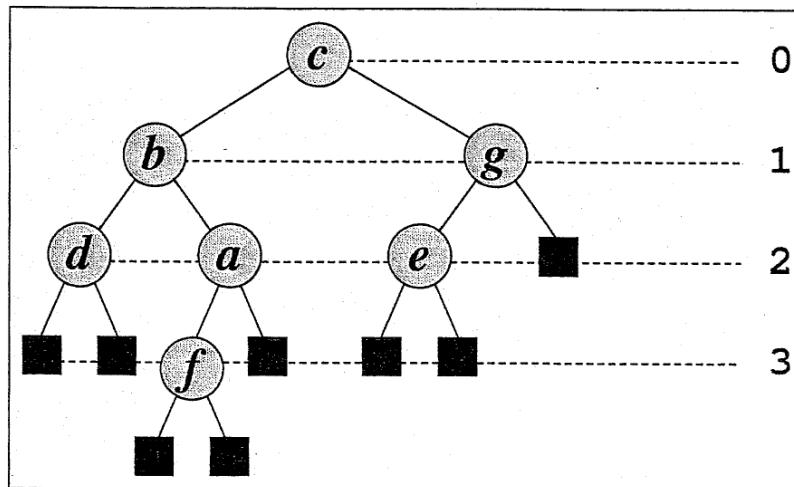
Ad esempio, l'albero $(c, (b, (d, \emptyset, \emptyset), (a, (f, \emptyset, \emptyset), \emptyset)), (g, (e, \emptyset, \emptyset), \emptyset)) \in \text{Tree}(N)$, con $N = \{a, b, c, d, e, f, g\}$, si rappresenta graficamente nel seguente modo:



Spesso la rappresentazione dei sottoalberi vuoti è omessa. In tal caso occorre prestare attenzione a distinguere tra figlio sinistro e destro nel caso in cui un nodo abbia un solo figlio. Ad esempio i due alberi binari seguenti sono diversi:



In un albero (r, T_s, T_d) se il sottoalbero sinistro T_s non è vuoto allora la sua radice r_s si dice *figlio sinistro* di r (e quindi r sarà il *padre* di r_s). Se T_s è vuoto diciamo che r non ha figlio sinistro. Analogamente per T_d . I nodi che non hanno né figlio sinistro né figlio destro si dicono *foglie* dell'albero. I *discendenti* di un nodo sono i figli e tutti i discendenti dei figli. Gli *ascendenti* (o *antenati*) di un nodo sono il padre e tutti gli ascendenti del padre. La *profondità* di un nodo n in un albero T è la lunghezza del cammino dalla radice dell'albero al nodo n . Ad esempio i nodi dell'albero $T = (c, (b, (d, \emptyset, \emptyset), (a, (f, \emptyset, \emptyset), \emptyset)), (g, (e, \emptyset, \emptyset), \emptyset))$ hanno le seguenti profondità:



L'*altezza* di un albero binario è la profondità massima delle foglie dell'albero. Ad esempio, il precedente albero ha altezza tre.

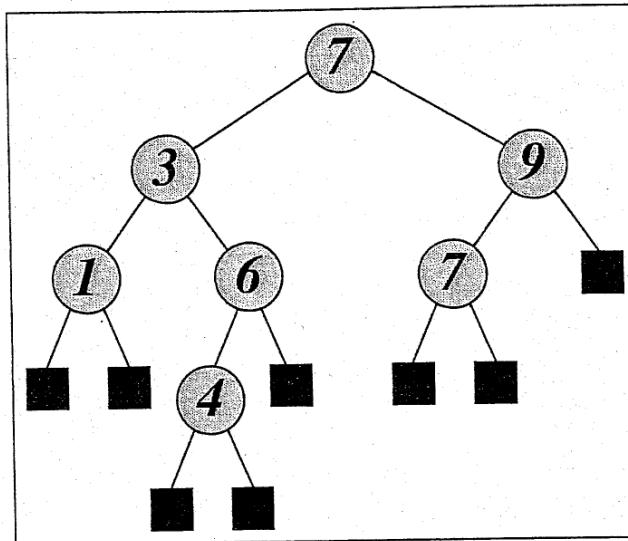
Useremo l'abituale rappresentazione ricorsiva per i nodi di un albero binario: una classe *nodo* avente un campo *info* in cui viene memorizzato il valore nel nodo e tre campi dati puntatore *padre*, *sinistro* e *destro* che puntano, rispettivamente, al *nodo padre*, al *nodo figlio sinistro* ed al *nodo figlio destro*.

Alberi binari di ricerca. Supponiamo che l'insieme N sia ordinato. Un albero binario di ricerca $T \in \text{Tree}(N)$ è un albero binario in cui il valore di ogni nodo di T è maggiore

4.4 Esempio di template di classe: alberi binari di ricerca

dei valori dei nodi del suo sottoalbero sinistro e minore o uguale dei valori dei nodi del suo sottoalbero destro.

Ad esempio, l'albero binario della seguente figura in cui i valori dei nodi sono dei numeri interi è un albero binario di ricerca.



Per rappresentare gli alberi binari di ricerca useremo un template di classe `AlbBinRic<T>` parametrico rispetto al tipo `T` dei valori dei nodi. I nodi saranno rappresentati con il template di classe `Nodo<T>`. Per ogni istanza del template di classe `AlbBinRic` ci sarà la possibilità di creare, modificare e accedere ai campi dei nodi appartenenti all'istanza associata del template di classe `Nodo`. D'altra parte l'operazione di ricerca in un albero binario di ricerca dovrà ritornare un puntatore al nodo trovato. Deve quindi essere possibile usare puntatori ai nodi anche all'esterno della classe. Una soluzione consiste nel dichiarare privati tutti i campi dati e metodi del template di classe `Nodo`, compresi i costruttori, e dichiarare quindi friend le istanze associate del template di classe `AlbBinRic`. Definiremo quindi i template di classe `Nodo<T>` e `AlbBinRic<T>` come segue.

```
// dichiarazione incompleta di AlbBinRic
template <class T> class Nodo;

// dichiarazione di operator<<
template <class T>
ostream& operator<< (ostream&, Nodo<T>*);

template <class T>
class Nodo {
    friend class AlbBinRic<T>; // classe friend associata
    // funzione friend associata
    friend ostream& operator<< <T>(ostream&, Nodo<T>*);
private:
    T info;
```

4 TEMPLATE

```
// notare il costruttore privato
Nodo *sinistro, *destro, *padre;
Nodo(const T& v, Nodo* p=0, Nodo* s=0, Nodo* d=0) :
    info(v), padre(p), sinistro(s), destro(d) {}
};

template <class T>
ostream& operator<< (ostream& os, Nodo<T>* p) {
    if (!p) os << '@'; // caso base
    else os << "(" << p->info << "," << p->sinistro
        << "," << p->destro << ")"; // passo ricorsivo
    return os;
}
```

```
// dichiarazione di operator<<
template <class T>
ostream& operator<< (ostream&, AlbBinRic<T>&);

template <class T>
class AlbBinRic {
    friend ostream& operator<< <T>(ostream&, const AlbBinRic<T>&);
public:
    AlbBinRic() : radice(0) {}
    Nodo<T>* Find(T) const;
    Nodo<T>* Minimo() const;
    Nodo<T>* Massimo() const;
    Nodo<T>* Succ(Nodo<T>*) const;
    Nodo<T>* Pred(Nodo<T>*) const;
    void Insert(T);
    static T Valore(Nodo<T>* p) { return p->info; }
private:
    Nodo<T>* radice; // puntatore alla radice
    // metodi privati di utilità
    static Nodo<T>* FindRic(Nodo<T>*, T);
    static Nodo<T>* MinimoRic(Nodo<T>*);
    static Nodo<T>* MassimoRic(Nodo<T>*);
    static void InsertRic(Nodo<T>*, T);
};
```

Passiamo ad esaminare le definizioni dei metodi del template di classe `AlbBinRic<T>`. Gli algoritmi sugli alberi binari di ricerca sono approfonditamente studiati in un corso di Algoritmi e Strutture Dati.

L'algoritmo di ricerca sfrutta la definizione di albero binario di ricerca.

```
template <class T>
```

4.4 Esempio di template di classe: alberi binari di ricerca

```
Nodo<T>* AlbBinRic<T>::Find(T v) const {
    return FindRic(radice,v);
}

template <class T>
Nodo<T>* AlbBinRic<T>::FindRic(Nodo<T>* x, T v) {
    if (!x) return x; // caso base: albero vuoto
    if(x->info == v) return x; // caso base: v è nella radice
    if (v < x->info) // cerca ricorsivamente nel sottoalbero sx
        return FindRic(x->sinistro,v);
    else // cerca ricorsivamente nel sottoalbero dx
        return FindRic(x->destro,v);
}
```

L'idea dell'algoritmo che calcola il valore massimo di un albero binario di ricerca è la seguente: il valore massimo è memorizzato nella "foglia più a destra".

```
template <class T>
Nodo<T>* AlbBinRic<T>::Massimo() const {
    if (!radice) return 0;
    return MassimoRic(radice);
}

template <class T>
Nodo<T>* AlbBinRic<T>::MassimoRic(Nodo<T>* x) {
    if (!(x->destro)) return x;
    else return MassimoRic(x->destro);
}
```

Naturalmente, il minimo è duale al massimo.

```
template <class T>
Nodo<T>* AlbBinRic<T>::Minimo() const {
    if (!radice) return 0;
    return MinimoRic(radice);
}

template <class T>
Nodo<T>* AlbBinRic<T>::MinimoRic(Nodo<T>* x) {
    if (!(x->sinistro)) return x;
    else return MinimoRic(x->sinistro);
}
```

L'idea dell'algoritmo che calcola il successore del valore di un nodo puntato da x in un albero T è la seguente: se (il nodo puntato da) x ha sottoalbero destro x_d allora il

4 TEMPLATE

successore di x è il minimo di x_d ; se il sottoalbero destro di x è vuoto allora il successore di x è l'antenato di x più giovane di cui x è discendente sinistro

```
template <class T>
Nodo<T>* AlbBinRic<T>::Succ(Nodo<T>* x) const {
    if (!x) return 0;
    if (x->destro) return MinimoRic(x->destro);
    // caso x->destro == 0
    while (x->padre && x->padre->destro == x) x = x->padre;
    return x->padre;
}
```

La funzione Pred è duale alla precedente Succ.

```
template <class T>
Nodo<T>* AlbBinRic<T>::Pred(Nodo<T>* x) const {
    if (!x) return 0;
    if (x->sinistro) return MassimoRic(x->sinistro);
    while (x->padre && x->padre->sinistro == x) x = x->padre;
    return x->padre;
}
```

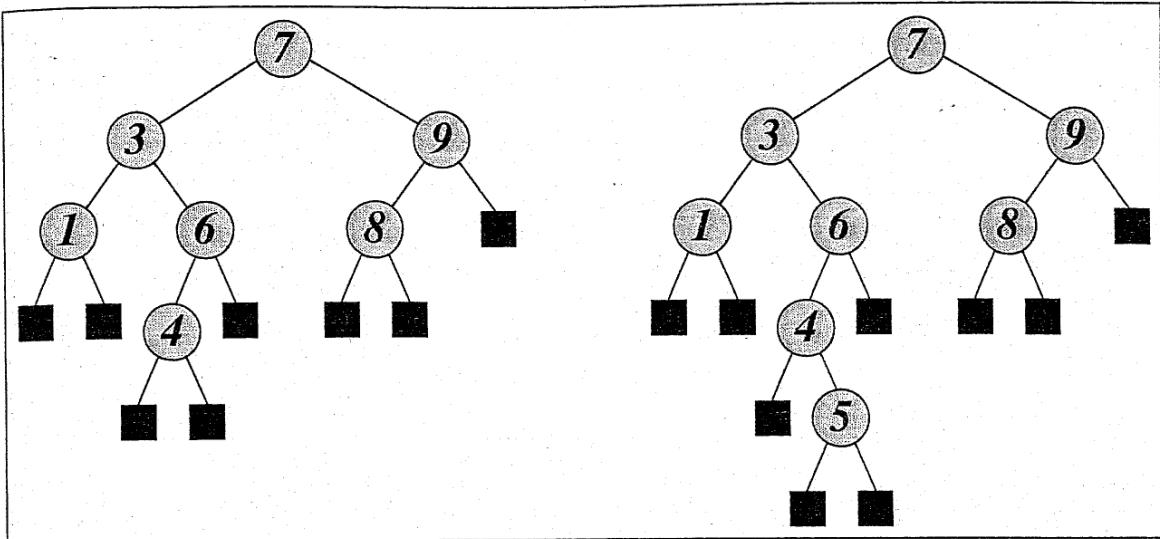
L'idea dell'algoritmo per inserire un valore v nell'albero T è la seguente: se T è vuoto inserisco v come radice, altrimenti se $T = (r, T_s, T_d)$ e $v < r$ allora inserisco v in T_s , mentre se $v \geq r$ inserisco v in T_d .

```
template <class T>
void AlbBinRic<T>::Insert(T v) {
    if (!radice) radice = new Nodo<T>(v);
    else InsertRic(radice, v);
}

template <class T>
void AlbBinRic<T>::InsertRic(Nodo<T>* x, T v) {
    if (v < x->info)
        if (x->sinistro == 0) x->sinistro = new Nodo<T>(v, x);
        else InsertRic(x->sinistro, v);
    else
        if (x->destro == 0) x->destro = new Nodo<T>(v, x);
        else InsertRic(x->destro, v);
}
```

Ad esempio, se t è un puntatore alla radice dell'albero di destra della seguente figura allora $t.insert(5)$ punterà alla radice dell'albero di sinistra.

4.4 Esempio di template di classe: alberi binari di ricerca



Infine, l'operatore di output semplicemente richiama l'operatore di output per puntatori a Nodo sulla radice dell'albero.

```
template <class T>
ostream& operator<<(ostream& os, const AlbBinRic<T>& A) {
    os << A.radice << endl; // stampa di Nodo<T>*
    return os;
}
```

Il seguente esempio di main() istanzia il template AlbBinRic<int> e usa le varie funzionalità disponibili.

```
// file "main-abr.cpp"
#include<iostream>
#include "nodo.h"
#include "AlbBinRic.h"
using namespace std;

int main() {
    AlbBinRic<int> a;
    a.Insert(3); a.Insert(2); a.Insert(3);
    a.Insert(1); a.Insert(6); a.Insert(5);
    cout << a;

    cout << "minimo: " << AlbBinRic<int>::Valore(a.Minimo()) << endl;
        // minimo: 1
    cout << "massimo: " << AlbBinRic<int>::Valore(a.Massimo()) << endl;
        // massimo: 6

    cout << "successore minimo: "
```

4 TEMPLATE

```
<< AlbBinRic<int>::Valore(a.Succ(a.Minimo())) << endl;
// successore minimo: 2
cout << "predecessore massimo: "
<< AlbBinRic<int>::Valore(a.Pred(a.Massimo())) << endl;
// predecessore massimo: 5

if(a.Find(2)) cout << "Il valore " << 2 << " è presente\n";
// Il valore 2 è presente
}
```

Esercizio 4.4.1. Definire il metodo di cancellazione:

```
template<class T> void Delete(T);
```

che rimuove un nodo contenente un dato valore, se un tale nodo esiste. Si tratta della funzione più complicata, studiata dettagliatamente in un corso di Algoritmi e Strutture Dati.

Capitolo 5

I Contenitori della Libreria STL

La Standard Template Library, STL come acronimo, è parte della libreria standard del C++ ed è una libreria di classi contenitore, di classi iteratore e di algoritmi su tali classi. STL è una libreria generica, cioè i suoi membri, classi e funzioni, sono dei template. Illustreremo sinteticamente le principali classi contenitore e gli iteratori della libreria STL, mentre non saranno trattati gli algoritmi generici. Una valida e consigliata documentazione delle componenti della libreria STL è la “Standard Template Library’s Programming Guide” della SGI, Silicon Graphics Inc., liberamente disponibile on-line. libstdc++ è l’implementazione GNU della libreria STL per il compilatore g++, che aderisce quindi al C++ standard. Esistono varie altre implementazioni di STL, quale ad esempio la stdcxx della Apache Software Foundation che pure aderisce alla definizione del C++ standard.

5.1 Classi contenitore

In generale, una istanza `c` di una classe contenitore `C` permette di memorizzare altri oggetti, ossia gli elementi di `c`. La classe contenitore `C` mette a disposizione vari metodi per accedere agli elementi di `c`. Inoltre, ogni classe contenitore `C` ha una classe interna iteratore `i` cui oggetti possono essere usati per iterare sugli elementi delle istanze di `C`.

Un contenitore è un template di classe `C` parametrico sul tipo `T` degli elementi contenuti (alcuni contenitori come `map` e `multimap` sono invece parametrici su due tipi). In realtà, il template `C` ha un ulteriore parametro di tipo `Alloc` per cui `C` dichiara un parametro di tipo di default: non ci occuperemo del significato del parametro `Alloc`, menzioniamo solamente che si tratta di un cosiddetto “allocatore”, che si occupa di gestire la memoria degli oggetti contenitore. Inoltre, i contenitori associativi come `map`, `multimap`, `set` e `multiset` hanno dei parametri addizionali aventi dei valori di default che ci permettono di ignorarli in questa trattazione.

Ogni classe contenitore `C` ha tra i suoi membri i seguenti tipi:

5 I CONTENITORI DELLA LIBRERIA STL

`C::value_type`: Si tratta del tipo degli oggetti memorizzati in un oggetto contenitore. `value_type` è quindi l'istanziazione del parametro di tipo `T` del template `C<T>`. Il tipo `value_type` deve essere assegnabile ma non deve necessariamente fornire un costruttore di default.

`C::iterator`: È il tipo iteratore usato per iterare sugli elementi di un contenitore. Il tipo `iterator` fornisce l'operatore di dereferenziazione `operator*()` che ritorna un riferimento a `value_type`, cioè se `it` è un iteratore allora `*it` ha tipo `value_type&`.

`C::const_iterator`: È il tipo iteratore costante che può essere usato per accedere agli elementi di un contenitore ma non permette di modificarli. Tipicamente gli iteratori costanti vengono usati per accedere agli elementi di contenitori costanti.

`C::size_type`: È un tipo integrale senza segno usato per rappresentare la distanza tra due iteratori.

Ogni classe contenitore `C` fornisce i seguenti costruttori, metodi e operatori:

`C(const C&)`: ridefinizione del costruttore di copia. Se `c` è costruito come copia di `b` allora `c` contiene una copia di ognuno degli elementi di `b`.

`C& operator=(const C&)`: ridefinizione dell'assegnazione. Dopo una assegnazione `c=b`, il contenitore `c` contiene una copia di ognuno degli elementi di `b`.

`~C()`: ridefinizione del distruttore. Nella distruzione di un oggetto contenitore `c` ogni elemento di `c` è distrutto e la memoria allocata per ogni elemento (se ve ne è) viene deallocata.

`size_type size()`: una invocazione `c.size()` ritorna la dimensione del contenitore `c`, cioè il numero di elementi contenuti in `c`, ed è quindi un intero ≥ 0 . Se `c` è vuoto allora `c.size()` ritorna 0.

`bool empty()`: `c.empty()` è equivalente al test `c.size() == 0`.

`size_type max_size()`: `c.max_size()` ritorna la massima dimensione che il contenitore `c` può avere.

Le classi contenitore che prenderemo in esame sono quelle più diffusamente utilizzate: `vector`, `list`, `slist`, `deque`, `set`, `map`, `multiset`, `multimap`. Per queste classi contenitore gli elementi sono memorizzati in un ordine ben definito che non cambia da una iterazione all'altra: ciò permette la definizione dell'operatore di uguaglianza ordinatamente elemento-per-elemento (sempre che l'istanziazione del parametro di tipo `T` renda disponibile l'operatore di uguaglianza) e dell'ordinamento lessicografico. Quindi, per tutte queste classi contenitore sono a disposizione le seguenti funzionalità:

`operator== : b==c` ritorna `true` se `b.size() == c.size()` e se ogni elemento di `b` è uguale, per una invocazione di `operator==` del tipo sottostante, al corrispondente (secondo l'ordine di memorizzazione degli elementi) elemento di `c`. Altrimenti, ritorna `false`. Naturalmente è disponibile anche l'operatore di disegualanza `operator!=`.

`operator< : b < c` ritorna `true` se la sequenza di elementi di `b` è minore per l'ordine lessicografico, determinato dall'ordine di memorizzazione degli elementi in un contenitore, della sequenza di elementi di `c`. Sono inoltre disponibili `operator<=`, `operator>`, `operator>=`.

5.2 Iteratori

Abbiamo già visto come il concetto di iteratore possa essere concepito come una generalizzazione del concetto di puntatore: gli iteratori sono oggetti che “puntano” ad altri oggetti. Come suggerisce il nome, gli iteratori sono spesso usati per iterare su un intervallo di oggetti: se un iteratore `it` punta ad un elemento in un intervallo allora è possibile incrementare `it` in modo che esso punti all'elemento successivo nell'intervallo. La dereferenziazione di un iteratore `it` ritorna l'oggetto puntato da `it`. Vi è un particolare iteratore che non punta ad alcun oggetto: un iteratore `it` viene detto “past-the-end” (“successivo alla fine”) se `it` punta oltre l'ultimo elemento di un contenitore; possiamo pensare che punti alla “posizione successiva” a quella dell'ultimo elemento di un contenitore. Gli iteratori past-the-end non sono dereferenziabili. Un iteratore su un contenitore viene detto valido se è dereferenziabile oppure se è l'iteratore past-the-end.

Ogni classe contenitore `C` ha a disposizione i seguenti due metodi che ritornano degli iteratori:

`C::iterator begin() : c.begin()` ritorna un iteratore che punta al primo elemento di `c`. Se `c` non contiene elementi, cioè `c.size() == 0`, allora `c.begin()` è l'iteratore past-the-end di `c`. Se il contenitore `c` è costante allora `c.begin()` ritorna un iteratore costante, cioè di tipo `C::const_iterator`.

`C::iterator end() : c.end()` è l'iteratore past-the-end di `c`. Se il contenitore `c` è costante allora `c.end()` ritorna un iteratore costante.

Abbiamo già visto che su ogni tipo `iterator` è sempre disponibile l'operatore di dereferenziazione `operator*()`. Sono inoltre disponibili i seguenti operatori di incremento e decremento:

`C::iterator& operator++() : se it è un iteratore dereferenziabile allora l'incremento prefisso ++it sposta l'iteratore it all'elemento successivo; in particolare, se it puntava all'ultimo elemento del contenitore allora dopo l'incremento it diventa l'iteratore past-the-end.`

5 I CONTENITORI DELLA LIBRERIA STL

`C::iterator operator++(int)` : è l'operatore di incremento postfisso con l'usuale comportamento.

`C::iterator& operator--()` : se `it` è un iteratore valido allora il decremento pre-fisso `--it` sposta l'iteratore `it` sull'elemento precedente; in particolare, se `it` era l'iteratore `past-the-end` allora dopo il decremento `it` punta all'ultimo elemento del contenitore.

`C::iterator operator--(int)` : è l'operatore di decremento postfisso con l'usuale comportamento.

Gli iteratori per i contenitori `vector` e `deque` permettono di avanzare e di retrocedere di un numero arbitrario di elementi. Sono inoltre disponibili gli operatori di confronto per questi iteratori. Ci si riferisce in questo ad *iteratori ad accesso casuale*. In quanto segue assumiamo che `n` sia una variabile intera e che `it`, `it1` e `it2` siano iteratori ad accesso casuale.

`it += n` : Per $n > 0$, è equivalente ad eseguire n volte l'incremento `++it`. Per $n < 0$, è equivalente ad eseguire n volte il decremento `--it`. Se $n == 0$ allora non provoca alcun effetto.

`it + n` : se `temp = it` allora ritorna `temp += n`.

`it -= n` e `it - n` : sono i decrementi duali ai precedenti incrementi.

`it[n]` : è equivalente alla dereferenziazione `*(it+n)`.

`it1 < it2` : ritorna `true` se `it1` punta ad un elemento che precede nel contenitore l'elemento puntato da `it2`, altrimenti ritorna `false`. `it1` e `it2` possono anche essere gli iteratori `past-the-end`. Sono inoltre disponibili con l'usuale significato gli operatori relazionali `<=`, `>` e `>=`.

5.3 Sequenze

Un *contenitore sequenza* è caratterizzato dalla proprietà che i suoi elementi sono memorizzati secondo un ordine lineare stretto determinato dall'utente del contenitore. Un contenitore sequenza `c` supporta l'inserimento e la rimozione di elementi in qualsiasi posizione di `c` puntata da qualche iteratore. Le classi `vector`, `list`, `slist` e `deque` sono dei contenitori sequenza.

Un contenitore sequenza `C` fornisce le seguenti funzionalità:

`C(C::size_type n, C::value_type t)` : `c(n, t)` costruisce il contenitore sequenza `c` contenente `n` copie dell'elemento `t`.

5.3 Sequenze

`c(C::size_type n)` : `c(n)` costruisce il contenitore sequenza `c` contenente `n` elementi inizializzati al valore di `default_value_type()` (quindi, il tipo attuale `T` del contenitore `C<T>` dovrà mettere a disposizione il costruttore di `default T()`).

`C::iterator insert(C::iterator it, C::value_type t)` : se `c` è un contenitore e se `it` è un iteratore valido di `c` allora `c.insert(it, t)` inserisce l'elemento `t` nella sequenza `c` immediatamente prima dell'elemento puntato da `it` e ritorna un iteratore che punta all'elemento appena inserito. Quindi `c.insert(c.begin(), t)` inserisce `t` all'inizio di `c` mentre `c.insert(c.end(), t)` inserisce `t` in coda a `c`.

`void insert(C::iterator it, C::size_type n, C::value_type t)` : se `c` è un contenitore e se `it` è un iteratore valido di `c` allora `c.insert(it, n, t)` inserisce `n` copie dell'elemento `t` nella sequenza `c` immediatamente prima dell'elemento puntato da `it`.

`C::iterator erase(C::iterator it)` : se `c` è un contenitore e se `it` è un iteratore dereferenziabile allora `c.erase(it)` distrugge l'elemento puntato da `it`, lo rimuove dalla sequenza `c` e ritorna un iteratore che punta all'elemento che seguiva immediatamente l'elemento rimosso (che sarà l'iteratore `past-the-end` se l'elemento rimosso era l'ultimo).

`C::iterator erase(C::iterator it1, C::iterator it2)` : se `c` è un contenitore e se `it1, it2` sono iteratori tali che l'intervallo (chiuso a sinistra ed aperto a destra) `[it1, it2)` è valido in `c` allora `c.erase(it1, it2)` distrugge gli elementi nell'intervallo `[it1, it2)`, li rimuove dalla sequenza `c` e ritorna un iteratore che punta all'elemento che seguiva immediatamente l'intervallo di elementi rimosso.

`void clear()` : `c.clear()` è equivalente a `c.erase(c.begin(), c.end())`.

Per i contenitori `vector`, `list` e `deque` sono a disposizione i seguenti due metodi di inserimento e rimozione in coda, che sono usati frequentemente:

`void push_back(C::value_type t)` : una invocazione `c.push_back(t)` è equivalente all'invocazione `c.insert(c.end(), t)` e quindi inserisce in coda a `c` l'elemento `t`.

`void pop_back()` : una invocazione `c.pop_back()` è equivalente alla seguente invocazione: `c.erase(--c.end())`, ovvero rimuove da `c` l'elemento in coda a `c`.

Infine, per i contenitori sequenza `list` e `deque` sono a disposizione i seguenti due metodi di inserimento e rimozione in testa:

`void push_front(C::value_type t)` : una invocazione `c.push_front(t)` è equivalente all'invocazione `c.insert(c.begin(), t)` e quindi inserisce in testa a `c` l'elemento `t`.

`void pop_front()` : una invocazione `c.pop_front()` è equivalente all'invocazione `c.erase(c.begin())`, ovvero rimuove l'elemento in testa a `c`.

5.3.1 Sequenze ad accesso casuale

Le classi `vector` e `deque` sono dei *contenitori ad accesso casuale*. Ciò significa che è disponibile l'operatore di indicizzazione `operator[]`: se `c` è un contenitore e `n` è un intero tale che $0 \leq n \leq c.size()$ allora `c[n]` ritorna l' n -esimo elemento di `c` (a partire dall'elemento iniziale di `c`). La caratteristica fondamentale dei contenitori ad accesso casuale è che un accesso `c[n]` mediante l'operatore di indicizzazione ha una complessità di tempo media (ammortizzata) costante.

5.4 Contenitori associativi

Un *contenitore associativo* permette un accesso efficiente agli elementi tramite delle *chiavi*. Fornisce funzionalità di inserimento e rimozione di elementi, ma diversamente da un contenitore sequenza non sono disponibili funzionalità per inserire un elemento in una posizione specifica. Come per tutti i contenitori, gli elementi in un oggetto `c` di un contenitore associativo `C` hanno tipo `C::value_type`. Inoltre, ogni elemento in `c` ha una chiave di tipo `C::key_type`. Nei contenitori associativi `set` e `multiset` il tipo `key_type` coincide con il tipo `value_type`, cioè gli elementi del contenitore sono anche le chiavi di accesso. Nei contenitori associativi `map` e `multimap` la chiave d'accesso è una componente dell'elemento memorizzato nell'oggetto contenitore. Poiché la memorizzazione degli elementi nel contenitore è determinata dalle loro chiavi e le chiavi devono essere non modificabili, una caratteristica importante dei contenitori associativi consiste nel fatto che il tipo `value_type` di un contenitore associativo non è modificabile e quindi assegnabile. Quindi, se `it` è un iteratore dereferenziabile su un contenitore associativo `c` e `t` è di tipo `C::value_type` allora l'assegnazione `*it = t;` è illegale (cioè non compila). I contenitori associativi `set` e `map` sono detti *unici* perché non possono esistere due elementi con la stessa chiave. D'altra parte i contenitori `multiset` e `multimap` sono detti *multipli* perché più elementi con la stessa chiave sono invece permessi.

Ogni contenitore associativo `C` offre le seguenti funzionalità:

`C::size_type count(C::key_type)` : una invocazione `c.count(k)` ritorna il numero di elementi di `c` con chiave `k`.

`C::size_type erase(C::key_type)` : una invocazione `c.erase(k)` distrugge tutti gli elementi di `c` con chiave `k` e li rimuove da `c`. Inoltre ritorna il numero di elementi rimossi da `c`, cioè ritorna `c.count(k)`.

`void erase(C::iterator)` : se `it` è un iteratore dereferenziabile su `c` allora l'invocazione `c.erase(it)` distrugge l'elemento puntato da `it` e lo rimuove da `c`.

`void erase(C::iterator, C::iterator)` : l'invocazione `c.erase(it1, it2)` distrugge gli elementi nell'intervallo `[it1, it2]` e li rimuove da `c`.

5.5 Contenitore vector

`void clear(): c.clear()` è equivalente a `c.erase(c.begin(), c.end())`, e quindi distrugge e rimuove tutti gli elementi in `c`.

`C::iterator find(C::key_type)`: l'invocazione `c.find(k)` ritorna un iteratore che punta ad un elemento con chiave `k` se questo esiste altrimenti ritorna l'iteratore past-the-end `c.end()`. Si noti che se `c` contiene più di un elemento con chiave `k` allora in generale non è possibile sapere a quale specifico elemento con chiave `k` punta l'iteratore ritorna da `c.find(k)`.

Possiamo infine menzionare il fatto che i contenitori associativi qui considerati, cioè `set`, `multiset`, `map` e `multimap`, sono *ordinati*, cioè il tipo `key_type` delle chiavi di accesso deve rendere disponibile l'operatore di confronto `operator<`. La relazione di ordinamento su `key_type` viene usata per determinare l'ordine di memorizzazione degli elementi nel contenitore.

5.5 Contenitore vector

Un `vector v` è un contenitore sequenza che supporta l'accesso casuale agli elementi, la rimozione e l'inserimento di elementi in coda a `v` in tempo costante, e la rimozione e l'inserimento di elementi in testa o alla metà di `v` in tempo lineare sulla dimensione di `v` (cioè `v.size()`). Essenzialmente un `vector` è implementato tramite un array dinamico che viene ridimensionato (solitamente la lunghezza viene raddoppiata ma ciò può variare da compilatore a compilatore) all'occorrenza. Quindi il numero di elementi in un `vector` può variare dinamicamente e la gestione della memoria è automatica e quindi trasparente al programmatore. La classe `vector` è il più semplice contenitore della libreria STL (è il contenitore "di base" di STL) e per molti contesti applicativi anche il più efficiente. Oltre alle funzionalità già descritte, `vector` supporta il metodo

`vector::size_type capacity() const`

tal che una invocazione `v.capacity()` ritorna il numero di elementi che `v` può contenere senza richiedere una nuova allocazione di memoria (possiamo pensare che è la lunghezza dell'array dinamico usato per memorizzare gli elementi di `v`). Quindi, l'espressione booleana

`v.size() <= v.capacity()`

ritorna sempre `true`.

Per poter dichiarare un `vector` sono necessarie una direttiva di inclusione del file header associato:

```
#include <vector>
```

ed una dichiarazione/direttiva d'uso del namespace `std` che ne contiene la definizione. Inoltre bisogna specificare il parametro di tipo attuale al template di classe `vector<T>` a cui sarà istanziato il tipo `vector::value_type`, cioè il tipo degli oggetti memorizzati

5 I CONTENITORI DELLA LIBRERIA STL

nel vector. Ricordiamo che il tipo `vector` rende disponibile l'accesso agli elementi tramite l'operatore di indicizzazione `operator []`, in cui l'indice parte da zero. Vediamo alcuni semplici esempi d'uso.

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

template <class T>
void stampa(const vector<T>& v) {
    for (int i = 0; i < v.size(); i++) cout << v[i] << endl;
}

int main() {
    const int n = 5;
    vector <int> iv(n);
    int ia[n] = {2,4,5,2,-2};
    for (int i = 0; i < n; i++) iv[i] = ia[i] + 1;
    vector<int> w = iv; // costruttore di copia
    vector<int> u(10,-2);
    u[0]=w[0];
    stampa(u);
}
```

Bisogna prestare attenzione al fatto che, diversamente dagli array, non è possibile in alcun modo inizializzare un `vector` con una data sequenza di valori.

```
int ia[6] = {-1,5,-7,0,12,3}; // Ok
vector <int> ivec(6) = {-1,5,-7,0,12,3}; // Illegale
```

Possiamo invece inizializzare un `vector` con un segmento di un array o di un `vector` tramite gli opportuni costruttori.

```
int ia[20];
vector<int> iv(ia,ia+6); // Ok
cout << iv.size() << endl; // size: 6
vector<int> iv2(iv.begin(),iv.end()-2); // Ok
cout << iv2.size() << endl; // size: 4
```

Il metodo `push_back()` di inserimento in coda è la funzionalità di inserimento più semplice e comunemente usata. Vediamo un esempio d'uso.

```
vector<string> sv;
string x;
while (cin >> x) sv.push_back(x);
```

5.6 Altri contenitori

```
cout << endl << "Abbiamo letto:" << endl;
for (int i = 0; i < sv.size(); i++) cout << sv[i] << endl;
```

Il precedente frammento di codice legge delle stringhe dallo standard input `cin`, separate da spazi, tabulazioni o CR (Carriage Return, cioè il carattere di fine linea che si inserisce tramite il tasto `<Enter>`) fino al carattere di EOF (End Of File), e le inserisce in coda al vector `sv`. È utile ricordare che quando `cin` è connesso alla tastiera, è possibile inviare il carattere di EOF da tastiera tramite una particolare combinazione di tasti, tipicamente `<Ctrl>+<d>`. In modo del tutto analogo possiamo leggere le stringhe di un file (`dati.txt` nel prossimo esempio) separate da spazi, tabulazioni o CR e memorizzarle in un vector (l'Input/Output sarà esaminato nel Capitolo 8). Questa versione del codice usa gli iteratori.

```
vector<string> sv; string x;
ifstream file("dati.txt",ios::in);
while (file >> x) sv.push_back(x);
cout << "Abbiamo letto:" << endl;
vector<string>::iterator it;
for (it = sv.begin(); it != sv.end(); it++) cout << *it << endl;
```

Sfruttando il fatto che i contenitori di STL sono dei template di classe è possibile definire dei template di funzioni che usano dei contenitori parametrici sul tipo, come nel seguente esempio:

```
template <class T>
void leggi_scrivi() {
    vector<T> v;
    vector<T>::iterator i;
    T x;
    while (cin >> x) v.push_back(x); // fino a EOF
    cout << endl << "Abbiamo letto:" << endl;
    for (i = v.begin(); i != v.end(); i++) cout << *i << endl;
}
```

Esercizio 5.5.1. Definire un template di funzione che estrae (e quindi anche rimuove) il minimo elemento da un insieme memorizzato in un vector usando solamente il metodo `pop_back()`. Si assuma che sul tipo di base del vector sia definito l'operatore di confronto `operator<` e l'ordine indotto sia totale.

5.6 Altri contenitori

Illustriamo succintamente le caratteristiche salienti degli altri contenitori.

5.6.1 list

Un oggetto di tipo `list` è implementato come una lista doppiamente collegata, cioè ogni nodo della lista memorizza un puntatore al nodo precedente ed al nodo successivo. Un oggetto `list` è un contenitore sequenza che supporta l'inserimento e la rimozione di elementi all'inizio, alla fine e alla metà della lista in tempo costante. Quindi, le operazioni di inserimento e rimozione di elementi in ogni posizione sono più efficienti che in un `vector` ma, d'altra parte, l'accesso agli elementi è meno efficiente che in un `vector` dal momento che sarà sempre necessario percorrere la lista. Menzioniamo inoltre che STL include anche il contenitore `slist` che implementa una lista singolarmente collegata, cioè una lista dove ogni nodo contiene un puntatore al nodo successivo ma non al precedente.

Sono interessanti le seguenti funzionalità rese disponibili da `list`:

`void merge(list<T>& x)`: se `l` e `x` sono oggetti distinti (cioè `&x != &l`) di `list<T>` e `T` è un tipo ordinato, ovvero è disponibile l'operatore relazionale `operator<` su `T`, allora l'invocazione `l.merge(x)` rimuove tutti gli elementi dalla lista `x` e li inserisce in ordine, secondo l'ordinamento `<`, nella lista `l`.

`void reverse()`: una invocazione `l.reverse()` rovescia l'ordine di memorizzazione degli elementi nella lista `l`.

`void sort()`: se `T` è un tipo ordinato, ovvero è disponibile l'operatore relazionale `operator<` su `T`, allora l'invocazione `l.sort()` ordina la lista `l` secondo l'ordinamento `<`.

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

int main() {
    list<int> L;
    L.push_back(0); L.push_front(0);
    L.insert(++L.begin(), 2); // inserisce prima del secondo elemento
    L.push_back(5); L.push_front(6);

    list<int>::const_iterator i;

    // stampa: 6 0 2 0 5
    for(i=L.begin(); i != L.end(); ++i) cout << *i << " ";
}
```

5.6.2 deque

Un contenitore `deque` è molto simile ad un `vector`: è un contenitore sequenza che supporta l'accesso casuale agli elementi (cioè l'operatore di indicizzazione `operator[]` in

5.6 Altri contenitori

tempo costante), l'inserimento e la rimozione in coda alla sequenza in tempo costante e l'inserimento e la rimozione nel mezzo della sequenza in tempo lineare. La principale differenza consiste nell'inserimento e rimozione di elementi all'inizio della sequenza che per un deque avviene in tempo costante. Questo dipende dal fatto che deque è implementato come una "double-ended queue" (questa struttura dati è studiata nel corso di Algoritmi e Strutture Dati). Inoltre, la classe deque non rende disponibile il metodo `capacity()`.

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

int main() {
    // costruisce un deque<int> vuoto e lo riempie con push_back
    deque<int> dq;
    for (int i=0; i<5; ++i) dq.push_back(i);

    // stampa: 01234
    for (int i=0; i<dq.size(); ++i) cout << dq.at(i);
    cout << endl;

    // inserimento in testa al deque con push_front
    for (int i=0; i<3; ++i) dq.push_front(8);

    // stampa: 88801234
    for (int i=0; i<dq.size(); ++i) cout << dq.at(i);
    cout << endl;

    // rimozione di primo e ultimo elemento con pop_front e pop_back
    dq.pop_front(); dq.pop_back();

    // stampa: 880123
    for (int i=0; i<dq.size(); ++i) cout << dq.at(i);
    cout << endl;
}
```

5.6.3 set e multiset

set è un contenitore associativo ordinato che memorizza oggetti di tipo `set::key_type`. Quindi, una istanziazione `set<K>` richiede che il tipo K sia ordinato (cioè sia disponibile `operator<`). Un set modella il concetto matematico di insieme e quindi accetta solamente una occorrenza per ogni valore nell'insieme. La chiave di accesso di un elemento di un set coincide con il valore stesso dell'elemento, cioè gli elementi contenuti in un set devono avere valori tutti distinti. Vediamo un esempio.

```
#include<set>

set<string> cod_istr;
cod_istr.insert("LOAD");
cod_istr.insert("STORE");
cod_istr.insert("STOP");
string x("BRANCH");

if (cod_istr.find(x) == cod_istr.end()) ... // x non è un codice
else ... // x è un codice

// oppure equivalentemente:
if (cod_istr.count(x)) ... // x è un codice
else ... // x non è un codice
```

Un multiset invece modella il concetto di multiinsieme, ovvero di insieme di elementi che può contenere occorrenze multiple di un elemento. Anche in questo la chiave è data dal valore dell'elemento. La differenza consiste nel fatto che un oggetto di multiset può contenere occorrenze multiple dello stesso elemento, ovvero più elementi con lo stesso valore.

5.6.4 map e multimap

Un oggetto *m* di tipo *map<Key, Data>* è un contenitore associativo ordinato che associa oggetti di tipo *Key* con oggetti di tipo *Data*. Il tipo *Key* delle chiavi deve supportare l'operatore di ordinamento *operator<*. Gli elementi del *map m* (cioè i valori di *map::value_type*) sono coppie chiave-valore (*k,v*). Un *map m* non può contenere due elementi con la stessa chiave, cioè se (*k₁,v₁*) e (*k₂,v₂*) sono due elementi di *m* allora *k₁ != k₂*. Con i *map* bisogna prestare attenzione al seguente punto: se *map<Key, Data> m* è un *map* e *k* è un valore del tipo chiave *Key* che non occorre in qualche elemento di *m* allora una assegnazione

Data x = m[k];

che tenta di recuperare il valore associato alla chiave *k* non presente in *m* aggiunge l'elemento (*k,Data()*) al *map m*. Vediamo un semplice esempio d'uso di un *map*.

```
#include<map>
#include<string>

map<string, int> tavola_etichette;
// tabella delle etichette di un programma: associa al nome
// dell'etichetta il numero d'ordine dell'istruzione
```

5.6 Altri contenitori

```
...
string x = "for", y = "if"; int n = 5, m = 6;
map<string,int>::value_type v(x,n);
map<string,int>::value_type w(y,m);
tavola_etichette.insert(v); tavola_etichette.insert(w);
int numIstr = tavola_etichette["while"];
...
...
```

Per quanto riguarda il contenitore `multimap` l'unica differenza dal `map` consiste nella possibilità di memorizzare in un `multimap` più volte degli elementi con la stessa chiave, cioè è possibile avere nel `multimap` più coppie chiave-valore con la stessa chiave.

5 I CONTENITORI DELLA LIBRERIA STL

Capitolo 6

Ereditarietà

Si è già menzionato che l'ereditarietà è uno dei concetti fondamentali della programmazione ad oggetti. Lo strumento centrale per l'ereditarietà è la derivazione tra classi. In questo capitolo introdurremo gradualmente i concetti legati all'ereditarietà, in particolare tramite l'esempio della classe `orario`.

6.1 Sottoclassi

Abbiamo modellato il concetto di orario tramite la classe `orario`. Supponiamo ora di voler modellare il concetto di "orario con data" che quindi raffina il concetto di orario. Possiamo utilizzare la classe `orario` per definire una nuova classe `dataora` che eredita da essa tutte le proprietà di `orario` ed a cui attribuiamo le ulteriori proprietà che ci interessano per modellare il concetto di orario con data.

```
// dichiarazione classe orario
class orario {
public:
    orario(int o = 0, int m = 0, int s = 0);
    int Ore() const;
    int Minuti() const;
    int Secondi() const;
    orario operator+(const orario&) const;
    bool operator==(const orario&) const;
    bool operator<(const orario&) const;
    friend ostream& operator<<(ostream&, const orario&);
private:
    int sec;
};
```

6 EREDITARIETÀ

Dichiariamo quindi la classe dataora *derivata* dalla classe orario come segue:

```
class dataora : public orario {  
public:  
    int Giorno() const;  
    int Mese() const;  
    int Anno() const;  
private:  
    int giorno;  
    int mese;  
    int anno;  
};
```

Diremo che la classe orario è una *classe base* e che la classe dataora è una *classe derivata* (direttamente) da orario. Si dice anche che dataora è una *sottoclasse* (diretta) di orario e che orario è una *superclasse* (diretta) di dataora. La parola chiave `public` (si tratta di uno specificatore d'accesso) che precede il nome della classe base orario indica una *derivazione pubblica*. Ogni oggetto della classe derivata dataora contiene come *sottooggetto* un oggetto della classe base orario. Ciò significa che tutti i membri (campi dati, metodi e classi annidate) della classe base orario vengono implicitamente ereditati dalla classe derivata dataora che li può usare liberamente e direttamente come se fossero membri propri. Possiamo quindi scrivere:

```
dataora d;  
int i = d.Ore();
```

in quanto il metodo `Ore()` viene ereditato in dataora da orario. Una classe derivata potrà avere, e nella maggior parte dei casi avrà, ulteriori membri propri (campi dati, metodi e classi annidate) oltre a quelli ereditati dalla classe base.

Caratteristica fondamentale dell'ereditarietà: *Ogni oggetto della classe derivata è utilizzabile anche come oggetto della classe base.* Ciò significa che se D è una classe derivata da una classe base B allora vi è una conversione隐式 da D a B che estrae da ogni oggetto x di D il sottooggetto di x della classe base B. In altri termini, ogni oggetto di una classe derivata può essere convertito implicitamente in un oggetto di una classe base. Questa conversione隐式 vale anche per riferimenti e puntatori: un riferimento ad una classe derivata o un puntatore a una classe derivata può essere convertito implicitamente in un riferimento a una classe base o in un puntatore a una classe base.

Diremo inoltre che la classe derivata dataora è un *sottotipo* (diretto) della classe base orario e, dualmente, che la classe base orario è un *supertipo* (diretto) della classe derivata dataora. Per ereditarietà è quindi possibile usare un oggetto di un sottotipo di un tipo T ovunque sia richiesto un oggetto di tipo T. L'intuizione è che un oggetto di un sottotipo di T è, in particolare, anche di tipo T. Si tratta della cosiddetta relazione "is-a", in italiano "è un": per un qualsiasi oggetto x si usa quindi la terminologia "x è un T" per indicare che l'oggetto x ha un tipo S che è sottotipo di T. Naturalmente, una classe D derivata direttamente da una classe base B può a sua volta agire da classe base per qualche altra classe E.

6.1 Sottoclassi

che derivi direttamente da B : si parla quindi di *gerarchie di classi*. Anche in questo caso diciamo che E è una sottoclasse (indiretta) di B , che E è un sottotipo (indiretto) di B e che B è un supertipo (indiretto) di E .

Bisogna prestare attenzione alla terminologia e alla notazione usati per la relazione di sottotipo/supertipo. Siano X e Y due tipi qualsiasi. La terminologia “ X è un sottotipo di Y ” è leggermente abusata poiché essa spesso denota il fatto che il tipo X o è uguale al tipo Y stesso oppure X è un sottotipo diretto o indiretto del tipo Y . Si usa quindi la terminologia più precisa “ X è un sottotipo proprio di Y ” per riferirsi al fatto che X è un sottotipo di Y diverso da Y stesso. Inoltre, a volte viene usata la notazione $X \leq Y$ per denotare che X è un sottotipo di Y e $X < Y$ per denotare che X è un sottotipo proprio di Y . Terminologia e notazioni analoghe valgono per la relazione di supertipo.

Naturalmente, in ogni gerarchia di classi le conversioni implicite da sottotipo a supertipo valgono lungo tutta la gerarchia: data una classe B , per ogni sottotipo D (in tutta generalità indiretto) di B valgono quindi le seguenti conversioni implicite:

- $D \Rightarrow B$ (tra oggetti)
- $D\& \Rightarrow B\&$ (tra riferimenti)
- $D^* \Rightarrow B^*$ (tra puntatori)

Quindi, se D è una sottoclasse di B allora abbiamo che il tipo puntatore D^* è sottotipo di B^* e che il tipo riferimento $D\&$ è sottotipo di $B\&$.

Nel nostro esempio, grazie alla conversione implicita `dataora` \Rightarrow `orario` possiamo usare un oggetto `dataora` ovunque sia richiesto un oggetto di tipo `orario`. Ad esempio:

```
int F(orario o) { ... }
dataora d;
int i = F(d);
```

Ovviamente, il viceversa non vale. Ad esempio le seguenti istruzioni sono illegali (ovvero non compilano):

```
int G(dataora d) { ... }
orario o;
int i = G(o);
```

Infatti, un oggetto di `dataora` è (in particolare) un oggetto di `orario`, mentre un `orario` non è un `dataora`!

Citiamo dal classico e diffuso testo “*C++ Primer*” (“*C++: Corso di Programmazione*” nella traduzione in italiano) di Lippman e Lajoie: “Uno degli aspetti ironici della programmazione object-oriented in C++ è che, per supportarla, occorre usare puntatori e riferimenti e non oggetti”. Spieghiamo il significato di ciò. Se D è sottotipo di B , b è un oggetto di tipo B e d un oggetto di tipo D allora l’assegnazione $b=d$; estrae (“fisicamente”, cioè a livello di memoria) da d il sottooggetto di tipo B che esso contiene ignorando quindi l’ulteriore informazione contenuta in d specifica della sottoclasse D . D’altronde, la parte di d specifica della classe D , ovvero non ereditata da B , non può essere contenuta nella memoria allocata

per contenere l'oggetto b. Possiamo quindi dire che il C++ non supporta il polimorfismo direttamente tramite oggetti. Il polimorfismo, essenza della programmazione orientata agli oggetti, in C++ è abilitato soltanto quando il sottotipo corrispondente ad una classe derivata è utilizzato indirettamente mediante puntatori o riferimenti alla classe base. Quindi, in C++ quando si parla di polimorfismo si intende principalmente la capacità di un puntatore o riferimento ad una classe base B di riferirsi a una qualsiasi classe derivata da B.

6.1.1 Tipo statico e tipo dinamico

Supponiamo che D sia una sottoclasse di B e consideriamo il seguente frammento:

```
D d; B b;
D* pd=&d;
B* pb=&b;
pb=pd;
```

pb è quindi un puntatore a B. Il tipo B* viene anche detto il *tipo statico* del puntatore pb ("tipo statico" si riferisce al fatto che il tipo è staticamente derivabile dal codice sorgente, ossia viene determinato dal compilatore). Dopo l'assegnazione pb=pd;, che è legale per polimorfismo, pb punta all'oggetto d della classe D: si dice quindi che il *tipo dinamico* di pb dopo una tale assegnazione è D* ("tipo dinamico" si riferisce al fatto che il tipo è deducibile solamente a run-time). Quindi: il tipo statico di un puntatore p è il tipo S* di dichiarazione di p, mentre se in qualche stato dell'esecuzione (cioè in qualche istante dell'esecuzione) il tipo dell'oggetto a cui p effettivamente punta è T allora in quello stato dell'esecuzione T* è il tipo dinamico di p. Mentre il tipo statico è determinato dalla dichiarazione del puntatore e non cambia, il tipo dinamico di un puntatore può quindi variare a run-time. Bisogna prestare particolare attenzione al fatto che la nozione di tipo dinamico è un mero concetto astratto che si usa per ragionare logicamente sull'evoluzione dinamica del tipo degli oggetti a cui puntano le variabili puntatore. Sia per il compilatore che per il sistema run-time che gestisce l'esecuzione di un programma, ogni puntatore ha solo e solamente il tipo con cui è stato dichiarato nel programma, ovvero il suo tipo statico.

Concetti e terminologia analoghi valgono per i riferimenti, come nel seguente frammento:

```
D d; B b;
D& rd=d;
B& rb=d;
```

In questo esempio, B& è il tipo statico del riferimento rb. Con l'inizializzazione rb=d;, che è legale per polimorfismo, rb viene definito come un alias dell'oggetto d di tipo D: D& è quindi il tipo dinamico di rb. Quindi: il tipo statico di un riferimento r è il tipo S& di dichiarazione di r, mentre se il tipo dell'oggetto a cui effettivamente si riferisce r dopo la sua inizializzazione è T allora T& è il tipo dinamico del riferimento r.

Useremo le notazioni TS(p) e TD(p) per denotare, rispettivamente, il tipo statico e dinamico di un puntatore p, e, analogamente, TS(r) e TD(r) denoteranno, rispettivamente, il

6.1 Sottoclassi

tipo statico e dinamico di un riferimento *r*. Vedremo nel seguito l'importanza e le relazioni che intercorrono tra le nozioni di tipo dinamico e polimorfismo.

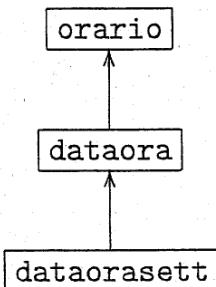
6.1.2 Gerarchie di classi

Abbiamo già osservato che una classe derivata può a sua volta essere usata come classe base per un ulteriore passo di derivazione. Ad esempio, se vogliamo definire un tipo che oltre alle proprietà di dataora memorizzi anche il giorno della settimana possiamo farlo con la seguente derivazione di classe:

```
// tipo enumerazione settimana
enum settimana {lun,mar,mer,gio,ven,sab,dom};

class dataorasett : public dataora {
public:
    settimana GiornoSettimana() const;
private:
    settimana giornosettimana;
};
```

In questo modo abbiamo definito una semplice gerarchia di tre classi, che rappresentiamo tramite la seguente figura detta *diagramma della gerarchia*:



6.1.3 Accessibilità

Una domanda naturale è la seguente: una classe derivata ha accesso alla parte privata di una sua classe base? La risposta è negativa. La parte privata di una qualsiasi classe *B* è inaccessibile alle classi derivate da *B* come lo è per ogni altra classe diversa da *B*. Se *D* è una classe derivata da una classe base *B*, i membri privati di *B* non sono in alcun modo accessibili da *D*, né dai metodi di *D* né dalle funzioni amiche della classe di *D*. Ad esempio, volendo aggiungere alla classe *dataora* un metodo *set2000()* che assegna all'oggetto di invocazione le ore 00:00:00 del 1 gennaio 2000 la seguente definizione sarebbe quindi illegale:

```
dataora::set2000() {
    sec = 0; // NO, illegale!
```

6 EREDITARIETÀ

```
giorno = 1;  
mese = 1;  
anno = 2000;  
}
```

perché il campo dati `sec` è nella parte privata della classe base `orario`.

D'altra parte, una classe derivata da una classe base `B` ha certamente una relazione privilegiata rispetto ad una qualsiasi altra classe esterna a `B`. Infatti, oltre alle parti private e pubbliche di una classe `B` è prevista anche una cosiddetta *parte protetta*, individuata dalla parola chiave `protected`, i cui membri risultano accessibili alle classi derivate da `B` ma non risultano accessibili alle classi esterne a `B`. Tramite la dichiarazione:

```
class orario {  
    ...  
protected:  
    int sec;  
};
```

la definizione del precedente metodo `set2000()` della classe derivata `dataora` risulta corretta, poiché il campo dati `sec` marcato ora come protetto in `orario` diventa accessibile alla sottoclassse `dataora`.

La classe `dataora` è ottenuta dalla classe `orario` per *derivazione pubblica*:

```
class dataora : public orario
```

Con la derivazione pubblica i campi protetti e pubblici della classe base mantengono lo stesso livello di accessibilità anche nella classe derivata. Quindi, ad esempio, il campo protetto `sec` della classe `orario` rimane accessibile anche in `dataorasett` dato che le due derivazioni

`orario` \Rightarrow `dataora` \Rightarrow `dataorasett`

sono entrambe pubbliche.

Vi sono altri due tipologie di derivazione che permettono di modificare i livelli di accessibilità dei membri ereditati dalla classe base. La *derivazione privata*, la cui keyword corrispondente è `private`, rende privati nella classe derivata i membri protetti e pubblici della classe base. La *derivazione protetta*, la cui keyword corrispondente è `protected`, rende protetti nella classe derivata i membri pubblici e protetti della classe base. Entrambe le derivazioni non hanno effetto sui membri privati della classe base, dal momento che sono inaccessibili per la classe derivata (anzi, per tutte le classi esterne). La seguente tabella riassume l'accessibilità di un membro di una classe base `B` in una classe `D` derivata da `B` nelle tre possibili tipologie di derivazione.

6.1 Sottoclassi

Membro	Derivazione	public	protected	private
private		inaccessibile	inaccessibile	inaccessibile
protected		protetto	protetto	privato
public		pubblico	protetto	privato

La forma di derivazione più diffusa è senz'altro quella pubblica, detta anche *ereditarietà di tipo*. In questo caso i metodi pubblici della classe base rimangono pubblici nella classe derivata. Quindi l'interfaccia pubblica di una classe D derivata direttamente da una classe base B si ottiene dall'interfaccia pubblica originaria di B aggiungendo i nuovi membri dichiarati pubblici nella classe derivata D. Ciò permette di realizzare fedelmente la relazione di sottotipo “is-a”: un oggetto della classe derivata è anche un oggetto della classe base. La derivazione privata viene invece detta *ereditarietà di implementazione*. In questo caso l'interfaccia pubblica della classe base B non è inclusa nell'interfaccia pubblica della classe derivata D, visto che la parte pubblica di B diviene privata in D. Tipicamente, l'interfaccia pubblica di B è utilizzata da D nella propria implementazione, in particolare della propria interfaccia pubblica, e ciò giustifica la terminologia “ereditarietà di implementazione”.

Le conversioni implicite indotte dalla derivazione valgono solamente per la derivazione pubblica che è quindi l'unica tipologia di derivazione che supporta la relazione “is-a”. La derivazione protetta e la derivazione privata non inducono alcuna conversione implicita. Vediamo un esempio.

```

class C {
private:
    int i;
protected:
    char c;
public:
    float f;
};

class D: private C { }; // derivazione privata
class E: protected C { }; // derivazione protetta

int main() {
    C c, *pc; D d, *pd; E e, *pe;
    c=d;      // Illegale
    c=e;      // Illegale
    pc=&d;    // Illegale
    pc=&e;    // Illegale
    C& rc=d; // Illegale
}

```

È importante osservare che il concetto di membro protetto di una classe va contro il principio dell'incapsulamento dei dati e dell'information hiding. Infatti, una modifica di un membro protetto di una classe base B potenzialmente potrebbe richiedere la successiva modifica di tutte le classi derivate da B. Per quanto concerne i campi dati di una classe, l'attributo `protected` va quindi usato con oculatezza e parsimonia: per quanto possibile dal contesto, la prassi generale da seguire nella progettazione di una classe rimane quella di dichiarare privati i campi dati di una classe.

```

class C {
private:
    int priv;
protected:
    int prot;
public:
    int publ;
};

class D: private C {
    // prot e publ divengono qui privati
};

class E: protected C {
    // prot e publ divengono qui protetti
};

class F: public D {
    // prot e publ sono qui inaccessibili
public:
    void fF(int i, int j){
        prot=i; // Illegale
        publ=j; // Illegale
    }
};

class G: public E {
    // prot e publ rimangono qui protetti
    void fG(int i, int j){
        prot=i; // OK
        publ=j; // OK
    }
};

```

Sul significato di inaccessible. Supponiamo che `b` sia un campo dati di una classe B, che D sia una sottoclassa di B in cui il campo dati `b` di B diventa inaccessible (ad esempio

6.1 Sottoclassi

perché `b` è privato in `B` e `D` è derivata direttamente e pubblicamente da `B`). Se `d` è un oggetto di `D` allora il sottooggetto di `d` della classe base `B` comunque contiene `b` tra i suoi campi dati. Il seguente esempio mostra questo fatto.

```
class C {
private:
    int i;
public:
    C(): i(1) {}
    void print() {cout << ' ' << i;}
};

class D: public C {
private:
    double z;
public:
    D(): z(3.14) {}
    void print() {
        C::print(); // l'oggetto di invocazione di C::print() è il
                    // sottooggetto di tipo C dell'oggetto di invocazione
        cout << ' ' << z;
    }
};

int main() {
    C c; D d;
    c.print(); cout << endl; // stampa: 1
    d.print(); cout << endl; // stampa: 1 3.14
}
```

Sul significato di protected. Supponiamo che `b` sia un membro di una classe `B`, che `D` sia una sottoclasse di `B` che eredita `b` come membro protetto: il caso comune è che `b` sia protetto in `B` e `D` sia derivata direttamente e pubblicamente da `B`. Questo permette alla classe `D` di accedere al membro `b` dei sottooggetti di tipo `B` degli oggetti appartenenti alla classe `D` ma non di accedere al membro `b` degli oggetti che invece appartengono alla classe base `B`. Ad esempio, supponendo che il membro `b` di `B` sia un campo dati, nel corpo di un metodo `T f(..., B x, ...)` della classe `D` che abbia un parametro `x` di tipo `B`, non si ha accesso al campo dati `x.b`: infatti il membro `b` di `B` risulta comunque inaccessibile in un tentativo di accesso tramite oggetti di `B`. Invece, se la segnatura del metodo è `T f(..., D x, ...)` allora naturalmente si ha accesso al campo dati `x.b`, in quanto `b` è ereditato in `D` come campo dati protetto. Vediamo un esempio.

```
class B {
protected:
```

6 EREDITARIETÀ

```
int i;
void protected_printB() const {cout << ' ' << i;}
public:
    void printB() const {cout << ' ' << i;}
};

class D: public B {
private:
    double z;
public:
    static void stampa(const B& b,const D& d) {
        cout << ' ' << b.i; // Illegale:
                            // "B::i is protected in this context"
        b.printB();         // OK
        b.protected_printB(); // Illegale: "B::protected_printB() is
                            // protected within this context"
        cout << ' ' << d.i; // OK
        d.printB();         // OK
        d.protected_printB(); // OK
    }
};
```

Ereditarietà e amicizia. Supponiamo che una classe base B dichiari friend una certa funzione esterna f ed una classe esterna C e supponiamo che D sia una sottoclass di B. Che rapporto esiste tra D e f e C? Nessuno, perché D non “eredita” in alcun modo le “amicizie” di B, né le funzioni amiche né le classi amiche. Si considerino infatti i seguenti esempi.

```
class C {
private:
    int i;
public:
    C(): i(1) {}
    friend void print(C);
};

class D: public C {
private:
    double z;
public:
    D(): z(3.14) {}
};
void print(C x) {
    cout << x.i << endl; D d;
    cout << d.z; // Illegale: "z is private within this context"
}
```

6.1 Sottoclassi

```
int main() {
    C c; D d;
    print(c); // stampa: 1
    print(d); // OK, stampa: 1
}
```

```
class C {
    friend class Z;
private:
    int i;
public:
    C(): i(1) {}
    friend void print(C);
};

class D: public C {
private:
    double z;
public:
    D(): z(3.14) {}
};

class Z {
public:
    void m() {C c; D d; cout << c.i; // OK
               // cout << d.z; // Illegale: "z is private within this context"
    }
};

int main() {
    Z z;
    z.m(); // stampa: 1
}
```

```
class C {
private:
    int i;
public:
    C(): i(1) {}
    friend void print(C);
};

void print(C x) { cout << x.i << endl; }
```

6 EREDITARIETÀ

```
class D: public C {  
private:  
    double z;  
public:  
    D(): z(3.14) {}  
    friend void print(D);  
};  
void print(D x) { cout << x.z << endl; }  
  
int main() {  
    C c; D d;  
    print(c); // stampa: 1  
    print(d); // stampa: 3.14  
}
```

Conversioni esplicite $B^* \Rightarrow D^*$ e $B& \Rightarrow D&$. È sempre possibile effettuare una conversione esplicita tramite uno `static_cast` per convertire un puntatore ad una classe base B in un puntatore ad una qualsiasi classe D derivata da B . Analogamente, tramite uno `static_cast` possiamo convertire riferimenti ad una classe base B in riferimenti ad una classe D derivata da B . In generale, si dovrà garantire la correttezza di queste conversioni, cioè prima di effettuare queste conversioni esplicite si dovrà avere la certezza che il tipo dinamico del puntatore o riferimento oggetto della conversione esplicita sia E^* oppure $E&$ per un qualche tipo E che sia sottotipo di D . Ad esempio, nel caso di una conversione di un puntatore pb di tipo B^* al tipo D^* , il tipo dinamico di pb immediatamente prima della conversione esplicita dovrà essere E^* per un tipo E che sia sottotipo di D . Naturalmente si tratta di test dinamici a tempo di esecuzione. Se ciò non dovesse essere verificato, chiaramente il codice potrebbe avere un comportamento non prevedibile o si potrebbero addirittura ottenere degli errori fatali in esecuzione. La responsabilità della correttezza di queste conversioni è quindi lasciata completamente al programmatore. Si consideri il prossimo esempio.

```
class C {  
public:  
    int x;  
    void f() { x=4; }  
};  
  
class D: public C {  
public:  
    int y;  
    void g() { x=5; y=6; }  
};  
  
class E: public D {  
public:
```

6.2 Ridefinizione di metodi e campi dati

```
int z;
void h() { x=7; y=8; z=9; }
};

/* VI significa Valore (intero) Imprevedibile */
int main() {
    C c; D d; E e;
    c.f(); d.g(); e.h();

    // PERICOLOSO
    D* pd = static_cast<D*> (&c);

    // errore run-time o stampa: 4 VI
    cout << pd->x << " " << pd->y << endl;

    // PERICOLOSO
    E& re = static_cast<E&> (d);

    // errore run-time o stampa: 5 6 VI
    cout << re.x << " " << re.y << " " << re.z << endl;

    C* pc = &d; pd = static_cast<D*> (pc);           // OK
    cout << pd->x << " " << pd->y << endl;        // stampa: 5 6
    D& rd = e; E& s = static_cast<E&> (rd);         // OK
    cout << s.x << " " << s.y << " " << s.z << endl; // stampa: 7 8 9
}
```

6.2 Ridefinizione di metodi e campi dati

Abbiamo visto che in una classe **D** derivata da **B** tipicamente si aggiungono dei membri propri (campi dati, metodi, classi annidate) ai membri ereditati dalla classe base **B**. D'altra parte, in **D** è anche possibile *ridefinire* i campi dati ed i metodi ereditati da **B**. Ciò significa che nella classe derivata **D** si ridefinisce il significato di un membro **b** ereditato da **B** tramite una nuova definizione che nasconde quella ereditata da **B**. In **D** è possibile usare l'operatore di scoping **B::b** per accedere (quando ciò sia possibile) al membro **b** definito in **B**. Tipicamente, la ridefinizione riguarda i metodi ereditati da una classe base. In inglese si usa il termine “*to redefine*”, oppure in modo improprio “*to override*” (traducibile come sovrascrivere) visto che questo termine è usato più specificatamente e correttamente per la ridefinizione dei metodi virtuali che vedremo in seguito. Ad esempio, è naturale pensare di ridefinire l'operatore di somma della classe **dataora** in modo diverso da come è definito nella classe **orario**, cioè ridefinire il significato del metodo **operator+** di **orario** ereditato in **dataora**. La definizione di **operator+** in **dataora** sarà quindi la seguente, dove

6 EREDITARIETÀ

sommiamo l'oggetto di invocazione di tipo dataora con un parametro di tipo orario e ritorniamo un oggetto di tipo dataora.

```
dataora dataora::operator+(const orario& o) const {  
    dataora aux = *this;  
    aux.sec = sec + 3600*o.Ore() + 60*o.Minuti() + o.Secondi();  
    if (aux.sec >= 86400) {  
        aux.sec = aux.sec - 86400;  
        aux.AvanzaUnGiorno();  
    }  
    return aux;  
}
```

Si presti attenzione al fatto che l'istruzione `aux.sec = sec + o.sec;` darebbe un errore di compilazione perché, anche se `sec` è dichiarato protetto in `orario`, in `dataora` abbiamo che `o.sec` è comunque inaccessibile. Il metodo `AvanzaUnGiorno()` è definito come segue:

```
dataora::AvanzaUnGiorno() {  
    if (giorno < GiorniDelMese()) giorno++;  
    else if (mese < 12) { giorno = 1; mese++; }  
    else { giorno = 1; mese = 1; anno++; }  
}
```

A questo punto, possiamo invocare l'operatore di somma di `orario` o `dataora` come nel seguente frammento di codice:

```
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+  
dataora x = o1 + o2; // Illegale: nessuna conversione  
// orario => dataora  
orario y = d1 + d2; // OK  
d1.orario::operator+(d2); // Invoca orario::operator+
```

Possiamo anche ridefinire un campo dati `x` di una classe `B` in una classe `D` derivata da `B`, naturalmente se `x` è accessibile in `D` (altrimenti non ha senso). Si tratta quindi di definire nella classe derivata `D` un nuovo campo dati con lo stesso identificatore `x`, ed il cui tipo potrà naturalmente essere diverso dal tipo di `x` in `B`. È molto meno comune della ridefinizione dei metodi ed ha come semplice effetto il mascheramento del corrispondente campo dati ereditato. Vediamo un esempio.

6.2 Ridefinizione di metodi e campi dati

```
class B {
protected:
    int x;
public:
    B() : x(2) {}
    void print() { cout << x << endl; }
};

class D: public B {
private:
    double x; // ridefinizione del campo dati x
public:
    D() : x(3.14) {}
    // ridefinizione di print()
    void print() { cout << x << endl; } // è la x di D
    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
}
```

Sia D una classe derivata da una classe base B. Sia m() un nome di metodo, possibilmente sovraccaricato, nella classe B che sia accessibile in D. Allora, una ridefinizione in D del nome 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 di scoping B::.. Questa regola è nota con il nome di *name hiding rule*. Questo vale quindi per ogni ridefinizione del metodo m(), che naturalmente può essere di tre tipologie: (1) stessa segnatura (lista dei parametri e tipo di ritorno) di una delle versioni disponibili in B; (2) stessa lista dei parametri ma diverso tipo di ritorno; (3) diversa lista dei parametri. Overloading e ridefinizione sono quindi concetti ben diversi. La name hiding rule del C++ appare ad alcuni esperti di programmazione ad oggetti discutibile e con motivazioni non chiare. Possiamo segnalare il fatto che il linguaggio di programmazione ad oggetti Java non ha adottato la name hiding rule.

Ad esempio, se in dataora ridefiniamo il metodo Ore con la nuova segnatura:

```
int dataora::Ore(int) const
allora non possiamo più scrivere:
dataora d;
cout << d.Ore(); // Illegale
```

6 EREDITARIETÀ

per invocare il "vecchio" metodo `Ore` della classe `orario`, perché questo è mascherato in `dataora` dalla ridefinizione. Possiamo però usare l'operatore di scoping per invocare il metodo `Ore` di `orario`:

```
dataora d;  
cout << d.orario::Ore();
```

Una diversa possibilità consiste nell'utilizzare una *dichiarazione di uso* che introduce ogni membro citato della classe base nel campo d'azione della classe derivata. Ad esempio, se nella classe `dataora` ridefiniamo il metodo `Ore` della classe `orario` con la nuova segnatura:

```
int dataora::Ore(int) const;
```

e contestualmente aggiungiamo in `dataora` la seguente dichiarazione di uso

```
using dataora::Ore;
```

allora la seguente invocazione del metodo `orario::Ore()` diventa legale:

```
dataora d;  
cout << d.Ore(); // OK
```

Infatti, come effetto della dichiarazione di uso, il membro `orario::Ore()` della classe base `orario` è ora inserito nell'insieme dei metodi di istanza sovraccaricati associati al nome di metodo `Ore` della classe derivata `dataora`. Si noti che una dichiarazione di uso per un metodo specifica solamente l'identificatore del metodo e non la lista dei parametri. Quindi, se il metodo `m()` oggetto della dichiarazione di uso nella classe `D` derivata da `B` è sovraccaricato all'interno di `B` allora tutti gli overloading di `m()` in `B` vengono ereditati in `D`. Si consideri il seguente esempio.

```
class B {  
public:  
    // overloading di m  
    void m(int x) {cout << "B::m(int)";}  
    void m(int x, int y) {cout << "B::m(int,int)";}  
};  
  
class D: public B {  
public:  
    // dichiarazione di uso di B::m  
    using B::m;  
  
    // ridefinizione di m  
    void m(int x) {cout << "D::m(int)";}  
    void m() {cout << "D::m()";}  
};
```

6.2 Ridefinizione di metodi e campi dati

```
int main () {
    D d;
    d.m(3);      // Compila e stampa: D::m(int)
    d.m();        // Compila e stampa: D::m()
    d.m(3,5);    // Compila e stampa: B::m(int,int)
    d.B::m(4);   // Compila e stampa: B::m(int)
}
```

Supponiamo che `D` sia una sottoclasse di `B`, che `m` sia un membro (campo dati o metodo) di `B` accessibile in `D` e che `m` sia ridefinito in `D`. Sia `B*` `pb` un puntatore alla classe base definito in un contesto in cui `pb` ha accesso al membro `m`. Allora `pb->m` seleziona sempre il membro `m` della classe base `B` e non il membro `m` ridefinito in `D`, anche quando `TD(pb) = D*`. Analogamente, se `rb` è un riferimento alla classe base `B` allora `rb.m` seleziona sempre il membro `m` della classe base `B` indipendentemente dal tipo dinamico di `rb`. Consideriamo il caso più comune in cui `m()` è un metodo. Allora questo succede perché il legame — *binding* in inglese — tra l’oggetto d’invocazione ed il metodo invocato `m()` è statico, ovvero è determinato staticamente dal compilatore considerando il tipo statico del puntatore `pb` o del riferimento `rb`. Nell’invocazione `pb->m()`, poiché `pb` ha tipo statico `B*`, il compilatore controlla che il metodo `m()` sia disponibile nella classe `B` e quindi l’invocazione `pb->m()` viene compilata nella chiamata a questo metodo `m()` di `B`. Vedremo nel seguito come sarà invece possibile ritardare a tempo di esecuzione il binding tra oggetto di invocazione e metodo in modo da selezionare in modo polimorfo un metodo ridefinito nella classe derivata che corrisponde al tipo dinamico di un puntatore o riferimento. Naturalmente, per un oggetto `b` della classe `B`, se `d` è un oggetto di `D`, anche dopo l’assegnazione `b=d`, la selezione `b.m` si riferirà sempre e comunque al membro `m` della classe `B`.

Consideriamo ora una serie di esempi ed esercizi sulla ridefinizione.

Esempio 6.2.1.

```
class B {
public:
    int f() const { cout << "B::f()\n"; return 1; }

    int f(string) const { cout << "B::f(string)\n"; return 2; }
};

class D : public B {
public:
    // ridefinizione con la stessa segnatura
    int f() const { cout << "D::f()\n"; return 3; }
};

class E : public B {
public:
```

6 EREDITARIETÀ

```
// ridefinizione con cambio del tipo di ritorno
void f() const { cout << "E::f()\n"; }
};

class H : public B {
public:
    // ridefinizione con cambio lista argomenti
    int f(int) const { cout << "H::f()\n"; return 4; }
};

int main() {
    string s; B b; D d; E e; H h;
    int x = d.f(); // Stampa: D::f()
    d.f(s);        // Illegale
    x = e.f();      // Illegale
    x = h.f();      // Illegale
    x = h.f(1);    // Stampa: H::f()
}
```

```
class C {
public:
    void f(int x) {}
    void f() {}
};

class D: public C {
    int y;
public:
    void f(int x) {f(); y=3+x;} // Illegale:
                                // "no matching function for D::f()"
};
```

```
class C {
public:
    int x;
    void f() {x=1;}
};

class D: public C {
public:
    int y;
    void f() {std::cout << "*"; y=3; f();}
    // Errore logico: è una ricorsione infinita
};
```

6.2 Ridefinizione di metodi e campi dati

```
int main() {
    D d; d.f(); // compila ma provoca un errore run-time:
} // provoca uno stack overflow
```

```
class C {
public:
    void f() {}
    void f(int x) {}
};

class D: public C {
public:
    int f() {return 1;} // cambia il tipo di ritorno
};

int main() {
    D d; d.f(); // OK, chiama D::f()
//d.f(2); // Illegale
}
```

```
class C {
public:
    void f() {cout << "C::f\n";}
};

class D: public C {
public:
    void f() {cout << "D::f\n";} // ridefinizione
};

class E: public D {
public:
    void f() {cout << "E::f\n";} // ridefinizione
};

int main() {
    C c; D d; E e;
    C* pc = &c; E* pe = &e;
    c = d; // OK: conversione D => C
    c = e; // OK: conversione E => C
    d = e; // OK: conversione E => D
    d = c; // Illegale: C non ha una classe base D
    C& rc=d; // OK: conversione D => C
    D& rd=e; // OK: conversione E => D
    pc->f(); // OK, stampa: C::f
```

6 EREDITARIETÀ

```
pc = pe; // OK: conversione E* => C*
rd.f(); // OK, stampa: D::f
c.f(); // OK, stampa: C::f
pc->f(); // OK, stampa: C::f
}
```

```
class C {
public:
    int i; double a;
    void f(double) { cout << "C::f(double)\n"; }
};

class D: public C {
public:
    int* a; // nasconde double C::a
    void f(int*) // nasconde void C::f(double)
};

void D::f(int* p){
    int j; double b; int* q;
    j=i; // OK
    q=a; // OK, a in D è di tipo int*
    b=a; // Illegale: in D a è di tipo int*
    b = C::a; // OK, uso dell'operatore di scoping
    cout << "D::f(int*)\n";
}

int main() {
    int n; double x; int* q;
    D d;
    d.i = n; // OK
    d.a = q; // OK
    d.a = x; // Illegale: d.a ha tipo int*
    d.C::a = x; // OK
    d.f(q); // OK, stampa: D::f(int*)
    d.f(x); // Illegale: d.f() richiede un parametro int*
    d.C::f(x); // OK, stampa: C::f(double)
}
```

Esercizio 6.2.2. I seguenti programmi compilano. Quali stampe provocano le loro esecuzioni?

```
// PROGRAMMA UNO
class C {
public:
```

6.2 Ridefinizione di metodi e campi dati

```
int x;
void f() { x=1; }

};

class D: public C {
public:
    int y;
    void f() { C::f(); y=2; }
};

int main() {
    C c; D d; c.f(); d.f();
    cout << c.x << endl;
    cout << d.x << " " << d.y;
}
```

```
// PROGRAMMA DUE
class C {
public:
    int a;
    void fC() { a=2; }
};

class D: public C {
public:
    double a; // ridefinizione
    void fD() { a=3.14; C::a=4; }
};

class E: public D {
public:
    char a; // ridefinizione
    void fE() { a='*'; C::a=5; D::a=6.28; }
};

int main() {
    C c; D d; E e;
    c.fC(); d.fD(); e.fE();
    D* pd = &d; E& pe = e;
    cout << pd->a << ' ' << pe.a << endl;
    cout << pd->a << ' ' << pd->D::a << ' ' << pd->C::a << endl;
    cout << pe.a << ' ' << pe.D::a << ' ' << pe.C::a << endl;
    cout << e.a << ' ' << e.D::a << ' ' << e.C::a << endl;
}
```

6.3 Costruttori, assegnazione e distruttore

Esaminiamo ora come si comportano i costruttori, l'assegnazione ed il distruttore nelle classi derivate. Naturalmente, i costruttori, l'assegnazione e il distruttore della classe base non sono ereditati dalla classe derivata, ma c'è la possibilità per costruttori, assegnazione e distruttore della classe derivata di invocare quelli della classe base.

Ricordiamo che ogni oggetto di una classe D derivata direttamente da una classe base B, per effetto dei campi dati ereditati in D da B, contiene un sottooggetto della classe base B. Dunque, quando si istanzia un oggetto d di D occorrerà richiamare, esplicitamente o implicitamente nel costruttore di D, un costruttore di B per creare ed inizializzare il sottooggetto di d della classe base B.

Invocazione esplicita: è possibile inserire esplicitamente nella lista di inizializzazione del costruttore di D un'invocazione esplicita di un qualsiasi costruttore di B.

Invocazione implicita: se la lista di inizializzazione del costruttore di D non include invocazioni esplicite di qualche costruttore di B allora viene implicitamente ed automaticamente invocato il costruttore di default di B, che dovrà quindi essere disponibile.

Quindi, la lista di inizializzazione di un costruttore di una classe D derivata direttamente da B in generale può contenere invocazioni di costruttori per i campi dati (propri) di D e l'invocazione di un costruttore della classe base B. Si presti attenzione a non confondere costruttori e campi dati della classe base: la lista di inizializzazione di D non può contenere invocazioni di costruttori per i campi dati della classe base B. L'esecuzione di un tale costruttore di D avviene nel seguente modo:

1. viene sempre e comunque invocato per primo un costruttore della classe base B, esplicitamente quando appare (in qualsiasi ordine) nella lista di inizializzazione, oppure implicitamente il costruttore di default di B quando la lista di inizializzazione non include una invocazione esplicita;
2. successivamente, secondo il comportamento già noto, viene eseguito il costruttore "proprio" di D, ossia vengono costruiti i campi dati propri di D e successivamente viene eseguito il corpo del costruttore.

In particolare, se nella classe derivata D si omette qualsiasi costruttore allora, come al solito, è disponibile il costruttore di default standard di D. Il suo comportamento è quindi il seguente:

1. richiama il costruttore di default di B;
2. successivamente si comporta come il costruttore di default standard "proprio" di D, ossia richiama i costruttori di default per tutti i campi dati di D.

Torniamo all'esempio della classe dataora. Nella dichiarazione
dataora d;

6.3 Costruttori, assegnazione e distruttore

l'oggetto d viene costruito dal costruttore di default standard della classe dataora che prima richiama il costruttore standard per il sottooggetto della classe base orario che assegna 0 al campo sec e poi richiama i "costruttori di default" per i campi dati giorno, mese e anno di tipo primitivo int, ovvero alloca la memoria per questi campi dati lasciandola indefinita. Quindi:

```
cout << d.Ore(); // stampa: 0  
cout << d.Giorno(); // stampa un valore indefinito
```

Se invece definiamo il costruttore di default per la classe derivata come segue otteniamo un diverso comportamento:

```
dataora::dataora(): giorno(1), mese(1), anno(2000) {}  
  
dataora d;  
cout << d.Ore(); // stampa: 0  
cout << d.Giorno(); // stampa: 1
```

È naturale definire un costruttore con parametri per la classe derivata dataora che inizializzi tutti i campi dati sia della classe base orario che della classe derivata richiamando esplicitamente il costruttore con parametri della classe orario.

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

In questo caso l'invocazione del costruttore orario(o,m,s) si sostituisce all'invocazione automatica del costruttore standard orario() che non viene più effettuata. Ad esempio:

```
dataora d(2003,11,17,11,55,13);  
cout << d.Ore(); // stampa: 11  
cout << d.Giorno(); // stampa: 17
```

Vediamo alcuni esempi.

Esempio 6.3.1.

```
class Z {  
public:  
    Z() {cout << "Z0 ";}  
};  
  
class C {  
private:  
    int x;  
public:  
    C(int z=1): x(z) {cout << "C01 ";}
```

6 EREDITARIETÀ

```
};

class D: public C {
private:
    int y;
    Z z;
};

int main() {
    D d; // costruttore standard
}
// Stampa: C01 Z0
```

```
class Z {
public:
    Z() {cout << "Z0 ";}
    Z(double d) {cout << "Z1 ";}
};

class C {
private:
    int x;
    Z w;
public:
    C(): w(6.28), x(8) {cout << x << " C0 ";}
    C(int z): x(z) {cout << x << " C1 ";}
};

class D: public C {
private:
    int y;
    Z z;
public:
    D(): y(0) {cout << "D0 ";}
    D(int a): y(a), z(3.14), C(a) {cout << "D1 ";}
};

int main() {
    D d; cout << endl; // Stampa: Z1 8 C0 Z0 D0
    D e(4);           // Stampa: Z0 4 C1 Z1 D1
}
```

Esercizio 6.3.2. Il seguente programma compila correttamente. Quale stampa provoca la sua esecuzione?

6.3 Costruttori, assegnazione e distruttore

```

class C {
private:
    int i;
protected:
    int p;
public:
    C() {cout << "C0 ";}
    C(int x) {cout << "C1 ";}
};

class D: private C {
protected:
    int j;
public:
    D(): C(2) {cout << "D0 ";}
    D(double x) {cout << "D1 ";}
};

class E: public C {
private:
    D d;
public:
    int k;
    E() : C(6) {cout << "E0 ";}
};

```

```

class F: public D {
protected:
    E e;
public:
    F() : D(3.2) {cout << "F0 ";}
    F(float x) {cout << "F1 ";}
};

class G: public E {
private:
    F f;
    C c;
public:
    G(): E() {cout << "G0 ";}
    G(char x) {cout << "G1 ";}
};

int main() { G g; }

```

Il costruttore di copia standard di una classe D derivata direttamente da una classe base B invocato su un oggetto x di D innanzitutto costruisce il sottooggetto di B invocando il costruttore di copia (standard oppure ridefinito) di B sul corrispondente sottooggetto di x e successivamente costruisce ordinatamente i campi dati propri di D invocando i relativi costruttori di copia. Naturalmente, il costruttore di copia può essere ridefinito in D: esso può invocare esplicitamente il costruttore di copia di B o qualsiasi altro costruttore di B. In mancanza di invocazioni esplicite, viene automaticamente invocato il costruttore di default di B (e non il costruttore di copia di B, attenzione!).

Esempio 6.3.3. Il confronto tra le stampe prodotte dall'esecuzione dei seguenti programmi mostra il comportamento del costruttore di copia standard.

```

class Z {
public:
    Z() {cout << "Z0 ";}
    Z(const Z& x) {cout << "Zc ";}
};

class C {

```

6 EREDITARIETÀ

```
private:  
    Z w;  
public:  
    C() {cout << "C0 ";}  
    C(const C& x): w(x.w) {cout << "Cc ";}  
};  
  
class D: public C {  
private:  
    Z z;  
public:  
    D() {cout << "D0 ";}  
};  
  
int main() {  
    D d; cout << "UNO\n";  
    D e = d; cout << "DUE"; // costruttore di copia standard  
}  
// Stampa:  
// Z0 C0 Z0 D0 UNO  
// Zc Cc Zc DUE
```

```
class Z {  
public:  
    Z() {cout << "Z0 ";}  
    Z(const Z& x) {cout << "Zc ";}  
};  
  
class C {  
private:  
    Z w;  
public:  
    C() {cout << "C0 ";}  
    C(const C& x): w(x.w) {cout << "Cc ";}  
};  
  
class D: public C {  
private:  
    Z z;  
public:  
    D() {cout << "D0 ";}  
    D(const D& x) {cout << "Dc ";}  
};  
  
int main() {
```

6.3 Costruttori, assegnazione e distruttore

```
D d; cout << "UNO\n";
D e = d; cout << "DUE"; // costruttore di copia ridefinito
}
// Stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 Dc DUE
```

L'assegnazione standard di una classe **D** derivata direttamente da una classe base **B** invoca preventivamente l'assegnazione (standard oppure ridefinita) della classe base **B** sul sottoggetto e successivamente esegue l'assegnazione ordinatamente membro a membro dei campi dati propri di **D** invocando le corrispondenti assegnazioni (standard oppure ridefinite). Si noti che se l'assegnazione viene ridefinita in **D** allora viene eseguito solamente il suo codice, cioè la ridefinizione dell'assegnazione non provoca alcuna invocazione implicita.

Esempio 6.3.4. Il confronto tra le stampe prodotte dall'esecuzione dei seguenti programmi mostra il comportamento dell'assegnazione standard.

```
class Z {
public:
    Z() {cout << "Z0 ";}
    Z(const Z& x) {cout << "Zc ";}
    Z& operator=(const Z& x) {cout << "Z= "; return *this;}
};

class C {
protected:
    Z w;
public:
    C() {cout << "C0 ";}
    C(const C& x): w(x.w) {cout << "Cc ";}
    C& operator=(const C& x) {w=x.w; cout << "C= "; return *this;}
};

class D: public C {
private:
    Z z;
public:
    D() {cout << "D0 ";}
    D(const D& x) {cout << "Dc ";}
};

int main() {
    D d; cout << "UNO\n";
    D e; cout << "DUE\n";
    e=d; cout << "TRE";
```

6 EREDITARIETÀ

```
}
```

```
// Stampa:
```

```
// Z0 C0 Z0 D0 UNO
```

```
// Z0 C0 Z0 D0 DUE
```

```
// Z= C= Z= TRE
```

```
class Z {
```

```
public:
```

```
    int x;
```

```
    Z(): x(0) {cout << "Z0 " ;}
```

```
    Z(const Z& x) {cout << "Zc " ;}
```

```
    Z& operator=(const Z& x) {cout << "Z= "; return *this; }
```

```
};
```

```
class C {
```

```
public:
```

```
    Z w;
```

```
    C() {cout << "C0 " ;}
```

```
    C(const C& x): w(x.w) {cout << "Cc " ;}
```

```
    C& operator=(const C& x) {w=x.w; cout << "C= "; return *this; }
```

```
};
```

```
class D: public C {
```

```
public:
```

```
    Z z;
```

```
    D(): cout << "D0 " ;
```

```
    D(const D& x) {cout << "Dc " ;}
```

```
    D& operator=(const D& x) {z=x.z; cout << "D= "; return *this; }
```

```
// l'assegnazione è definita male: chi ci pensa ad assegnare il
```

```
// campo dati w di C? E se w fosse marcato private?
```

```
};
```

```
int main() {
```

```
    D d; d.w.x = 3; cout << "UNO\n";
```

```
    D e; e.w.x = 5; cout << "DUE\n";
```

```
    e=d; cout << "TRE\n";
```

```
    cout << e.w.x << ' ' << d.w.x << " QUATTRO";
```

```
}
```

```
// Stampa:
```

```
// Z0 C0 Z0 D0 UNO
```

```
// Z0 C0 Z0 D0 DUE
```

```
// Z= D= TRE
```

```
// 5 3 QUATTRO
```

Il distruttore standard di una classe D derivata direttamente da B richiama implicitamente il distruttore (standard oppure ridefinito) della classe base B per distruggere il sottooggetto di B soltanto dopo l'azione di distruzione standard propria di D, cioè la distruzione dei

6.3 Costruttori, assegnazione e distruttore

campi dati propri di D nell'ordine inverso a quello di costruzione tramite l'invocazione dei corrispondenti distruttori (standard oppure ridefiniti). Se il distruttore viene ridefinito in D allora innanzitutto viene eseguito il codice di tale distruttore, quindi avviene la distruzione dei campi dati propri di D, ed infine viene invocato il distruttore (standard oppure ridefinito) della classe base B per distruggere il sottooggetto di B.

Esempio 6.3.5. Il confronto tra le stampe prodotte dall'esecuzione dei seguenti programmi mostra il comportamento del distruttore standard.

```
class Z {
public:
    Z() {cout << "Z0 ";}
    ~Z() {cout << "~Z ";}
};

class C {
protected:
    Z w;
public:
    C() {cout << "C0 ";}
    ~C() {cout << "~C ";}
};

class D: public C {
private:
    Z z;
public:
    D() {cout << "D0 ";}
};

int main() {
    D* p = new D; cout << "UNO\n";
    delete p; cout << "DUE"; // distruttore standard
}
// Stampa:
// Z0 C0 Z0 D0 UNO
// ~Z ~C ~Z DUE
```

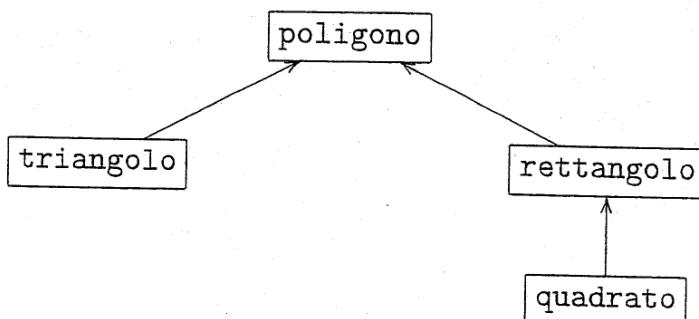
```
class Z {
public:
    Z() {cout << "Z0 ";}
    ~Z() {cout << "~Z ";}
};

class C {
```

6 EREDITARIETÀ

```
protected:  
    Z w;  
public:  
    C() {cout << "C0 ";}  
    ~C() {cout << "~C ";}  
};  
  
class D: public C {  
private:  
    Z z;  
public:  
    D() {cout << "D0 ";}  
    ~D() {cout << "~D ";}  
};  
  
int main() {  
    D* p = new D; cout << "UNO\n";  
    delete p; cout << "DUE"; // distruttore ridefinito  
}  
// Stampa:  
// Z0 C0 Z0 D0 UNO  
// ~D ~Z ~C ~Z DUE
```

Esempio 6.3.6. Il seguente esempio mostra come definire una semplice gerarchia di classi i cui oggetti rappresentano poligoni.



```
class punto {  
private:  
    double x, y;  
public:  
    double getX() const; // coordinata x  
    double getY() const; // coordinata y  
    // metodo statico che calcola la distanza tra due punti  
    static double lung(const punto& p1, const punto& p2);  
};
```

6.3 Costruttori, assegnazione e distruttore

```
class poligono {  
protected:  
    int nvertici;  
    punto* pp; // array dinamico di punti  
public:  
    // non è disponibile il costruttore di default  
    poligono(int, const punto v[]); // v array ordinato dei vertici  
    ~poligono(); // distruttore profondo  
    poligono(const poligono&); // copia profonda  
    poligono& operator=(const poligono&); // assegnazione profonda  
    double perimetro() const;  
};
```

Si lascia per esercizio la definizione dei metodi della classe `poligono`.

```
class rettangolo: public poligono {  
public:  
    rettangolo(const punto v[]) : poligono(4,v) {}  
    double perimetro() const; // ridefinizione  
    double area() const; // nuovo metodo  
};  
  
double rettangolo::perimetro() const {  
    double base = punto::lung(pp[1], pp[0]);  
    double altezza = punto::lung(pp[2], pp[1]);  
    return ((base + altezza)*2);  
}  
  
double rettangolo::area() const {  
    double base = punto::lung(pp[1], pp[0]);  
    double altezza = punto::lung(pp[2], pp[1]);  
    return (base * altezza);  
}
```

```
class quadrato: public rettangolo {  
public:  
    quadrato(const punto v[]) : rettangolo(v) {}  
    double perimetro() const; // ridefinizione  
    double area() const; // ridefinizione  
};  
  
double quadrato::perimetro() const {  
    double lato = punto::lung(pp[1], pp[0]);  
    return (lato * 4);  
}
```

```
double quadrato::area() const {
    double lato = punto::lung(pp[1], pp[0]);
    return (lato * lato);
}
```

```
class triangolo: public poligono {
public:
    triangolo(const punto v[]) : poligono(3, v) {}
    double area() const; // nuovo metodo
};

double triangolo::area() const { // usa la formula di Erone
    double p = perimetro()/2;
    double a=punto::lung(pp[1],pp[0]), b=punto::lung(pp[2],pp[1]),
           c=punto::lung(pp[0],pp[2]);
    return sqrt(p*(p-a)*(p-b)*(p-c));
}
```

6.4 Ereditarietà e template

Naturalmente, l'ereditarietà può essere usata anche con i template di classe. In particolare sia la classe base che la classe derivata possono essere definite come template di classe. Analizziamo sinteticamente le tre principali modalità di derivazione che coinvolgono i template di classe.

- (A) Classe base template e classe derivata da una istanza della classe base.

```
template <class T>
class base { ... };

class derivata : public base<int> { ... };
```

Ogni oggetto della classe derivata contiene come sottooggetto un oggetto dell'istanza `base<int>` della classe template base.

- (B) Classe base non template e classe derivata template.

```
class base { ... };

template <class T>
class derivata : public base { ... };
```

6.4 Ereditarietà e template

Ogni oggetto di ogni istanza della classe template derivata contiene come sottooggetto un oggetto della classe base.

- (C) Classe base e classe derivata entrambe template.

```
template <class T>
class base { ... };

template <class Tp>
class derivata : public base<Tp> { ... };
```

I parametri (di tipo o valore) della classe template derivata devono essere un sovrainsieme di quelli della classe base. Si tratta quindi di una derivazione associata. Ogni oggetto di una istanza della classe template derivata contiene come sottooggetto un oggetto dell'istanza associata della classe template base. In questa tipologia di derivazione bisogna prestare attenzione alle modalità di accesso ai membri della classe base: infatti per accedere ad un membro *m* (campo dati o metodo) della classe base tramite l'oggetto di invocazione è necessario usare esplicitamente la sintassi *this->m* (il compilatore g++ sino alla versione 3.3 era tollerante su questa regola mentre essa deve essere rispettata dalla versione 3.4), come nel seguente esempio.

```
template <class T> class B {
public:
    int m;
    int n;
    int f() {...}
    int g() {...}
};

int n = 1;      // variabile globale
int g() {...}  // funzione globale

template <class T> class D : B<T> {
public:
    void h() {
        m = 2;          // Illegale
        this->m = 2;   // OK
        f();           // Illegale
        this->f();    // OK
        n = 3;          // Assegnazione a ::n
        this->n = 3;  // Assegnazione al campo dati B::n
        g();           // Invoca ::g()
        this->g();    // Invoca il metodo B::g()
    }
};
```

```
    }
};
```

Invece, non si possono derivare sottoclassi non-template da una classe base template (ciò non avrebbe senso).

6.5 Metodi virtuali

Abbiamo visto che un oggetto di una classe derivata può essere usato ovunque sia richiesto un oggetto della classe base, ossia esiste una conversione implicita da oggetti di una classe derivata a oggetti di una classe base che estrae i corrispondenti sottooggetti. Ad esempio, la funzione

```
void F(orario o);
```

con un parametro formale di tipo `orario` può essere richiamata passandole un parametro attuale di tipo `dataora`:

```
dataora d; F(d);
```

Cosa succede di preciso? Poiché il parametro `o` è passato per valore viene invocato il costruttore di copia di `orario` che costruisce `o` copiando il sottooggetto di tipo `orario` del parametro attuale `d`. Consideriamo invece la seguente funzione `G`.

```
void G(const orario& o);
...
dataora d;
G(d);
```

Se il parametro formale di tipo `orario` è un riferimento (costante) possiamo ancora richiamare la funzione con un parametro attuale di tipo `dataora`: in questo caso non viene fatta una copia del sottooggetto di `d` ma il parametro formale `o` diventa invece un alias dell'oggetto `d` di tipo `dataora`. Nella funzione `G` abbiamo che $TS(o) = \text{const orario\&}$ e quindi nel corpo di `G` il parametro `o` viene usato come sottooggetto di tipo `orario` dell'oggetto `d` di tipo `dataora`. Non viene invece preso in considerazione il fatto che nell'invocazione `G(d)` il parametro `o` ha tipo dinamico `const dataora\&`, cioè il fatto che a run-time succede che `o` in effetti è un riferimento ad un oggetto di `dataora`. Supponiamo ad esempio che in `G` il parametro `o` venga usato come oggetto di invocazione di un metodo `Stampa()` della classe `orario` che è stato ridefinito nella classe `dataora`:

```
void G(const orario& o) { o.Stampa(); }
```

In questo caso se invochiamo `G` con parametro attuale `d` di tipo `dataora` verrà invocato il metodo `orario::Stampa()` o il metodo ridefinito `dataora::Stampa()`? Nel caso considerato il legame tra l'oggetto di invocazione `o` e la funzione `Stampa()` è *statico*, cioè viene effettuato a tempo di compilazione. Viene quindi invocato il metodo

6.5 Metodi virtuali

orario::Stampa(). Possiamo però fare in modo che, quando il parametro è passato per riferimento, l'associazione tra oggetto di invocazione e metodo da invocare venga effettuata a tempo di esecuzione in base al tipo dinamico del parametro e non in base al suo tipo statico. Un discorso analogo vale per un parametro di tipo puntatore, ad esempio orario*:

```
void G(orario* p) { p->Stampa(); }
```

Lo strumento C++ per poter fare ciò è la dichiarazione di un *metodo virtuale*. Dichiarendo virtuale, tramite la keyword **virtual**, il metodo Stampa() della classe base orario nel modo seguente:

```
class orario {
    virtual void Stampa();
    ...
};

void G(const orario& o) { o.Stampa(); }
```

nella funzione G verrà invocato dataora::Stampa() se TD(o) = const dataora& mentre verrà invocato orario::Stampa() se TD(o) = const orario&. In altre parole, se in una invocazione G(x); il parametro attuale x è di tipo dataora viene invocato il metodo dataora::Stampa() mentre se x è di tipo orario viene invocato il metodo orario::Stampa(). Si tratta del cosiddetto *legame dinamico* (o *dynamic binding* oppure *late binding*) tra oggetto di invocazione e metodo virtuale: il metodo virtuale da invocare effettivamente verrà selezionato solamente a tempo di esecuzione (a run-time) e non staticamente dal compilatore (tale meccanismo di invocazione viene anche detto *dynamic lookup*). Ciò vale per i riferimenti di una classe base ma naturalmente anche per i puntatori ad una classe base.

```
dataora d;
orario* p = &d;
p->Stampa(); // oppure: (*p).Stampa()
```

In questo caso, se orario::Stampa() è stato dichiarato virtuale allora viene invocato dataora::Stampa(), altrimenti se orario::Stampa() non è virtuale viene invocato (staticamente determinato) orario::Stampa().

Quindi marcando virtuale un metodo m() di una classe base B il progettista delega alle ridefinizioni di m() nelle sottoclassi di B il compito di implementare quel metodo m() in modo specifico alla particolare sottoclasse. Una ridefinizione di un metodo virtuale m() viene anche detta *overriding* di m(). Quando in una classe B si dichiara virtuale un metodo m(), esso resta virtuale in tutta la gerarchia di classi che derivano dalla classe B, anche se m() non è dichiarato esplicitamente virtuale nella ridefinizione operata dalle sottoclassi di B. A volte, per maggiore chiarezza, si segue la prassi di dichiarare esplicitamente che un metodo è virtuale in ogni sottoclasse dove viene ridefinito. Se una sottoclasse D di B

6 EREDITARIETÀ

non ridefinisce il metodo virtuale `m()` allora `D` semplicemente eredita la definizione di `m()` presente nella sua superclasse diretta.

Il meccanismo del dynamic binding si applica anche agli operatori: infatti, anche gli operatori (definiti internamente ad una classe) possono essere ridefiniti dichiarandoli virtuali ed in questo caso diventano dei metodi virtuali a tutti gli effetti.

Naturalmente, perché una invocazione di un metodo virtuale `m()` tramite un puntatore `p` di tipo `B*` (oppure tramite un riferimento) possa compilare correttamente è necessario che il metodo `m()` sia disponibile nella classe `B`. Ad esempio, si consideri la seguente situazione.

```
class B { };

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

int main() {
    B b; D d; B* p = &d;
    p->m(); // Illegale
}
```

Al momento della chiamata `p->m()`, il puntatore `p` ha tipo statico `B*` e tipo dinamico `D*`. Non si deve compiere il grave errore di pensare che la chiamata `p->m()` sia legale e la sua esecuzione comporti l'invocazione del metodo virtuale `m()` della classe `D`: la chiamata `p->m()` provoca un errore in compilazione perché il metodo `m()` non è disponibile nella classe `B` (il compilatore non è in grado di determinare i tipi dinamici di puntatori e riferimenti!). Vedremo nel seguito che il meccanismo dell'identificazione di tipo a run-time permetterà di convertire dinamicamente il puntatore `p` al tipo `D*` in modo da poter invocare il metodo `m()` sull'oggetto di tipo `D` puntato da `p`.

Nell'overriding bisogna prestare particolare attenzione alla segnatura dei metodi. Se consideriamo un metodo virtuale

```
virtual T m(T1, ..., Tn);
```

di una classe base `B`, allora l'overriding di `m()` in una classe `D` derivata da `B` deve mantenere la stessa segnatura, incluso il tipo di ritorno. Un tentativo di ridefinire `m()` con la stessa lista di argomenti (`T1, ..., Tn`) ma con un tipo di ritorno diverso da `T` provoca un errore di compilazione. D'altra parte, rimane valida la regola che l'overriding di un metodo virtuale `m()` in una classe `D` derivata da `B` nasconde in `D` tutti gli ulteriori eventuali overloading di `m()` in `B`. Naturalmente, questo vale per un metodo qualsiasi, in particolare per gli operatori. Consideriamo il seguente esempio di gerarchia:

```
class B {
public:
    virtual int f() {cout << "B::f()\n"; return 1;}
```

6.5 Metodi virtuali

```
virtual void f(string s) {cout << "B::f(string)\n";}
virtual void g() {cout << "B::g()\n";}
};

class D1 : public B {
public:
    // Overriding di un metodo virtuale non sovraccaricato
    void g() {cout << "D1::g()\n";}
};

class D2 : public B {
public:
    // Overriding di un metodo virtuale sovraccaricato
    int f() {cout << "D2::f()\n"; return 2;}
};

class D3 : public B {
public:
    // NON è possibile modificare il tipo di ritorno
    void f() {cout << "D3::f()\n"; // Illegale}
};

class D4 : public B {
public:
    // Lista degli argomenti modificata:
    // è una ridefinizione e non un overriding
    int f(int) {cout << "D4::f()\n"; return 4;}
};
```

Consideriamo quindi il seguente esempio di codice che usa le classi della precedente gerarchia

```
int main() {
    string s="ciao"; D1 d1; D2 d2; D4 d4;
    int x = d1.f(); // Stampa: B::f()
    d1.f(s); // Stampa: B::f(string)
    x = d2.f(); // Stampa: D2::f()
    d2.f(s); // Illegale: "no matching function"
    x = d4.f(1); // Stampa: D4::f()
    x = d4.f(); // Illegale: "no matching function"
    d4.f(s); // Illegale: "no matching function"
    B& br = d4; // Cast implicito
    br.f(1); // Illegale
    br.f(); // OK, stampa: B::f()
    br.f(s); // OK, stampa: B::f(string)
}
```

È ammessa un'unica eccezione alla regola della preservazione della segnatura nell'overriding:

6 EREDITARIETÀ

verriding nel caso in cui il tipo di ritorno sia un tipo puntatore o riferimento ad una classe: se la segnatura del metodo virtuale `m()` è

```
virtual X* m(T1, ..., Tn);
```

dove `X` è un tipo classe, allora se `Y` è una sottoclasse di `X` è permesso che l'overriding di `m()` possa cambiare il tipo di ritorno in `Y*` mentre deve sempre mantenere la stessa lista dei parametri. In questo caso si dice che i tipi di ritorno delle funzioni sono *covarianti*. Questa eccezione è permessa perché l'overriding

```
virtual Y* m(T1, ..., Tn);
```

ritorna un puntatore a `Y` sottoclasse di `X` e quindi quel puntatore può essere convertito implicitamente in un puntatore a `X`. Una regola del tutto analoga vale per i riferimenti. A volte questa possibilità si può rivelare utile, come illustra il seguente esempio.

```
class X {};
class Y: public X {};
class Z: public X {};

class B {
    X x;
public:
    virtual X* m() { cout << "B::m() "; return &x; }
};
class C: public B {
    Y y;
public:
    virtual X* m() { return &y; }
};

class D: public B {
    Z z;
public:
    virtual Z* m() { return &z; } // OK, overriding legale
};

int main() {
    C c; D d;
    Y* py = c.m(); // Illegale
    X* px = c.m();
    Z* pz = d.m();
}
```

In questo esempio, l'overriding di `m()` in `C` mantiene la stessa segnatura mentre l'overriding di `m()` in `D` modifica il tipo di ritorno da `X*` a `Z*`. Notiamo inoltre che l'overriding `C::m()` ritorna un puntatore ad un oggetto della classe `Y` sfruttando la conversione隐式 da `Y*`

6.5 Metodi virtuali

a `x*` mentre `D::m()` ritorna un puntatore ad un oggetto della classe `Z` e non necessita di alcuna conversione implicita grazie al cambiamento del tipo di ritorno. Osserviamo quindi nel `main()` che riusciamo ad accedere al campo dati di tipo `Z` dell'oggetto `d` grazie al puntatore ritornato dall'invocazione `d.m()`. D'altra parte, non è invece possibile ottenere l'accesso al campo dati di tipo `Y` dell'oggetto `c` perché l'overriding `C::m()` ritorna solamente un puntatore a `X`. In quest'ultimo caso l'unico modo per ottenere l'accesso al campo dati di tipo `Y` sarà quello di sfruttare un cosiddetto downcast dinamico che sarà esaminato nella Sezione 6.6.

Bisogna prestare attenzione all'overriding di metodi virtuali che hanno parametri che prevedono valori di default perché si potrebbe essere tratti in inganno. Infatti, nell'overriding di un metodo virtuale con valori di default il linguaggio non prevede che la segnatura del metodo debba necessariamente ripetere i valori di default, che quindi possono essere omessi. Potrebbe quindi sembrare che un tale overriding di metodo che omette i valori di default non sia effettivamente un overriding del metodo virtuale ma invece un nuovo metodo della classe derivata, ma così invece non è. Si considerino infatti i seguenti esempi.

```
class B {
public:
    virtual void m(int x=0) { cout << "B::m ";}
};

class D: public B {
public:
    // è un overriding di B::m
    virtual void m(int x) { cout << "D::m ";}

    // Legale: è un nuovo metodo in D e non un overriding di B::m
    virtual void m() { cout << "D::m() ";}
};

int main() {
    B* p = new D;
    D* q = new D;
    p->m(2); // Stampa D::m e non B::m
    p->m(); // Stampa D::m e non D::m()
    q->m(); // Stampa D::m() e non D::m
}
```

```
class B {
public:
    virtual void m(int x=0) { cout << "B::m ";}
};

class D: public B {
```

6 EREDITARIETÀ

```
public:  
    // è un overriding di B::m  
    virtual void m(int x=0) { cout << "D::m " ; }  
  
    // Legale: è un nuovo metodo in D e non un overriding di B::m  
    virtual void m() { cout << "D::m() " ; }  
};  
  
int main() {  
    B* p = new D;  
    D* q = new D;  
    p->m(2); // Stampa D::m e non B::m  
    p->m(); // Stampa D::m e non D::m()  
    q->m(); // Illegale: chiamata ambigua di m() sovraccaricato  
}
```

È possibile bloccare il late binding di un metodo virtuale tramite l'operatore di scoping. Consideriamo la seguente situazione.

```
class B {  
public:  
    virtual void m() { cout << "B::m() " ; }  
};  
  
class C: public B {  
public:  
    virtual void m() { cout << "C::m() " ; }  
};  
  
class D: public C {  
public:  
    virtual void m() { cout << "D::m() " ; }  
};  
  
int main() {  
    C* p = new D();  
    p->m(); // Dynamic binding, stampa: D::m()  
    p->B::m(); // Static binding, stampa: B::m()  
    p->C::m(); // Static binding, stampa: C::m()  
}
```

Il puntatore *p* di tipo statico *C** ha tipo dinamico *D** e quindi, per late binding, la chiamata *p->m()* provoca l'invocazione dell'overriding *D::m()*. È comunque possibile invocare

tramite un legame statico il metodo `m()` definito in `B` o in `C` usando il corrispondente operatore di scoping `B::m()` o `C::m()`. Ad esempio, l'invocazione `p->C::m()` è staticamente compilata in una chiamata al metodo `m()` definito in `C` evitando in tal modo il late binding a run-time. Questa possibilità di bloccare il meccanismo di legame dinamico tra oggetto di invocazione e metodo virtuale si può rivelare utile in contesti ove sorga la necessità di invocare un particolare overriding di qualche metodo virtuale.

Abbiamo quindi visto che l'invocazione di un metodo virtuale tramite un puntatore polimorfo — ci riferiamo in questo caso ad una *chiamata polimorfa* di un metodo — provoca effetti diversi a seconda del tipo dinamico del puntatore, ovvero del tipo effettivo dell'oggetto a cui punta il puntatore. Analogamente per i riferimenti. Spesso si usa il termine polimorfismo anche per riferirsi in generale a questa caratteristica fondamentale della programmazione orientata agli oggetti. In questo senso, il polimorfismo promuove l'*estensibilità* del software: i programmi che sfruttano il polimorfismo hanno un comportamento in qualche modo indipendente dal tipo statico dei puntatori e riferimenti che invocano metodi virtuali. Si possono aggiungere incrementalmente ad un sistema software polimorfo nuovi tipi di oggetti che reimplementano ogni volta in modo diverso e caratteristico del particolare tipo le funzionalità virtuali del sistema software.

Si pensi ad esempio ad uno screen manager, ovvero un modulo software che si occupa di gestire la politica di piazzamento di vari “oggetti” da visualizzare su uno schermo. Uno screen manager dovrà essere in grado di gestire una varietà di oggetti di tipi diversi, possibilmente di tipi che saranno aggiunti al sistema software globale anche dopo la sua scrittura. Lo screen manager potrà usare puntatori e riferimenti ad una classe base `Shape` per gestire tutti gli oggetti da visualizzare: `Shape` conterrà un metodo virtuale `draw()` e per visualizzare un oggetto lo screen manager semplicemente invocherà `p->draw()` con `p` puntatore alla classe base `Shape`. Il metodo `draw()` sarà quindi ridefinito in tutte le sottoclassi di `Shape`: ogni particolare sottoclasse saprà come visualizzare un proprio oggetto. Lo screen manager non si preoccupa del particolare tipo di oggetto da visualizzare, semplicemente ordina all’oggetto di visualizzarsi, chiunque esso sia.

Esempio 6.5.1. Riconsideriamo l'Esempio 6.3.6 della gerarchia di classi che ha come classe base `poligono`. Dichiariamo virtuale il metodo `perimetro()` che calcola il perimetro nella classe base `poligono`. La classe `triangolo`, derivata da `poligono`, non ha una propria versione di tale metodo, per cui eredita la versione definita in `poligono`. Invece, la classe `tri_rettangolo`, derivata da `triangolo`, ha una propria versione di `perimetro()`. Il metodo `area()` che calcola l'area è dichiarato virtuale in `triangolo` ed ha una versione specializzata in `tri_rettangolo`.

```
class punto {
private:
    double x, y;
public:
    double getX() const;
    double getY() const;
```

6 EREDITARIETÀ

```
    static double lung(const punto& p1, const punto& p2);  
  
class poligono {  
protected:  
    int nvertici;  
    punto* pp;  
public:  
    poligono(int n, const punto v[]);  
    ~poligono();  
    poligono(const poligono& pol);  
    poligono& operator=(const poligono& pol);  
    virtual double perimetro() const; // metodo virtuale  
};
```

```
class triangolo: public poligono {  
public:  
    triangolo(const punto v[]) : poligono(3, v) {}  
    virtual double area() const; // nuovo metodo virtuale  
};  
  
double triangolo::area() const { // usa la formula di Erone  
    double a = punto::lung(pp[1], pp[0]);  
    double b = punto::lung(pp[2], pp[1]);  
    double c = punto::lung(pp[0], pp[2]);  
    double p = (a + b + c)/2;  
    return sqrt(p*(p-a)*(p-b)*(p-c));  
}
```

```
class tri_rettangolo: public triangolo {  
public: // triangolo rettangolo in pp[0]  
    tri_rettangolo(const punto v[]) : triangolo(v) {}  
    virtual double perimetro() const; // overriding specializzato  
    virtual double area() const; // overriding specializzato  
};  
  
double tri_rettangolo::perimetro() const {  
    double a=punto::lung(pp[1],pp[0]), b=punto::lung(pp[0],pp[2]);  
    return (a + b + sqrt(a*a + b*b));  
}  
double tri_rettangolo::area() const {  
    double a=punto::lung(pp[1],pp[0]), b=punto::lung(pp[0],pp[2]);  
    return (a*b / 2);  
}
```

6.5 Metodi virtuali

Possiamo quindi scrivere delle funzioni che contengono delle chiamate polimorfe come nel seguente esempio.

```
void stampa_perimetro(poligono* p) {
    cout << "Il perimetro è " <<
        p->perimetro() << endl; // chiamata polimorfa
}

void stampa_area_triangolo(const triangolo& p) {
    cout << "L'area è " << p.area() << endl; // chiamata polimorfa
}
```

6.5.1 vtable

Il late binding implica un overhead dinamico a run-time in termini di tempo e spazio. L'impatto di tale costo aggiuntivo eventualmente potrà essere valutato nel particolare contesto applicativo, nonostante nelle moderne architetture in generale sia pressoché trascurabile. Il seguente semplice esempio mostra praticamente tale overhead (i tempi sono riferiti ad una architettura basata su Intel Core i7 2.66 GHz).

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

class D: public B {
public:
    void f() /* overriding */
};

int main() {
    B* p = new D;
    long int i=0;
    // late binding
    for (; i<1000000000; ++i) p->f(); // 2.845 sec
    // static binding
    for (; i<1000000000; ++i) p->B::f(); // 2.512 sec
}
```

L'implementazione del late binding viene usualmente studiata in un corso di Linguaggi di Programmazione. Qui accenniamo solamente al fatto che per ogni classe C che contiene almeno un metodo virtuale il compilatore crea anche una corrispondente tabella contenente gli indirizzi dei metodi virtuali di C (quindi si tratta di puntatori a funzioni) detta *vtable*.

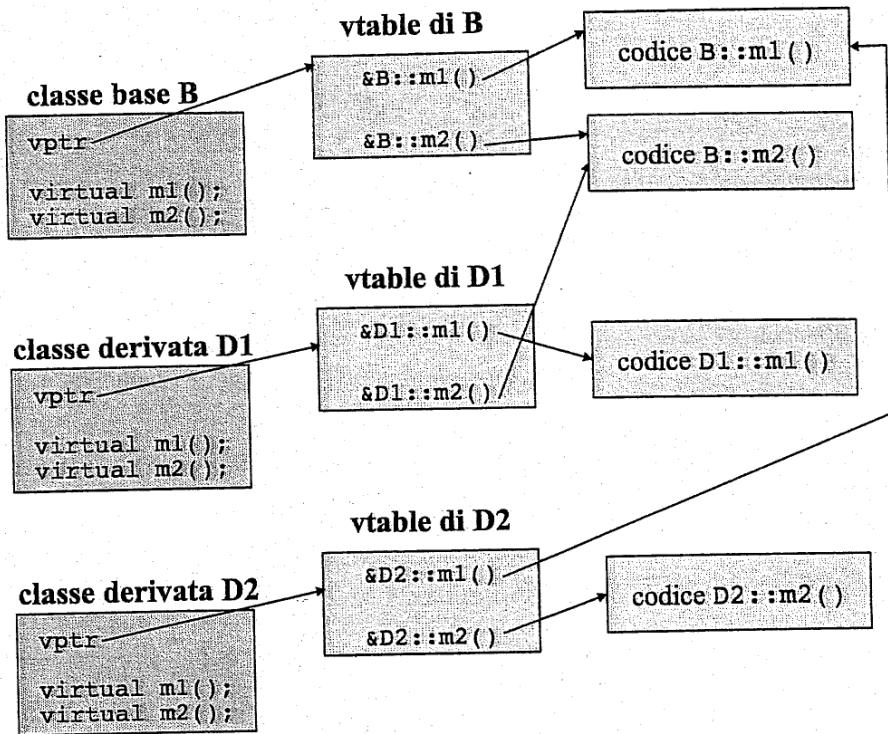
6 EREDITARIETÀ

(acronimo di *virtual table*). Inoltre, per ogni oggetto di tale classe C il compilatore "include" in quell'oggetto un puntatore a funzione (detto *vpointer*) alla vtable di C. La selezione a run-time di quale metodo invocare in una chiamata polimorfa `p->m()` avviene seguendo tali strutture aggiuntive di puntatori.

Ad esempio, per la seguente gerarchia di classi:

```
class B {  
public:  
    FunctionPointer* vptr; // vpointer aggiunto dal compilatore  
    virtual void m1() {}  
    virtual void m2() {}  
};  
  
class D1: public B {  
public:  
    virtual void m1() {} // overriding  
};  
  
class D2: public B {  
public:  
    virtual void m2() {} // overriding  
};
```

i virtual pointers e le virtual tables della classi nella gerarchia si possono schematizzare mediante la seguente figura.



6.5.2 Distruttori virtuali

Consideriamo il seguente frammento di codice in cui B è una classe base e D è una classe derivata da B:

```
D* pd = new D;
B* pb = pd; // TD(pb)=D*
delete pb;
```

In questa situazione `delete pb`; richiama il distruttore della classe base B su un oggetto della classe derivata D. Si può evitare ciò dichiarando virtuale il distruttore della classe base B: in questo modo, tutti i distruttori delle classi derivate da B diventano automaticamente virtuali ottenendo quindi l'effetto che quando viene applicato l'operatore `delete` ad un puntatore alla classe base B il cui tipo dinamico è D*, per qualche sottoclass D di B, viene effettivamente invocato il distruttore di D. Se il distruttore di B è virtuale allora tutti i distruttori delle classi derivate da B diventano automaticamente virtuali. Per una classe che contiene metodi virtuali è prassi e buona norma includere un distruttore virtuale, anche se per tale classe non dovesse essere strettamente necessario. Spesso si dichiara il distruttore virtuale e con corpo vuoto, rendendo quindi virtuale il distruttore standard. Per i costruttori, invece, non ha senso (e non è quindi possibile) la dichiarazione virtuale. Consideriamo il seguente esempio.

```

class B {
private:
    int* p;
public:
    B(int n, int v) : p(new int[n]) {
        for(int i=0; i<n; i++) p[i]=v;
    }
    virtual ~B() {delete[] p; cout <<"~B() ";} // distr. virtuale
};

class C : public B {
private:
    int* q;
public:
    C(int sizeB, int sizeC, int v): B(sizeB,v), q(new int[sizeC]) {
        for(int i=0; i<sizeC; i++) q[i]=v;
    }
    virtual ~C() {delete[] q; cout <<"~C() ";}
};

int main() {
    C* q = new C(4,2,18);
    B* p=q; // puntatore polimorfo
    delete p; // distruzione virtuale: invoca ~C()
}
// Stampa: ~C() ~B()
// Se ~B() non fosse virtuale verrebbe invocato solamente ~B()

```

Il meccanismo del late binding non funziona nel corpo dei costruttori, ovvero, se nel corpo di un costruttore di una classe B che contiene un metodo virtuale `m()` appare una invocazione di `m()`, allora quella invocazione si riferisce sempre alla versione di `m()` locale di B. Il meccanismo del late binding non funziona nemmeno nel corpo dei distruttori, siano essi virtuali o meno, cioè nel corpo di un distruttore di una classe B le chiamate ad un metodo virtuale `m()` di B sono sempre statiche e mai polimorfiche. In entrambi i casi le motivazioni di queste restrizioni sono dovute al meccanismo di implementazione del late binding.

6.5.3 Metodi virtuali puri

In alcune circostanze una classe base B viene progettata solamente allo scopo di poter successivamente definire delle classi derivate da B e non per creare e usare oggetti di B. In questa situazione una tale classe base B funziona da “interfaccia comune” per le classi derivate da B, ovvero B si limita a specificare la lista delle operazioni basilari che caratterizzano le classi derivate.

6.5 Metodi virtuali

rizzano tutti i sottotipi di *B*, operazioni che in generale verranno realizzate in modo diverso da sottotipo a sottotipo.

Un esempio di classe base “interfaccia” potrebbe essere la classe *Shape* precedentemente considerata nel contesto di uno screen manager, che specifica l’esistenza di un metodo virtuale *draw()* che le classi derivate da *Shape* (ad esempio, *Window*, *Icon*, *TaskBar*, etc.) implementano in diversi modi specifici. Consideriamo un ulteriore esempio: una classe base interfaccia *UnitaDiOutput* e classi derivate come *Videol*, *Stampante1*, *video2*, *Stampante2*, etc. In questo caso è inutile, e spesso impossibile per mancanza di sufficienti dettagli rappresentativi, scrivere il codice dei metodi virtuali della classe base in quanto essi non verranno mai usati. Come la maggior parte dei linguaggi ad oggetti, il C++ permette quindi di dichiarare un metodo virtuale senza definirne il corpo. Per avvisare il compilatore che il metodo non ha corpo basta aggiungere il marcitore “=0” alla fine della sua dichiarazione. In questo caso il metodo viene detto *metodo virtuale puro* (o funzione virtuale pura).

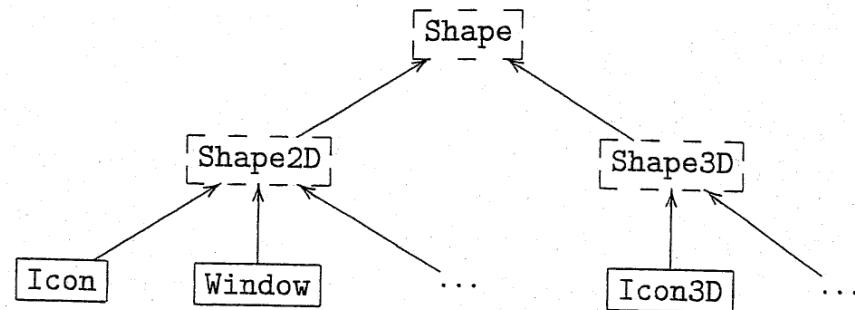
Il metodo *G()* è dichiarato virtuale puro nella seguente classe *B* e quindi non deve essere definito.

```
class B {  
...  
    virtual void G() = 0;  
...  
};
```

Quando una classe *B* contiene o eredita senza definirlo almeno un metodo virtuale puro, *B* viene detta *classe (base) astratta*. Il compilatore segnalerà un errore di compilazione in un tentativo di istanziare oggetti di una classe base astratta. Una sottoclasse *D* di una classe base astratta *B* si dice *concreta* se *D* implementa tutti i metodi virtuali puri di *B*. Gli oggetti di una classe base astratta *B* possono essere creati soltanto come sottooggetti di oggetti appartenenti ad una sottoclasse concreta di *B*. Una classe astratta può comunque contenere dei costruttori, anzi tipicamente li contiene. Inoltre, è sempre possibile dichiarare puntatori e riferimenti a classi astratte.

Una gerarchia di classi non deve necessariamente contenere delle classi astratte, ma spesso succede che nella fase di progettazione di una gerarchia si ravvisi l’opportunità di definire delle classi astratte nei primi livelli della gerarchia (quelli “più alti”). Ad esempio, considerando ancora l’esempio dello screen manager, la gerarchia potrebbe partire dalla classe base astratta *Shape* — le classi astratte vengono graficamente rappresentate con un contorno punteggiato — e potrebbe avere due sottoclassi di *Shape* ancora astratte come *Shape2D* e *Shape3D*, mentre dal terzo livello si potrebbe cominciare a definire delle sottoclassi concrete.

6 EREDITARIETÀ



Consideriamo il seguente esempio:

```
class B { // classe base astratta
public:
    virtual void f() = 0;
};

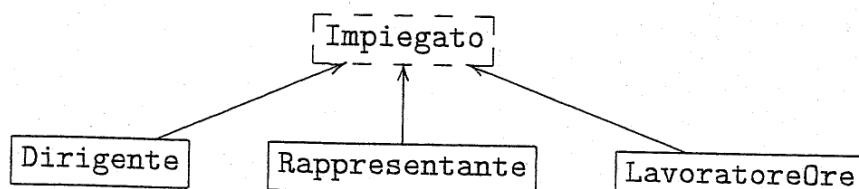
class C: public B { // sottoclasse astratta

};

class D: public B { // sottoclasse concreta
public:
    virtual void f() {cout << "D::f() ";}
};

int main() {
    C c;      // Illegale: "cannot declare c of type C ..."
    D d;      // OK, D è concreta
    B* p;    // OK, puntatore a classe astratta
    p = &d; // Puntatore polimorfo
    p->f(); // Stampa: D::f()
}
```

Esempio 6.5.2. In questo esempio definiamo una classe astratta `Impiegato` da cui derivano tre classi concrete.



```
class Impiegato { // classe base astratta
public:
    Impiegato(string s): nome(s) {}
    string getNome() const {return nome;}
    virtual double stipendio() const = 0; // virtuale puro
};
```

6.5 Metodi virtuali

```
virtual void print() const {cout << nome;} // virtuale  
private:  
    string nome;  
};
```

```
class Dirigente : public Impiegato {  
public:  
    Dirigente(string s, double d=0): Impiegato(s), fissoMensile(d){}  
    void setFissoMensile(double d) { // metodo non costante  
        fissoMensile = d > 0 ? d : 0;  
    }  
    virtual double stipendio() const { // implementazione  
        return fissoMensile;  
    }  
    virtual void print() const { // overriding  
        cout << "Il dirigente "; Impiegato::print();  
        // NB: invocazione statica di Impiegato::print()  
    }  
private:  
    double fissoMensile; // stipendio fisso  
};
```

```
class Rappresentante : public Impiegato {  
public:  
    Rappresentante(string s, double d=0, double e=0, int x=0):  
        Impiegato(s), baseMensile(d), commissione(e), tot(x) {}  
    void setBase(double d) { baseMensile = d > 0 ? d : 0; }  
    void setCommissione(double d) { commissione = d > 0 ? d : 0; }  
    void setVenduti(int x) { tot = x > 0 ? x : 0; }  
    virtual double stipendio() const { // implementazione  
        return baseMensile + commissione*tot;  
    }  
    virtual void print() const { // overriding  
        cout << "Il rappresentante "; Impiegato::print();  
    };  
private:  
    double baseMensile; // stipendio base fisso  
    double commissione; // commissione per pezzo venduto  
    int tot; // pezzi venduti in un mese  
};
```

```
class LavoratoreOre : public Impiegato {  
public:  
    LavoratoreOre(string s, double d=0, double e=0):
```

6 EREDITARIETÀ

```
Impiegato(s), pagaOraria(d), oreLavorate(e) {}

void setPaga(double d) { pagaOraria = d > 0 ? d : 0; }
void setOre(double d) {
    oreLavorate = d >= 0 && d <= 250 ? d : 0;
}
virtual double stipendio() const { // implementazione
    if ( oreLavorate <= 160 ) // nessuno straordinario
        return pagaOraria*oreLavorate;
    else // le ore straordinarie sono pagate il doppio
        return 160*pagaOraria+(oreLavorate-160)*2*pagaOraria;
};
virtual void print() const { // overriding
    cout << "Il lavoratore a ore "; Impiegato::print();
};

private:
    double pagaOraria;
    double oreLavorate; // ore lavorate nel mese
};
```

Possiamo quindi definire la seguente funzione con un parametro puntatore ad `Impiegato` che contiene due chiamate polimorfe.

```
void stampaStipendio(Impiegato* p) {
    p->print(); // chiamata polimorfa
    cout << " in questo mese ha guadagnato "
        << p->stipendio() << " Euro.\n"; // chiamata polimorfa
}
```

Ad esempio, il seguente `main()` provoca le stampe indicate.

```
int main() {
    Dirigente d("Paperino", 4000);
    Rappresentante r("Topolino", 1000, 3, 250);
    LavoratoreOre l("Pluto", 15, 170);
    stampaStipendio(&b); stampaStipendio(&r); stampaStipendio(&l);

}

/* STAMPA:
Il dirigente Paperino in questo mese ha guadagnato 4000 Euro.
L'impiegato Topolino in questo mese ha guadagnato 1750 Euro.
Il lavoratore a ore Pluto in questo mese ha guadagnato 2700 Euro.
*/
```

6.6 Identificazione di tipi a run-time

Il meccanismo dell'*identificazione di tipi a run-time* (RTTI, acronimo di Run Time Type Identification) permette di determinare il tipo dinamico di un puntatore o di un riferimento a tempo di esecuzione. I due principali operatori di RTTI sono `typeid` e `dynamic_cast`. L'operatore `typeid` è definito nel file header di libreria `typeinfo` che deve quindi essere incluso.

L'operatore `typeid` permette di determinare il tipo di una espressione qualsiasi a tempo di esecuzione. Se l'espressione è un riferimento polimorfo o un puntatore polimorfo dereferenziato allora `typeid` ritorna il tipo dinamico di quella espressione.

```
#include <typeinfo>
#include <iostream>
using std::cout; using std::endl;

int main() {
    int i=5;
    cout << typeid(i).name() << endl;           // Stampa: i(nt)
    cout << typeid(3.14).name() << endl;          // Stampa: d(double)
    if (typeid(i) == typeid(int)) cout << "Yes";
}
```

L'operatore `typeid` ha come argomento un'espressione o un tipo qualsiasi e ritorna un oggetto della classe `type_info`. La definizione della classe `type_info` è nel file header `typeinfo` ed ogni implementazione include almeno i seguenti metodi:

```
class type_info {
// rappresentazione dipendente dall'implementazione
private:
    type_info();
    type_info(const type_info&);
    type_info& operator=(const type_info&);
public:
    bool operator==(const type_info&) const;
    bool operator!=(const type_info&) const;
    const char* name() const;
};
```

Non è possibile dichiarare, modificare o assegnare oggetti di tipo `type_info`, dal momento che l'unico costruttore (quello di default) è privato così come lo sono il costruttore di copia e l'assegnazione. È possibile soltanto confrontare oggetti `type_info` con gli operatori di uguaglianza e disuguaglianza ed estrarne il loro "nome" tramite il metodo `name()`.

`typeid` è un operatore di RTTI caratterizzato dal seguente comportamento:

6 EREDITARIETÀ

- Se in una invocazione `typeid(ref)` l'espressione `ref` operando di `typeid` è un riferimento ad una classe che contiene almeno un metodo virtuale allora `typeid(ref)` restituisce un oggetto di `type_info` che rappresenta il tipo dinamico di `ref`.
- Se in una invocazione `typeid(*punt)` l'espressione `*punt` operando di `typeid` è un puntatore dereferenziato tale che `punt` è un puntatore ad una classe che contiene almeno un metodo virtuale allora `typeid` restituisce un oggetto di `type_info` che rappresenta il tipo `T` dove `T*` è il tipo dinamico di `punt`.

Diremo che un tipo `T` è polimorfo se `T` è un tipo classe che include tra i suoi almeno un metodo virtuale. Si noti che se `C` è una classe che contiene solamente il distruttore virtuale allora `C` è una classe polimorfa. Inoltre ogni sottotipo di un tipo polimorfo è a sua volta un tipo polimorfo. Quindi ogni classe derivata da una classe polimorfa è pure polimorfa.

Bisogna prestare attenzione alle seguenti tre regole:

- (a) Se la classe non è polimorfa, cioè non contiene metodi virtuali, allora `typeid` restituisce il tipo statico del riferimento o del puntatore dereferenziato.
- (b) `typeid` su un puntatore non dereferenziato restituisce sempre il tipo statico del puntatore.
- (c) `typeid` ignora sempre l'attributo di tipo `const`: ad esempio, l'espressione booleana `typeid(T) == typeid(const T)` si valuta sempre a `true`.

L'operatore di conversione esplicita `dynamic_cast` permette di convertire puntatori e riferimenti ad una classe base `B` polimorfa in puntatori e riferimenti ad una classe `D` derivata da `B`. Supponiamo che `p` sia un puntatore ad una classe polimorfa `B` che punta a qualche oggetto `obj` e che `D` sia una sottoclasse di `B`. L'operatore di conversione dinamica `dynamic_cast<D*>(p)` permette di tentare la conversione di `p` al tipo target `D*`. La conversione sarà possibile o meno a seconda del tipo dinamico del puntatore `p`. Supponiamo che `TD(p) = E*`:

1. Se `E` è sottotipo di `D` allora la conversione andrà a buon fine e `dynamic_cast<D*>(p)` ritirerà un puntatore di tipo `D*` all'oggetto `obj`. Quindi in questo caso la conversione ha successo perchè il tipo dinamico `E*` di `p` è compatibile con il tipo target `D*` della conversione (cioè `E*` è sottotipo di `D*`): ricordiamo infatti che il tipo statico di un puntatore deve sempre essere un supertipo del tipo dinamico.
2. Se invece `E` non è un sottotipo di `D` allora la conversione fallirà e `dynamic_cast<D*>(p)` ritirerà il puntatore nullo `0`. In questo caso si dice che il tipo dinamico del puntatore non è compatibile con il tipo target della conversione.

È importante ricordare che, analogamente a `typeid`, la classe `B` deve essere polimorfa, altrimenti la compilazione del `dynamic_cast` provocherà un errore.

Il `dynamic_cast` permette quindi di effettuare esplicitamente le conversioni da (puntatori e riferimenti a) classe base a (puntatori e riferimenti a) classe derivata

6.6 Identificazione di tipi a run-time

$B^* \Rightarrow D^*$ e $B& \Rightarrow D&$

in modo controllato, ovvero con la possibilità di gestire a run-time un errore di conversione. Si parla in questo caso di conversioni *safe* (o *safe cast*). Questa conversione safe esplicita viene anche detta *safe downcasting* — perché si converte “dall’alto verso il basso” nella gerarchia, ovvero da classe base a classe derivata — in opposizione alla conversione implicita.

$D^* \Rightarrow B^*$ e $D& \Rightarrow B&$

che viene detta *upcasting* — si converte “dal basso verso l’alto” nella gerarchia, ovvero da classe derivata a classe base.

Diversamente dagli altri operatori di conversione del C++, il `dynamic_cast` è necessariamente eseguito a run-time, perché il successo della conversione dipende dal tipo dinamico del puntatore o riferimento che si vuole convertire. Abbiamo visto che se il cast dinamico di un puntatore fallisce allora il `dynamic_cast` ritorna il puntatore nullo. Invece, nel caso dei riferimenti, se il `dynamic_cast` di un riferimento fallisce allora viene automaticamente lanciata un’eccezione di tipo `bad_cast` (definito nel file header `typeinfo`). Vediamo un esempio:

```
class X { public: virtual ~X() {} };
class B { public: virtual ~B() {} };
class D : public B {};

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

int main() {
    D d; B& b = d; // upcast
    try { X& xr = dynamic_cast<X&>(b); }
    catch(bad_cast e) { cout << "Fallimento!" << endl; }
}
```

In generale il safe downcasting va usato solamente in caso di necessità. Tipicamente, si ha la necessità di fare un downcasting di un puntatore ad una classe base `B` per ottenere la disponibilità dei membri di una classe derivata da `B` che non sono stati ereditati da `B`. Non si deve cadere nella tentazione di rimpiazzare le chiamate polimorfe automatiche ai metodi virtuali con delle istruzioni condizionali (ad esempio degli `switch`) che usano il `dynamic_cast` per effettuare type checking dinamico. Nella maggior parte dei casi una tale pratica va contro l’estensibilità del codice. Si tratta di un errore comune del principiante della programmazione ad oggetti che spesso denota difficoltà di comprensione del meccanismo delle chiamate polimorfe. Vediamo un tipico esempio d’uso del `dynamic_cast` per effettuare safe downcasting.

```
class impiegato { // classe astratta
```

6 EREDITARIETÀ

```
public:  
    virtual double stipendioBase() const = 0;  
};  
  
class manager : public impiegato {  
public:  
    virtual double stipendioBase() const { return 2000; }  
};  
  
class programmatore : public impiegato {  
public:  
    virtual double stipendioBase() const { return 1500; }  
    // metodo specifico della classe programmatore  
    virtual double bonus() const { return 300; }  
};  
  
class softwareHouse {  
public:  
    static double stipendio(const impiegato& p) {  
        const programmatore* q =  
            dynamic_cast<const programmatore*>(&p);  
        // se il cast ha successo, cioè  
        // se TD(&p) è sottotipo di programmatore*  
        if(q) return q->stipendioBase() + q->bonus();  
        // se q==0, cioè TD(&p) non è sottotipo di programmatore*  
        else return p.stipendioBase();  
    }  
};  
  
int main() {  
    manager m; programmatore p;  
    cout << "Il manager guadagna " << softwareHouse::stipendio(m)  
                                << " Euro" << endl;  
    cout << "Il programmatore guadagna " <<  
        softwareHouse::stipendio(p) << " Euro" << endl;  
}
```

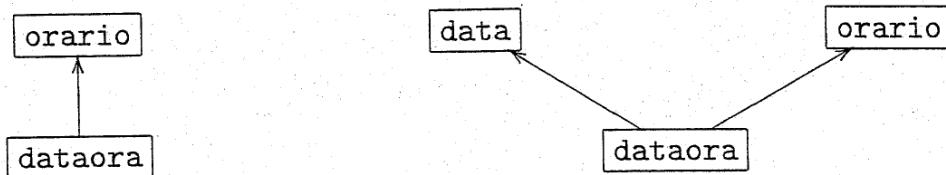
6.7 Ereditarietà multipla

Una classe può essere derivata da più di una classe base. Si parla in questo caso di *ereditarietà multipla*. Con l'ereditarietà multipla ogni oggetto della classe derivata contiene un sottooggetto per ognuna delle classi base da cui deriva.

Illustriamo la situazione con una definizione alternativa della classe dataora. Invece di

6.7 Ereditarietà multipla

definire la classe `dataora` come derivata dalla classe base `orario` e quindi aggiungendo dei campi dati e metodi che gestiscono la "data", possiamo invece definire una seconda classe base `data` i cui oggetti rappresentano una data e quindi definire la classe `dataora` come derivata sia dalla classe base `orario` che dalla classe base `data`.



Dichiaramo come segue la classe `data` e lasciamo per esercizio la definizione dei suoi metodi.

```
class data {  
public:  
    data(int =1, int =1, int =0);  
    int Giorno() const { return giorno; }  
    int Mese() const { return mese; }  
    int Anno() const { return anno; }  
protected:  
    int giorno, mese, anno;  
    void AvanzaUnGiorno();  
private:  
    int GiorniDelMese() const;  
    ~bool Bisestile() const;  
};
```

La definizione della classe `dataora` derivata sia dalla classe base `orario` che dalla classe base `data` è la seguente.

```
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;  
    ...  
};
```

Abbiamo usato la derivazione pubblica sia per la classe base `data` che per la classe base `orario`. In generale, la tipologia di derivazione può essere diversa per ogni classe base. Il costruttore di default `dataora()` definito esplicitamente con corpo vuoto richiama implicitamente i costruttori di default per i sottooggetti delle classi base nell'ordine in cui esse compaiono testualmente nella definizione della classe derivata `dataora`, cioè prima

6 EREDITARIETÀ

data e poi orario. Questo è anche il comportamento del costruttore di default standard. Nel nostro caso, esso è stato definito esplicitamente perché altrimenti sarebbe stato "nascosto" dalla presenza del costruttore con sei parametri. Il costruttore a sei parametri richiama esplicitamente i costruttori delle due classi base: data(a, me, g), orario(o, m, s). L'ordine in cui vengono eseguiti i costruttori delle classi base è comunque sempre quello in cui compaiono testualmente le classi base nella definizione della classe derivata. Quindi, per dataora, prima data(a, me, g) e successivamente orario(o, m, s).

Abbiamo visto che l'overriding in una classe D derivata direttamente da B di un metodo di nome m definito in B comporta che tale metodo m di B, e tutti gli eventuali metodi sovraccaricati di nome m in B, non siano visibili in D, e quindi per accedere a tali metodi è necessario ricorrere all'operatore di scoping B::.

Cosa succede invece se due metodi identificati dallo stesso nome sono definiti in due classi base diverse della stessa classe derivata? Ad esempio, supponiamo di aver aggiunto sia alla classe orario che alla classe data un metodo Stampa() che stampa, rispettivamente, un orario e una data.

```
void orario::Stampa() const {
    cout << Ore() << ':' << Minuti() << ':' << Secondi();
}
```

```
void data::Stampa() const {
    cout << Giorno() << '/' << Mese() << '/' << Anno();
}
```

Pertanto, la classe dataora eredita due funzioni diverse con lo stesso nome Stampa(). In questa situazione, le istruzioni

```
dataora d;
d.Stampa(); // Illegale
```

generano un errore in compilazione: a causa dell'*ambiguità* dell'invocazione il compilatore non è in grado di decidere quale dei due metodi usare. Si noti che, come al solito per gli errori di compilazione dovuti a situazioni di ambiguità, la generazione dell'errore avviene soltanto quando si cerca di usare tale metodo Stampa() su di un oggetto della classe derivata dataora (oppure indirettamente tramite un puntatore) e non a causa della definizione della classe derivata dataora stessa.

Nel caso precedente, i due metodi avevano la stessa segnatura. L'ambiguità rimane comunque anche se le segnature dei metodi nelle classi base dovessero essere diverse, come dimostra il seguente esempio:

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

6.7 Ereditarietà multipla

```
class B {
public:
    void f(int x) {cout << "B::f ";}
};

class D: public A, public B {

};

int main() {
    D d;
    d.f(); // Illegale: ambiguità
    d.f(2); // Illegale: ambiguità
}
```

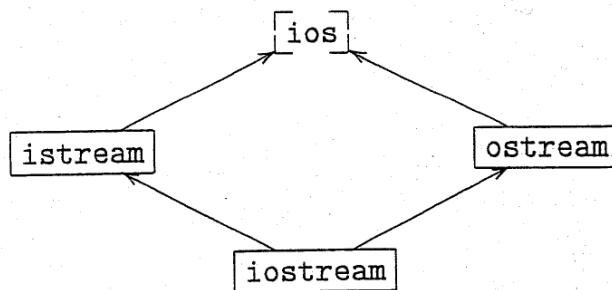
Possiamo risolvere esplicitamente l'ambiguità usando l'operatore di scoping:

```
dataora d;
d.data::Stampa();
// OPPURE
dataora d;
d.orario::Stampa();
```

Naturalmente, se avessimo ridefinito il metodo `Stampa()` nella classe `dataora` esso avrebbe nascosto entrambi i metodi `Stampa()` delle classi base e non ci sarebbe quindi stata alcuna ambiguità. Ad esempio, la seguente ridefinizione di `Stampa()` per la classe `dataora` usa l'operatore di scoping per richiamare le funzioni `Stampa()` delle classi base.

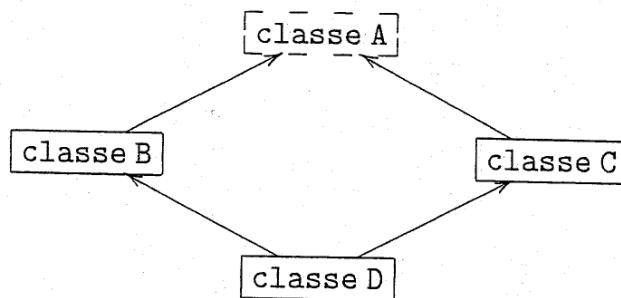
```
void dataora::Stampa() const {
    data::Stampa(); cout << ' ' ; orario::Stampa();
}
```

Un esempio di ereditarietà multipla è dato dalla classe `iostream` della gerarchia di classi di input/output della libreria standard che vedremo nel Capitolo 8. La classe `ios` è una classe base sia per la classe `ostream` che per la classe `istream`. La classe `iostream` deriva direttamente sia da `ostream` che da `istream`. Ciò consente agli oggetti di tipo `iostream` di fornire le funzionalità sia di `istream` che di `ostream`.



6.7.1 Derivazione virtuale

Nella programmazione OO una gerarchia di classi può essere molto complessa. In particolare, può accadere che una classe derivi da due classi base le quali a loro volta derivano, direttamente o indirettamente, da una stessa classe. Tale situazione viene anche detta *ereditarietà a diamante*. È proprio il caso precedentemente citato della classe `iostream`.

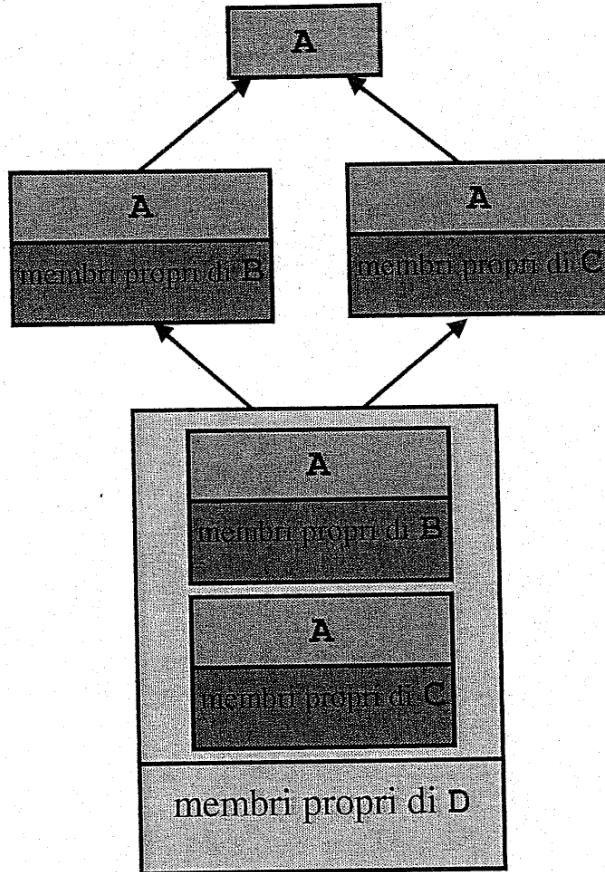


In questa situazione succede che ogni oggetto della classe D contiene due sottooggetti della classe A. Questo comporta i seguenti due problemi:

- produce una ambiguità che impedisce di accedere a tali sottooggetti ed ai metodi che operano su di essi;
- produce uno spreco di memoria.

La situazione è illustrata nella seguente figura.

6.7 Ereditarietà multipla



Esempio 6.7.1. Si consideri la seguente gerarchia di classi dove D è definita tramite derivazione multipla.

```
class A {  
public:  
    int a;  
    A(int x=1): a(x) {}  
};  
  
class B: public A {  
public:  
    B(): A(2) {}  
};  
  
class C: public A {  
public:  
    C(): A(3) {}  
};  
  
class D: public B, public C { };
```

6 EREDITARIETÀ

Il seguente frammento di codice non compila a causa dell'ambiguità generata dalla derivazione multipla.

```
D d;  
A* p = &d; // Illegale: A è una classe base ambigua per D  
cout << p->a; // quale sottooggetto di A si dovrebbe usare?
```

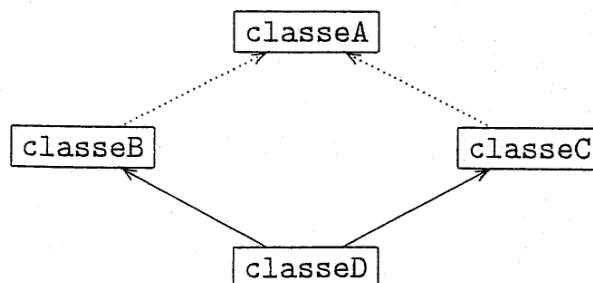
Si consideri ora la seguente gerarchia di classi.

```
class A {  
public:  
    virtual void print() =0; // virtuale puro  
};  
  
class B: public A {  
public:  
    virtual void print() {cout << "B ";} // implementazione  
};  
  
class C: public A {  
public:  
    virtual void print() {cout << "C ";} // implementazione  
};  
  
class D: public B, public C {  
    virtual void print() {cout << "D ";} // overriding  
};
```

In questo caso il seguente frammento di codice non compila ancora a causa dell'ambiguità dovuta all'ereditarietà multipla a diamante.

```
D d;  
A* p = &d; // Illegale: A è una classe base ambigua per D  
p->print(); // la chiamata polimorfa non è legale
```

Una soluzione migliore sarebbe quella di avere un unico sottooggetto di classe A in ogni oggetto della classe D:



6.7 Ereditarietà multipla

Questo si può ottenere utilizzando la *derivazione virtuale*, che nelle figure che rappresentano gerarchie di classi indicheremo con frecce tratteggiate. La derivazione virtuale si ottiene tramite la keyword **virtual** che è quindi sovraccaricata da un doppio significato. Le dichiarazioni delle classi A, B, C e D diventano le seguenti:

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

Si dice che A è una *classe base virtuale* (per le classi B e C). Per ogni classe D derivata da A e definita tramite derivazione multipla il fatto che A sia una classe base virtuale garantisce che ci sarà un solo sottooggetto di A in ogni oggetto della classe D. La derivazione virtuale. Lo standard del linguaggio non prevede come debba essere implementata la derivazione virtuale. Nella pratica essa è implementata tramite dei puntatori in un modello che possiamo astrattamente (cioè ad alto livello) descrivere nel seguente modo. Gli oggetti delle classi B e C che derivano virtualmente da A oltre al sottooggetto "ordinario" della classe A contegno un puntatore ad un oggetto della classe A. Gli oggetti della classe D conterranno invece un puntatore ad un unico sottooggetto di tipo A.

Esempio 6.7.2. Si consideri la seguente gerarchia di classi a diamante definita senza derivazione virtuale.

```
class A { int a[5]; }; // 20 byte  
  
class B: public A { int y[3]; }; // 12 byte  
  
class C : public A { int z[3]; }; // 12 byte  
  
class D: public B, public C {  
    int w[4]; // 16 byte  
};
```

Il seguente codice, che usa la funzione `sizeof()`, stampa la dimensione in byte degli oggetti delle classi nella gerarchia (in qualche ambiente di programmazione C++). In particolare, abbiamo che `sizeof(D)=sizeof(B)+sizeof(C)`.

6 EREDITARIETÀ

```
cout << "sizeof(A) == " << sizeof(A) << endl; // 20
cout << "sizeof(B) == " << sizeof(B) << endl; // 32=12+20
cout << "sizeof(C) == " << sizeof(C) << endl; // 32=12+20
cout << "sizeof(D) == " << sizeof(D);           // 80=16+32+32
```

Consideriamo ora la stessa gerarchia di classi definita con derivazione virtuale.

```
class A {int a[5];} // 20 byte

class B: virtual public A {
    int y[3];          // 12 byte + 1 puntatore
};

class C : virtual public A {
    int z[3];          // 12 byte + 1 puntatore
};

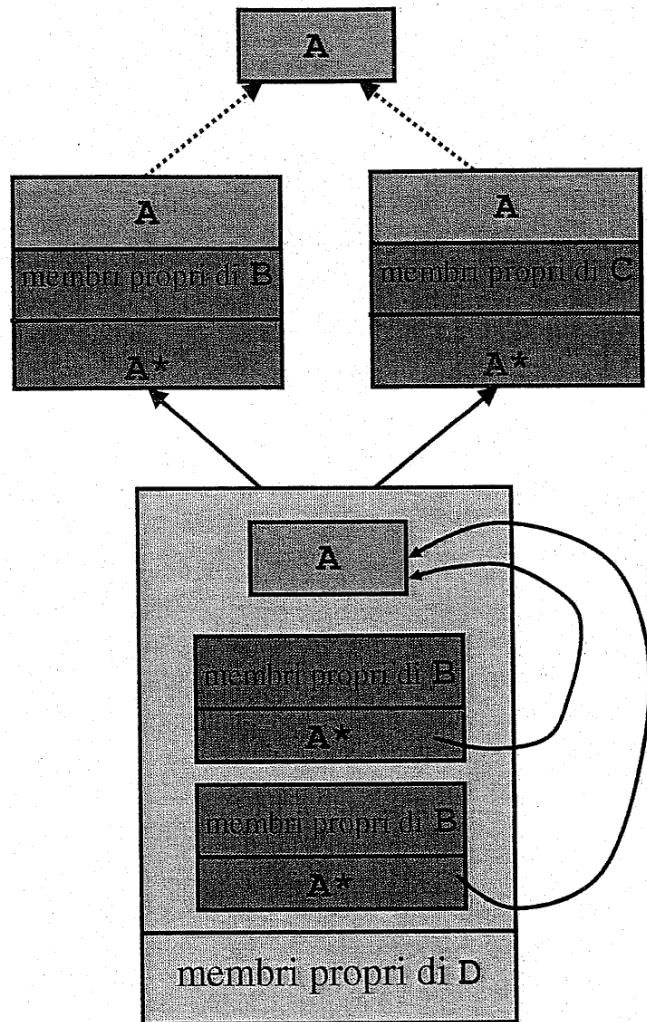
class D: public B, public C {
    int w[4];          // 16 byte
};
```

In questo caso le stampe prodotte dal precedente codice risultano piuttosto diverse:

```
cout << "sizeof(A) == " << sizeof(A) << endl;           // 20
cout << "sizeof(B) == " << sizeof(B) << endl;           // 36=12+4+20
cout << "sizeof(C) == " << sizeof(C) << endl;           // 36=12+4+20
cout << "sizeof(D) == " << sizeof(D); // 68=16+(12+4)+(12+4)+20
```

e si spiegano per l'utilizzo della derivazione virtuale. Infatti, in questo caso `sizeof(D)` è data dalla somma delle dimensioni dell'array `w` (16 byte) proprio di `D`, dell'array `y` (12 byte) proprio di `B`, dell'array `z` (12 byte) proprio di `C`, dei due puntatori in `B` e `C` (4 + 4 byte) che implementano la derivazione virtuale, e dall'array `a` (20 byte) dell'unico sottooggetto di `A` condiviso in `D`. La situazione è graficamente rappresentata nella seguente figura.

6.7 Ereditarietà multipla



Esempio 6.7.3. Consideriamo la gerarchia di classi per il secondo programma dell'Esempio 6.7.1, definita in questo caso tramite derivazione virtuale.

```
class A {  
public:  
    virtual void print()=0;  
};  
  
class B: virtual public A {  
public:  
    virtual void print() {cout << "B ";}  
};  
  
class C: virtual public A {  
public:  
    virtual void print() {cout << "C ";}  
};
```

6 EREDITARIETÀ

```
class D: public B, public C {  
    virtual void print() {cout << "D " ;}  
};
```

In questo caso è necessario definire l'overriding di `print()` in `D`, altrimenti si otterrebbe il seguente errore in compilazione: “no unique final overrider for `A::print()`”. Con queste nuove definizioni il seguente codice compila e produce la stampa indicata.

```
D d;  
A* p = &d; // Compila  
p->print(); // Stampa: D
```

Anche per la derivazione virtuale multipla, possiamo avere derivazione privata, pubblica o protetta. Quindi in una classe derivata può accadere di avere la stessa classe base virtuale indiretta ma con diverse regole d'accesso. In un caso del genere prevale la modalità di derivazione più permissiva, ovvero vale la seguente regola: la derivazione protetta prevale su quella privata, e la derivazione pubblica prevale su quella protetta. Ad esempio:

```
class A { public: void f() {cout << "A";} };  
  
class B: virtual private A {}; // derivazione virtuale privata  
  
class C: virtual public A {}; // derivazione virtuale pubblica  
  
class D: public B, public C {};// prevale la derivazione pubblica  
  
int main() {  
    D d;  
    d.f(); // OK, stampa: A (è il metodo C::f())  
//d.B::f(); // Illegale, A::f() inaccessibile  
}
```

Nei vari costruttori delle classi derivate vi è sempre, implicitamente o esplicitamente, una chiamata al costruttore della classe base virtuale. Sappiamo che un costruttore di una classe derivata richiama preliminarmente, o con una invocazione esplicita inclusa nella lista di inizializzazione oppure con una invocazione implicita, solamente i costruttori delle sue superclassi dirette. In presenza di classi base virtuali, il costruttore di una classe derivata richiama preliminarmente, o con una invocazione esplicita inclusa nella lista di inizializzazione (che è quindi consentita) oppure con una invocazione implicita, anche i costruttori delle classi virtuali che si trovano nella sua gerarchia di derivazione. La motivazione di ciò sta nel fatto che il sottooggetto di una classe base virtuale `A` è condiviso e quindi non avrebbe senso costruirlo più di una volta. La soluzione consiste pertanto nel costruire una

6.7 Ereditarietà multipla

sola volta tale sottooggetto di A condiviso e questa costruzione va fatta prima che avvenga la sua condivisione.

Tenendo in considerazione ereditarietà multipla e classi virtuali, aggiorniamo il comportamento di un costruttore di una classe derivata D come segue:

- (1) Per primi vengono richiamati, una sola volta, i costruttori delle classi virtuali che si trovano nella gerarchia di derivazione di D. Vi può essere più di una classe virtuale nella gerarchia di derivazione di D: la ricerca delle classi virtuali procede esaminando le superclassi dirette di D in ordine di dichiarazione di ereditarietà (cioè da sinistra verso destra nella gerarchia), e per ognuna di queste sottogerarchie si cercano, ricorsivamente, le classi base virtuali. Quindi si ricercano le classi virtuali nella gerarchia seguendo l'ordine da sinistra verso destra e dall'alto verso il basso ("left-to-right top-down order").
- (2) Una volta che sono stati invocati i costruttori delle classi virtuali nella gerarchia di derivazione di D, vengono richiamati i costruttori delle superclassi dirette non virtuali di D: questi costruttori escludono di richiamare eventuali costruttori di classi virtuali già richiamati al passo (1);
- (3) Infine viene eseguito il costruttore "proprio" di D, ovvero vengono costruiti i campi dati propri di D e quindi viene eseguito il corpo del costruttore di D.

Le chiamate dei costruttori dei punti (1) e (2), se non sono esplicitamente definite, vengono automaticamente inserite dal compilatore nella lista di inizializzazione del costruttore di D: in questo caso, come al solito, si tratta di chiamate implicite ai costruttori di default.

Quindi nel semplice caso "a diamante" seguente:

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

il compito di costruire il sottooggetto della classe base virtuale A viene lasciato alla classe dell'oggetto principale che stiamo definendo, nel nostro caso al costruttore della classe D. Pertanto, una eventuale chiamata esplicita al costruttore della classe virtuale A deve essere messa nella lista di inizializzazione della classe D. Di conseguenza, se nelle liste di inizializzazione delle classi B e C compare una invocazione al costruttore di A, nella costruzione di un oggetto di D tale chiamata viene ignorata. Vediamo alcuni esempi che illustrano l'ordine delle chiamate dei costruttori in una gerarchia di derivazione con classi base virtuali. In questi esempi le chiamate dei costruttori delle classi virtuali sono sempre implicite.

Esempio 6.7.4. Consideriamo la gerarchia di classi dell'Esempio 6.7.1, definita in questo caso tramite derivazione virtuale.

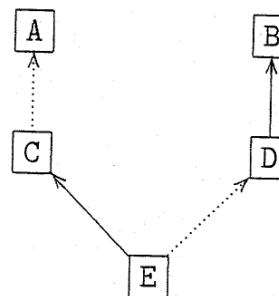
6 EREDITARIETÀ

```
class A {  
public:  
    int a;  
    A(int x=1): a(x) {}  
};  
  
class B: virtual public A {  
public:  
    B(): A(2) {}  
};  
  
class C: virtual public A {  
public:  
    C(): A(3) {}  
};  
  
class D: public B, public C {};
```

Il seguente codice compila ed esegue correttamente producendo, come ci si aspetta, la stampa riportata.

```
D d;  
A* p = &d; // Compila  
cout << p->a; // Stampa: 1 (e non 2 o 3!)
```

Esempio 6.7.5.

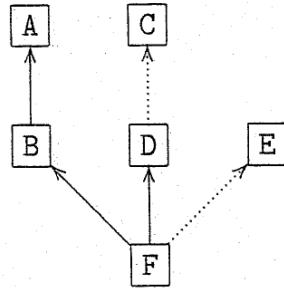


```
class A { public: A() {cout << "A ";}  
};  
class B { public: B() {cout << "B ";}  
};  
class C: virtual public A { public: C() {cout << "C ";}  
};  
class D: public B { public: D() {cout << "D ";}  
};
```

6.7 Ereditarietà multipla

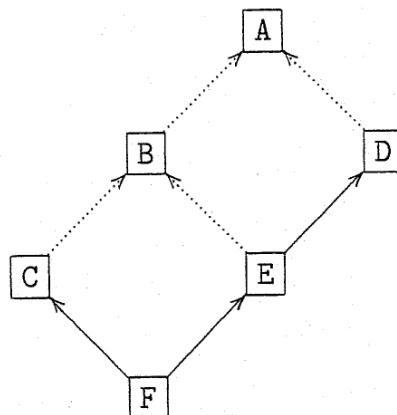
```
class E : public C, virtual public D {  
    public: E() {cout << "E ";}  
};  
int main() { E e; } // stampa: A B D C E
```

Esempio 6.7.6.



```
class A { public: A() {cout << "A ";}  
};  
class B: public A {  
    public: B() {cout << "B ";}  
};  
class C {  
    public: C() {cout << "C ";}  
};  
class D: virtual public C {  
    public: D() {cout << "D ";}  
};  
class E { public: E() {cout << "E ";} };  
  
class F: public B, public D, virtual public E {  
    public: F() {cout << "F ";}  
};  
int main() { F f; } // stampa: C E A B D F
```

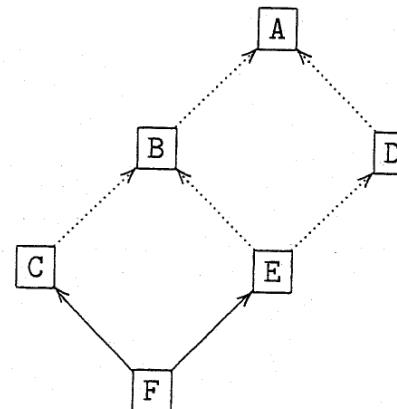
Esempio 6.7.7.



6 EREDITARIETÀ

```
class A { public: A() {cout << "A ";} };
class B: virtual public A {
    public: B() {cout << "B ";}
};
class D: virtual public A {
    public: D() {cout << "D ";}
};
class C: virtual public B {
    public: C() {cout << "C ";}
};
class E: virtual public B, public D {
    public: E() {cout << "E ";}
};
class F: public C, public E {
    public: F() {cout << "F ";}
};
int main() { F f; } // stampa: A B C D E F
```

Esempio 6.7.8.



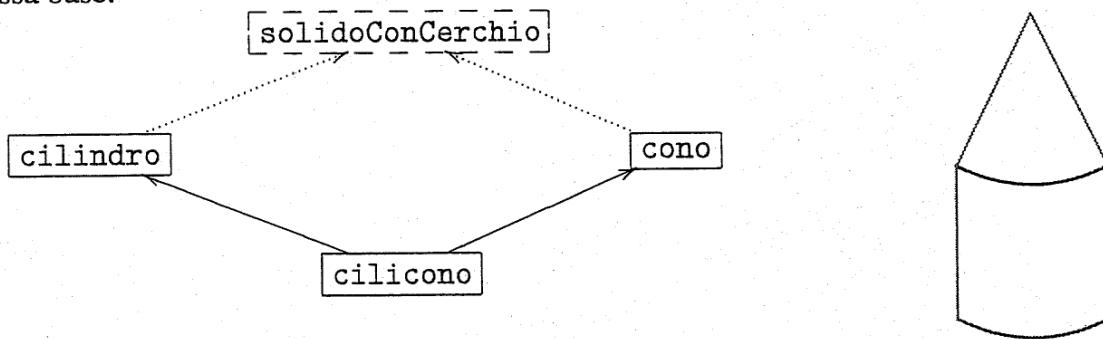
```
class A { public: A() {cout << "A ";} };
class B: virtual public A {
    public: B() {cout << "B ";}
};
class D: virtual public A {
    public: D() {cout << "D ";}
};
class C: virtual public B {
    public: C() {cout << "C ";}
};
```

6.7 Ereditarietà multipla

```
class E: virtual public B, virtual public D {  
    public: E() {cout << "E " ;}  
};  
class F: public C, public E {  
    public: F() {cout << "F " ;}  
};  
int main() { F f; } // stampa: A B D C E F
```

L'ordine delle invocazioni del costruttore di copia segue le stesse regole dei costruttori. L'ordine delle invocazioni dei distruttori è, come al solito, sempre inverso a quello dei costruttori. Le classi virtuali, invece, non hanno alcun effetto sull'assegnazione, eventualmente ridefinita.

Esempio 6.7.9. Si vuole mostrare una gerarchia di classi a diamante con una classe base virtuale ed astratta. L'obiettivo è definire una classe `cilicono` i cui oggetti rappresentano una figura geometrica solida composta da un cono sovrapposto ad un cilindro che abbiano la stessa base.



```
class solidoConCerchio { // classe base astratta virtuale  
protected:  
    double raggio;  
    double circonferenza() const {return (2*M_PI*raggio);}  
    double areaCerchio() const {return (M_PI*raggio*raggio);}  
public:  
    solidoConCerchio(double r): raggio(r) {}  
    virtual double area() const = 0; // virtuale pura  
    virtual double volume() const = 0; // virtuale pura  
};
```

```
// derivazione virtuale  
class cilindro: virtual public solidoConCerchio {  
protected:  
    double altezza;  
    double areaLaterale() const {return (circonferenza()*altezza);}}
```

6 EREDITARIETÀ

```
public:  
    cilindro(double a, double h): solidoConCerchio(a), altezza(h) {}  
    virtual double area() const {  
        return (2*areaCerchio() + areaLaterale());  
    }  
    virtual double volume() const {return (areaCerchio()*altezza);}  
};
```

```
// derivazione virtuale  
class cono: virtual public solidoConCerchio {  
protected:  
    double altezza;  
    double areaLaterale() const {  
        double apotema = sqrt(raggio*raggio + altezza*altezza);  
        return (M_PI*raggio*apotema);  
    }  
public:  
    cono(double a, double h): solidoConCerchio(a), altezza(h) {}  
    virtual double area() const {  
        return (areaCerchio() + areaLaterale());  
    }  
    virtual double volume() const {  
        return (areaCerchio()*altezza/3);  
    }  
};
```

```
// derivazione multipla: un solo sottooggetto solidoConCerchio  
class cilicono: public cilindro, public cono {  
public:  
    cilicono(double r, double h1, double h2) :  
        solidoConCerchio(r), cilindro(r, h1), cono(r, h2) {}  
    virtual double area() const {  
        return (cilindro::areaLaterale() +  
                cono::areaLaterale() + areaCerchio());  
    }  
    virtual double volume() const {  
        return (cilindro::volume() + cono::volume());  
    }  
};
```

Si noti che se il costruttore della classe cilicono non invocasse esplicitamente tramite `solidoConCerchio(r)` il costruttore di `solidoConCerchio` allora tale definizione non compilerebbe perché il costruttore di default di `solidoConCerchio` non è disponibile. Il seguente esempio di `main()` usa la precedente gerarchia di classi.

6.7 Ereditarietà multipla

```
int main() {
    cilindro cil(1, 2); cono co(1, 2); cilicono clc(1, 2, 2);
    solidoConCerchio* p = &cil;
    cout << "Area cilindro: " << p->area() << endl;
    cout << "Volume cilindro: " << p->volume() << endl;
    p = &co;
    cout << "Area cono: " << p->area() << endl;
    cout << "Volume cono: " << p->volume() << endl;
    p = &clc;
    cout << "Area cilicono: " << p->area() << endl;
    cout << "Volume cilicono: " << p->volume() << endl;
}
```

6 EREDITARIETÀ

Capitolo 7

Eccezioni

Talvolta, durante l'esecuzione di una qualsiasi funzione (in particolare un metodo di una classe), si possono riscontrare delle situazioni "eccezionali" in cui non è chiaro come la funzione possa procedere. Ad esempio, quando il metodo `Estrai_Una()` della classe `bolletta` che dovrebbe estrarre la telefonata in testa alla bolletta di invocazione viene invocato su una bolletta vuota non è ben chiaro come si dovrebbe procedere. In molti di questi casi la funzione, chiamiamola `F()`, non può fare altro che interrompersi. Infatti una tale situazione eccezionale può essere eventualmente risolta soltanto nel contesto di invocazione della funzione `F()`. Un modo per permettere alla funzione (che eventualmente può anche essere il `main()`) che ha invocato `F()` di gestire la situazione eccezionale potrebbe essere quello di segnalare la situazione eccezionale con un valore particolare del risultato ritornato da `F()`, ad esempio ritornando il puntatore nullo se il risultato è un puntatore, etc. Tuttavia, se non fosse possibile utilizzare valori particolari del risultato (ad esempio per una funzione che calcola la divisione intera) sarebbe necessario aggiungere alla funzione `F()` un parametro per riferimento in cui memorizzare la situazione eccezionale da segnalare. In ogni caso, nel codice che invoca tale funzione `F()` dovremmo aggiungere subito dopo ogni chiamata di `F()` un test del risultato o del parametro aggiuntivo e il codice per la gestione dell'eventuale situazione eccezionale. Questa tecnica potrebbe portare ad un programma prolioso, difficilmente leggibile e facilmente soggetto ad errori. Il C++ fornisce strumenti specifici per la gestione delle situazioni eccezionali che permettono di tenere nettamente separato il codice per la gestione dei casi eccezionali dal codice relativo ad un normale (cioè esente da situazioni eccezionali) flusso dell'esecuzione.

7.1 `throw` e `try/catch`

La funzione in cui la situazione eccezionale si verifica lancia (o solleva) una cosiddetta *eccezione* tramite il costrutto `throw`. Ad esempio, consideriamo il metodo `Estrai_Una`

7 ECCEZIONI

(nella versione con puntatori smart):

```
telefonata bolletta::Estrai_Una() {
    if (Vuota()) throw Ecc_Vuota();
    telefonata aux = first->info;
    first = first->next;
    return aux;
}
```

L'oggetto lanciato Ecc_Vuota() appartiene ad una classe definita allo scopo:

```
class Ecc_Vuota {
    ...
};
```

L'esecuzione di throw comporta la terminazione dell'esecuzione della funzione Estrai_Una() con il lancio dell'eccezione Ecc_Vuota() (un oggetto eccezione anonimo costruito con il costruttore di default di Ecc_Vuota) alla funzione chiamante (che può anche essere il main()).

Nella funzione chiamante, il codice contenente la chiamata o le chiamate alla funzione è racchiuso in un blocco try ed è seguito da una lista di procedure, le clausole catch (dette gestori di eccezioni), che catturano e gestiscono le eccezioni sollevate. La funzione abort() fa parte della libreria standard del C — che va inclusa con #include<cstdlib> — e provoca la terminazione “brutale” del main() (in particolare, a differenza della funzione exit() che viene chiamata alla normale terminazione del main(), non vengono richiamati i distruttori per gli oggetti statici e globali).

```
try { b. Estrai_Una(); }
catch (Ecc_Vuota) {
    cout << "La bolletta è vuota" << endl;
    abort();
}
```

Un altro esempio di funzione che può sollevare delle eccezioni potrebbe essere l'operatore di input della classe orario.

```
istream& operator>>(istream& is, orario& o) {
    // formato di input: hh:mm:ss
    char c; int ore, minuti, secondi;
    string::size_type pos;
    string cifre("0123456789");
    is >> c; // prima cifra delle ore
    pos = cifre.find(c); ore = pos;
    is >> c;
```

7.1 throw e try/catch

```
if (c != ':') {
    // seconda cifra delle ore
    pos = cifre.find(c);
    ore = ore * 10 + pos;
    is >> c; // input di ':'
} // ho letto le ore e c = ':'
is >> c; // prima cifra dei minuti
pos = cifre.find(c);
minuti = pos;
is >> c;
if (c != ':') {
    // seconda cifra dei minuti
    pos = cifre.find(c);
    minuti = minuti * 10 + pos;
    is >> c; // input di ':'
} // ho letto i minuti e c = ':'
is >> c; // prima cifra dei secondi
pos = cifre.find(c);
secondi = pos;
is >> c;
if(is && cifre.find(c)!= string::npos){
    // seconda cifra secondi
    pos = cifre.find(c);
    secondi = secondi * 10 + pos;
} // ho letto i secondi
else if (is) // carattere non cifra
    is.putback(c);
o.sec = ore*3600 + minuti*60 + secondi;
return is;
}
```

Con questa definizione dell'operatore di input può accadere che:

- (a) I valori di ore, minuti e secondi non siano formati da uno o due caratteri cifra (caratteri '0' ... '9') oppure non siano separati dal carattere ':'.
- (b) Lo stream termini (su cin con **<Ctrl>+<D>**, da file per un EOF) prima che sia stato letto secondi;
- (c) I valori di ore, minuti e secondi non rispettino i limiti (ore ≤ 23 , minuti ≤ 59 , secondi ≤ 59).

Definiamo allora le seguenti classi di eccezioni:

```
class err_sint {};      // errore di sintassi
```

7 ECCEZIONI

```
class fine_file {}; // file finito prematuramente  
class err_ore {}; // ora > 23  
class err_min {}; // minuti > 59  
class err_sec {}; // secondi > 59
```

che solleveremo all'interno della funzione nel modo seguente:

```
istream& operator>>(istream& is, orario& o) {  
    char c; string::size_type pos;  
    string cifre("0123456789");  
    int ore, minuti, secondi;  
    if (!(is >> c)) throw fine_file();  
    pos = cifre.find(c);  
    if (pos == string::npos) throw err_sint();  
    ore = pos;  
    if (!(is >> c)) throw fine_file();  
    if (c != ':') {  
        pos = cifre.find(c);  
        if (pos == string::npos) throw err_sint();  
        ore = ore * 10 + pos;  
        if (ore > 23) throw err_ore();  
        if (!(is >> c)) throw fine_file();  
    }  
    if (c != ':') throw err_sint();  
    if (!(is >> c)) throw fine_file();  
    pos = cifre.find(c);  
    if (pos == string::npos) throw err_sint();  
    minuti = pos;  
    if (!(is >> c)) throw fine_file();  
    if (!(is >> c)) throw fine_file();  
    if (c != ':') {  
        pos = cifre.find(c);  
        if (pos == string::npos) throw err_sint();  
        minuti = minuti * 10 + pos;  
        if (minuti > 59) throw err_min();  
        if (!(is >> c)) throw fine_file();  
    }  
    if (c != ':') throw err_sint();  
    if (!(is >> c)) throw fine_file();  
    pos = cifre.find(c);  
    if (pos == string::npos) throw err_sint();  
    secondi = pos;  
    is >> c;
```

7.1 throw e try/catch

```
if (is && cifre.find(c) != string::npos) {
    pos = cifre.find(c);
    secondi = secondi * 10 + pos;
    if (secondi > 59) throw err_sec();
}
else if (is) is.putback(c);
o.sec = ore*3600 + minuti*60 + secondi;
return is;
}
```

Possiamo usare tale operatore di input di orario in una funzione che prende in input due oggetti di tipo orario e li somma.

```
orario somma() {
    orario o1,o2;
    try { cin >> o1; }
    catch (err_sint) {cerr <<"Errore di sintassi"; return orario();}
    catch (fine_file) {cerr <<"Errore EOF"; abort();}
    catch (err_ore) {cerr <<"Errore nelle ore"; return orario();}
    catch (err_min) {cerr <<"Errore in minuti"; return orario();}
    catch (err_sec) {cerr <<"Errore in secondi"; return orario();}
    try { cin >> o2; }
    catch (err_sint) {cerr <<"Errore di sintassi"; return orario();}
    catch (fine_file) {cerr <<"Errore EOF"; abort();}
    catch (err_ore) {cerr <<"Errore ore"; return orario();}
    catch (err_min) {cerr <<"Errore minuti"; return orario();}
    catch (err_sec) {cerr <<"Errore secondi"; return orario();}
    return o1+o2;
}
```

Se una clausola catch non contiene l'istruzione return (o abort ()) l'esecuzione continua dal punto di programma che immediatamente segue le clausole catch del blocco try.

La soluzione vista funziona correttamente ma la gestione delle eccezioni è mescolata alle istruzioni relative al normale flusso dell'esecuzione. Una soluzione migliore consiste nel racchiudere tutto il corpo della funzione in un unico blocco try e mettere tutte le clausole catch alla fine.

```
orario somma() {
    try {
        orario o1, o2; cin >> o1 >> o2; return o1 + o2;
    }
    catch (err_sint) {cerr <<"Errore di sintassi"; return orario();}
    catch (fine_file) {cerr <<"Errore EOF"; abort();}
    catch (err_ore) {cerr <<"Errore ore"; return orario();}
```

```
    catch (err_min) {cerr << "Errore minuti"; return orario();}  
    catch (err_sec) {cerr << "Errore secondi"; return orario();}  
}
```

È anche possibile usare la seguente sintassi compatta quando l'intero corpo di una funzione è contenuto in un blocco try:

```
orario somma() try {  
    orario o1, o2; cin >> o1 >> o2; return o1 + o2;  
}  
catch (err_sint) {cerr << "Errore di sintassi"; return orario();}  
catch (fine_file) {cerr << "Errore EOF"; abort();}  
catch (err_ore) {cerr << "Errore ore"; return orario();}  
catch (err_min) {cerr << "Errore minuti"; return orario();}  
catch (err_sec) {cerr << "Errore secondi"; return orario();}
```

Sebbene le eccezioni siano spesso oggetti di qualche classe, una throw può lanciare un'espressione di qualsiasi tipo. Ad esempio:

```
enum Errori {ErrSint, ErrEOF, ErrOre, ErrMin, ErrSec};  
  
if (t.secondi > 59) throw ErrSec;  
...
```

7.1.1 Ricerca della clausola catch

Quando in una funzione F() viene sollevata una eccezione tramite una istruzione throw inizia la ricerca della clausola catch in grado di catturarla. Se l'espressione throw è collocata in un blocco try, l'esecuzione abbandona il blocco try e vengono esaminate in successione tutte le clausole catch associate a tale blocco per vedere se ne esiste una in grado di catturare l'eccezione. Se la si trova l'eccezione viene catturata e viene eseguito il codice della catch; eventualmente, al termine dell'esecuzione del corpo della catch il controllo dell'esecuzione passa al punto di programma che immediatamente segue l'ultimo blocco catch. Se non la si trova oppure se l'istruzione throw non era collocata all'interno di un blocco try la ricerca continua nella funzione che ha invocato la funzione F(). Questo processo continua fino a che viene individuata una clausola catch in grado di catturare l'eccezione oppure, se non ne esiste alcuna, si arriva al metodo main() e in questo caso viene richiamata la funzione di libreria terminate() che per default chiama la funzione abort() che fa terminare il programma con un errore.

7.1.2 catch generica

Quando si esce da una funzione a causa del sollevamento di una eccezione vengono richiamati i distruttori per le variabili locali della funzione. Questo non garantisce il recupero di

7.1 throw e try/catch

eventuale memoria dinamica allocata dalla funzione o la chiusura di file aperti dalla funzione. In altre parole potremmo volere che certe azioni vengano comunque eseguite quando si esce dalla funzione.

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

void F() { A* p = new A[3]; throw 1; delete p; }

int main() {
    try { F(); }
    catch (int) {cout << "int ";
    cout <<"fine ";
}
// Stampa: int fine
// Ovviamente NON stampa: ~A ~A ~A
```

Il seguente è uno schema generale di esempio:

```
gestore () {
    risorsa rs; // alloco una risorsa
    rs.use();
    ...
    // codice che puo' sollevare eccezioni
    rs.release(); // non viene eseguita in caso di eccezione
}
```

Se viene sollevata una eccezione e questa non viene catturata all'interno della funzione si esce dalla funzione senza rilasciare la risorsa.

Possiamo gestire questo caso tramite una cosiddetta *catch generica*.

```
gestore () try {
    risorsa rs;
    rs.use();
    ...
    // codice che puo' sollevare eccezioni
    rs.release();
}
catch (...) { // catch generica
    rs.release();
    throw; // rilancia l'eccezione al chiamante
}
```

La sintassi `catch(...)` denota una *catch generica* in grado di catturare tutte le eccezioni possibili. Quindi, se si definisce un blocco *catch generico* prima di altri blocchi *catch*, questi blocchi non potranno mai essere eseguiti: si tratterebbe di un errore logico di programmazione. Una *catch generica* deve quindi sempre essere l'ultima nella lista delle *catch* che seguono un blocco *try*.

7.1.3 Rilanciare eccezioni

È possibile che una clausola `catch` si accorga di non poter gestire direttamente una eccezione. In tal caso essa può rilanciare l'eccezione alla funzione chiamante tramite una `throw`. Ad esempio:

```
orario somma() try {
    orario o1, o2; cin >> o1 >> o2; return o1 + o2;
}
catch (err_sint) {cerr << "Errore di sintassi"; return orario();}
catch (fine_file) {cerr << "Errore EOF"; throw;}
catch (err_ore) {cerr << "Errore ore"; return orario();}
catch (err_min) {cerr << "Errore minuti"; return orario();}
catch (err_sec) {cerr << "Errore secondi"; return orario();}
```

Si noti che una `catch` che come unica azione rilanci l'eccezione al chiamante non è significativa, in quanto si comporta come se non fosse presente (equivale all'azione di default).

Se durante un blocco `try` non si verificano eccezioni, le `catch` relative a tale blocco vengono saltate e l'esecuzione continua dal punto di programma che immediatamente segue l'ultimo blocco `catch`. Ogni `catch` specifica il tipo di eccezione che può catturare ed opzionalmente il nome di un parametro. Quando una `catch` cattura un'eccezione viene quindi eseguito il codice presente nel blocco. La `catch` che cattura un'eccezione è la prima `catch` incontrata durante la ricerca che abbia un *tipo compatibile* con il tipo dell'eccezione lanciata. Le regole che definiscono la compatibilità tra il tipo `T` del parametro di una `catch` non generica ed il tipo `E` dell'eccezione sono le seguenti:

- (1) Il tipo `T` è uguale al tipo `E`;
- (2) `E` è un sottotipo derivato pubblicamente da `T`;
- (3) `T` è un tipo puntatore `B*` ed `E` è un tipo puntatore `D*` dove `D` è una classe derivata da `B`;
- (4) `T` è un tipo riferimento `B&` ed `E` è un tipo riferimento `D&` dove `D` è una classe derivata da `B`;
- (5) `T` è il tipo `void*` ed `E` è un qualsiasi tipo puntatore;
- (6) Non possono essere applicate conversioni implicite.

Notiamo quindi che, a causa delle regole (2) e (3), se si definisce una `catch` che cattura un oggetto di (o un puntatore a) una classe base prima di una `catch` che cattura un oggetto di una sua classe derivata si commette un errore logico di progettazione. Infatti, la `catch` della classe base catturerà tutti gli oggetti delle classi derivate e quindi la `catch` della classe derivata non potrà mai essere eseguita. L'organizzazione delle eccezioni in una gerarchia di

classi offre il vantaggio di utilizzare il meccanismo della derivazione e dei metodi virtuali. Molto spesso è quindi questo l'approccio usato nella specifica delle eccezioni che sono previste per un insieme di moduli software.

Le catch possono essere definite in svariati modi a seconda del contesto e delle esigenze. Alcuni comportamenti tipici sono i seguenti:

- esaminare l'errore ed invocare `terminate()`;
- rilanciare un'eccezione;
- convertire un tipo di eccezione in un altro, lanciando un'eccezione diversa;
- cercare di ripristinare il funzionamento, in modo che il programma possa continuare dal punto di programma che immediatamente segue l'ultima catch;
- analizzare la situazione che ha causato l'errore, eliminarne eventualmente la causa e riprovare a chiamare la funzione che ha causato originariamente l'eccezione;
- restituire un valore di stato all'ambiente.

7.1.4 Specifica di eccezioni

Nella dichiarazione di una funzione è opportuno indicare le eccezioni che essa può sollevare. Potremmo farlo in modo informale usando dei commenti:

```
istream& operator>>(istream& is, orario& o);  
// input nel formato hh:mm:ss.  
// puo' sollevare le eccezioni err_sint, err_sint,  
// err_ore, err_sec, err_min
```

Un modo più conveniente è usare una *specifica esplicita delle eccezioni*:

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

```
orario somma() throw(fine_file) {  
...  
}
```

La specifica delle eccezioni fa parte del prototipo della funzione. La stessa specifica delle eccezioni deve quindi comparire sia nella definizione della funzione che in tutte le eventuali dichiarazioni. Se si scrive `throw()` dopo la lista dei parametri di una funzione si dichiara che tale funzione non può lanciare alcuna eccezione. Una funzione può

lanciare solamente le eccezioni con i tipi specificati o con dei tipi da essi derivati (non è comunque un errore di sintassi!). Se durante l'esecuzione della funzione viene sollevata una eccezione non indicata nella specifica essa non viene propagata ma causa la terminazione immediata del programma. Questo succede perché viene automaticamente lanciata l'eccezione di tipo predefinito (di libreria) `bad_exception` il cui blocco `catch` predefinito invoca la funzione `unexpected()` che per default invoca la funzione `terminate()`. La funzione `unexpected()` potrebbe essere ridefinita passando la ridefinizione alla funzione `set_unexpected()` (tralasciamo i dettagli).

7.2 La gerarchia exception

Il C++ standard prevede una gerarchia di classi di eccezioni predefinita. `exception` è la classe base (la cui dichiarazione è visibile tramite `#import <exception>`), da cui derivano `runtime_error` e `logic_error` (visibili con `#import <stdexcept>`), da cui derivano parecchie altre classi. Derivano da `logic_error`: `invalid_argument`, `length_error`, `out_of_range`, etc. Derivano da `runtime_error`: `overflow_error`, `underflow_error`, etc.

Derivano da `exception` anche le seguenti classi di eccezioni:

`bad_cast`, le cui eccezioni sono lanciate dall'operatore `dynamic_cast`.

`bad_alloc`, lanciata dalla `new` quando lo heap è esaurito (il gestore di default invoca la `terminate()`).

`bad_exception`, che abbiamo già visto.

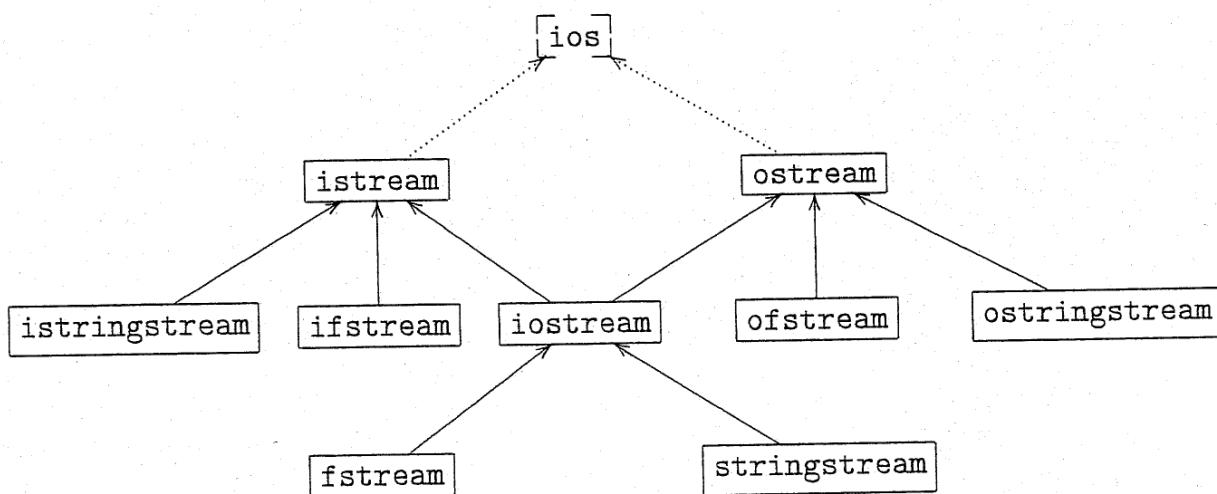
`bad_typeid`, viene lanciata dall'operatore `typeid` quando ha come argomento un puntatore nullo.

Un uso indiscriminato dello strumento C++ di gestione delle eccezioni può appesantire la programmazione e peggiorare sensibilmente l'efficienza del programma. Il compilatore infatti inserisce, dopo ogni chiamata di funzione, del codice che controlla a run-time se sono state sollevate eccezioni, che cerca le eventuali clausole `catch` e trasferisce il controllo ad una di esse oppure propaga l'eccezione ad un livello superiore. Lo strumento C++ di gestione delle eccezioni va quindi usato con parsimonia effettuando una gestione locale dei casi eccezionali ogni volta che questo risulti possibile.

Capitolo 8

La Gerarchia di Classi per l'I/O

Le classi della libreria standard per la gestione dell'Input/Output che esamineremo in questo capitolo formano la gerarchia rappresentata nella seguente figura.



Gli oggetti delle varie classi di I/O vengono detti *stream*: sono delle sequenze non limitate di celle ognuna contenente un byte. La posizione di una cella di uno stream è un intero che parte da 0, come negli array. Ogni stream ha associato un buffer attraverso il quale passano effettivamente le operazioni di I/O. Per utilizzare le superclassi di `iostream` occorre includere il file header `<iostream>`, mentre il file header `<fstream>` contiene le dichiarazioni per le classi `fstream`, `ifstream` e `ofstream`. In effetti si tratta di classi che sono istanziazioni di template di classe al tipo `char`. Per i dettagli della libreria di I/O si può consultare la documentazione della implementazione GNU libio che si può trovare liberamente disponibile on-line.

8 LA GERARCHIA DI CLASSI PER L'I/O

La classe ios

ios è la classe base astratta della gerarchia che permette di controllare lo stato di funzionamento di uno stream. Inoltre permette di determinare il formato di input/output dei caratteri nello stream, ma nel seguito non analizzeremo questo aspetto. Per quanto concerne lo stato di uno stream, la dichiarazione della classe ios è la seguente:

```
class ios {
    int state;
public:
    enum io_state {goodbit=0, eofbit=1, failbit=2, badbit=4};
    int good() const;
    int eof() const;
    int fail() const;
    int bad() const;
    int rdstate() const;
    void clear(int i=0);
    ...
};
```

Uno stream può trovarsi in 8 ($= 2^3$) stati di funzionamento diversi. Lo stato è un intero nell'intervallo [0,7] rappresentato dal campo dati state che corrisponde al numero binario

bad fail eof

dove *bad*, *fail* ed *eof* sono dei bit (0 o 1) di stato.

eof=1 se e soltanto se si è verificato un “end-of-input” nello stream: ad esempio, si è alla fine fisica di un file stream (ovvero si è a end-of-file) oppure l’utente ha determinato mediante <Ctrl>-<Z> (o <Ctrl>-<D>) l’end-of-input per uno stream di input da console.

fail=1 se e soltanto se una precedente operazione di I/O sullo stream è fallita: si tratta di un errore senza perdita di dati, normalmente è possibile continuare. Ad esempio, ci si aspettava in input un carattere cifra e si è invece trovato un carattere lettera. Il bit di *fail* viene messo a 1 anche quando si verifica un end-of-input.

bad=1 se e soltanto se una precedente operazione di I/O sullo stream è fallita con perdita dei dati: normalmente non è possibile continuare.

Gli stati *goodbit*, *eofbit*, *failbit* e *badbit* corrispondono rispettivamente ai numeri binari 000, 001, 010, 100. Se lo stream è nello stato *goodbit* significa che la precedente operazione sullo stream ha avuto successo e sullo stream può essere fatta una nuova operazione. Il metodo *good()* ritorna 1 se siamo nello stato *goodbit* altrimenti ritorna 0. I metodi *eof()*, *fail()* e *bad()* ritornano, rispettivamente, il valore dei bit di stato *eof*, *fail* e *bad*. Il metodo *rdstate()* restituisce lo stato di uno stream. Una volta che un

bit di stato viene messo a 1, esso rimane anche successivamente con il valore 1, e ciò non è sempre quello che si vuole. Per ripristinare lo stato di uno stream possiamo allora usare il metodo `clear`. Il metodo `clear(int x=0)` porta lo stream di invocazione nello stato `x`, in particolare una invocazione `s.clear()` porta lo stream di invocazione `s` nello stato `goodbit`.

La classe `istream`

Gli oggetti della sottoclasse `istream` rappresentano stream di input. In particolare, `cin` è un oggetto di `istream`. La classe `istream` include l'overloading dell'operatore di input `operator>>` per i tipi primitivi e per gli array di caratteri.

```
class istream: public virtual ios {  
public:  
    istream& operator>>(char&);  
    istream& operator>>(int&);  
    istream& operator>>(double&);  
    istream& operator>>(char*);  
    ...  
};
```

Tutti gli operatori di input ignorano le spaziature (cioè spazi, tabulazioni o CR) presenti prima del valore da prelevare.

`operator>>(char& c)` : preleva dall'`istream` di invocazione un singolo carattere e lo assegna a `c`.

`operator>>(int & i)` e `operator>>(double & d)` : prelevano dall'`istream` di invocazione una sequenza di caratteri che rispetta la sintassi dei valori di `int` e `double` e converte tale sequenza nella rappresentazione numerica di `int` o `double` assegnandolo a `i` o a `d`. Se la sequenza di caratteri non soddisfa la sintassi prevista, l'operazione è nulla e l'`istream` passa ad uno stato di errore recuperabile: `fail=1` e `bad=0`.

`operator>>(char* s)` : preleva dall'`istream` di invocazione una sequenza di caratteri fino ad incontrare il carattere spazio (che non viene prelevato), a questa sequenza viene aggiunto il carattere nullo (codice ASCII 0) e viene quindi fatta puntare da `s`.

Bisogna prestare attenzione al fatto che quando una operazione di input fallisce (cioè `fail=1`) non viene effettuato alcun prelievo dallo stream e la variabile argomento di `operator>>` non subisce modifiche.

Le operazioni di input ritornano un riferimento all'`istream` di invocazione e ciò quindi consente l'usuale sequenza di input consecutivi:

8 LA GERARCHIA DI CLASSI PER L'I/O

```
cin >> x >> y;  
// equivale a:  
(cin.operator>>(x)).operator>>(y);
```

Vediamo un esempio di overloading di `operator>>` in una classe.

```
class Punto {  
    friend istream& operator>>(istream&, Punto&);  
    // legge un Punto nel formato di input (x1,x2)  
private:  
    double x, y;  
};  
  
istream& operator>>(istream& in, Punto& p) {  
    char cc; in >> cc;  
    if (cc=='q') return in; // ha prelevato 'q' ed esce  
    if (cc != '(') { in.clear(ios::failbit); return in; }  
    // ha prelevato il carattere e setta lo stato failbit  
    else {  
        in >> p.x;  
        if(!in.good()) { in.clear(ios::failbit); return in; }  
        in >> cc;  
        if (cc != ',') { in.clear(ios::failbit); return in; }  
        else {  
            in >> p.y;  
            if(!in.good()) { in.clear(ios::failbit); return in; }  
            in >> cc;  
            if (cc != ')') { in.clear(ios::failbit); return in; }  
        }  
    }  
    return in;  
}
```

Possiamo quindi inserire degli oggetto di `Punto` tramite il precedente operatore di input nel seguente modo:

```
Punto p; cout <<"Inserisci nel formato (x,y) ['q' per uscire]\n";  
while(cin.good()) { // while(stato == 0)  
    cin >> p;  
    if(cin.fail()) {  
        cout <<"Input non valido, ripetere!\n";  
        cin.clear(ios::goodbit); char c=0;  
        // 10 e' il codice ASCII del carattere newline  
        while(c!=10) { cin.get(c); } // svuota cin  
        cin.clear(ios::goodbit);  
    }  
}
```

```
    else cin.clear(ios::failbit); // stato 2
}
```

La classe ostream

Gli oggetti della sottoclasse ostream derivata da ios rappresentano stream di output. In particolare, cout e cerr sono oggetti di ostream. La classe ostream include l'overloading dell'operatore di output operator<< per i tipi primitivi e per gli array di caratteri costanti.

```
class ostream: public virtual ios {
public:
    ostream& operator<<(char);
    ostream& operator<<(int);
    ostream& operator<<(double);
    ostream& operator<<(const char*); // stringhe
    ostream& operator<<(const void*); // puntatori
    ostream& flush();
    ...
};
```

Questi operatori convertono i parametri in sequenze di caratteri che vengono scritti (immessi) nelle celle dell'ostream di invocazione. Per quanto riguarda l'output di stringhe, i caratteri della stringa vengono scritti nell'ostream fino al carattere nullo escluso. Le operazioni di output su uno ostream avvengono effettivamente su un buffer associato all'ostream. Questo buffer viene svuotato dal metodo flush(): tale metodo viene invocato automaticamente quando si scrive un carattere di fine riga '\n' oppure il manipolatore endl.

Operatori in ios

Uno stream può essere usato come condizione booleana. La condizione vale true se e soltanto se il metodo fail() sullo stream ritorna 0, altrimenti vale false. Ciò funziona perché ios fornisce i seguenti operatori:

```
class ios {
public:
    // operatore di conversione esplicita a void*
    operator void*() const;
    // ritorna il puntatore nullo se e solo se fail() ritorna 1

    // operatore di negazione !
    int operator!() const;
    // ritorna 0 se e solo se fail() vale 0
```

```
};
```

Vediamo un esempio di utilizzo.

```
int main() {
    int i;
    // ciclo infinito che legge interi da cin
    while (cin >> i) { cout << i << endl; }
    cout << "Non hai immesso un intero" << endl;
}
```

L'operazione di input `cin >> i` restituisce un riferimento a `cin`, e su tale riferimento viene applicato l'operatore di conversione esplicita a `void*`. Pertanto la condizione è falsa quando viene immessa in input una sequenza di caratteri che non rappresenta un valore intero.

I/O testuale e binario

L'input/output sugli stream tramite gli operatori `>>` e `<<` considerano gli stream come dei contenitori di testo, cioè in *formato testo*. Questo significa che quando si legge una certa variabile `x` di tipo `T` tramite `operator>>` dallo stream viene prima prelevata una sequenza di caratteri che costituisce la rappresentazione testuale di `x` e poi tale sequenza di caratteri viene convertita ad un valore di tipo `T` che viene assegnato a `x`. Dualmente, quando si scrive un certo valore `x` di tipo `T`, prima il valore viene convertito in una sequenza di caratteri che costituisce la sua rappresentazione testuale e poi tale sequenza di caratteri viene immessa nello stream. Questo implica che l'informazione da leggere o scrivere su uno stream deve avere una natura testuale, ma spesso ciò non è vero (almeno in modo naturale). In questo caso, vogliamo considerare lo stream in *formato binario*, cioè tutti i singoli caratteri dello stream vengono trattati allo stesso modo senza alcuna interpretazione per i caratteri di controllo o conversione fra sequenze di caratteri e valori.

L'input da uno `istream` in binario, cioè carattere per carattere, può essere fatto tramite i seguenti metodi di `istream`.

```
class istream: public virtual ios {
public:
    int get();
    istream& get(char& c);
    istream& get(char* p, int n, char c = '\n');
    istream& getline(char* p, int n, char c = '\n');
    istream& read(char* p, int n);
    istream& ignore(int n=1, int e = EOF);
    int gcount() const;
```

```
}; ...
```

Per dettagli ed esempi d'uso la documentazione fa da riferimento. Ci limitiamo ad una sintetica descrizione di questi metodi.

Il metodo `get()` preleva un singolo carattere (cioè 1 byte) dall'oggetto `istream` di invocazione e lo restituisce convertito ad intero. Se si è tentato di leggere EOF ritorna -1. Il metodo `get(char& c)` invece memorizza in `c` il carattere prelevato.

Il metodo `get(char* p, int n, char c='\\n')` preleva `n-1` caratteri dall'oggetto `istream` di invocazione oppure `m < n-1` caratteri sino a che incontra il carattere di terminazione `c`, che però non viene prelevato, componendo una stringa con carattere nullo finale che viene fatta puntare da `p`. Il metodo `getline(char*, int, char)` si comporta allo stesso modo con la differenza che preleva dallo stream anche il carattere di terminazione tuttavia senza memorizzarlo nella stringa.

Il metodo `read(char* p, int n)` preleva dall'oggetto `istream` di invocazione `n` caratteri, a meno che non incontri prima EOF, e li memorizza in una stringa puntata da `p`.

Il metodo `ignore(int n, int e=EOF)` effettua il prelievo di `n` caratteri ma non li memorizza.

Il metodo `gcount()` ritorna il numero di caratteri effettivamente prelevati nell'ultima invocazione del metodo `getline()` oppure del metodo `read()`.

L'output su uno `ostream` in binario può essere fatto tramite i seguenti metodi di `ostream`.

```
class ostream: public virtual ios {  
public:  
    ostream& put(char c);  
    ostream& write(const char* p, int n);  
    ...  
};
```

Il metodo `put(char c)` scrive il carattere `c` sull'`ostream` di invocazione. Il metodo `write(const char* p, int n)` scrive sull'`ostream` di invocazione i primi `n` caratteri della stringa puntata da `p`.

Stream di file

Si possono definire stream associati a file. Questi stream devono essere oggetti delle classi `ifstream`, `ofstream` e `fstream`. Sono disponibili diversi costruttori (si veda la documentazione), i più comuni dei quali sono i seguenti:

8 LA GERARCHIA DI CLASSI PER L'I/O

```
ifstream(const char* nomefile, int modalita=ios::in);
ofstream(const char* nomefile, int modalita=ios::out);
fstream(const char* nomefile, int modalita);
```

La stringa `nomefile` è il nome del file associato allo stream, mentre le modalità di apertura dello stream sono specificate da un tipo enum nella classe `ios`.

```
class ios {
public:
    enum open_mode {
        in,           // lettura
        out,          // scrittura
        ate,          // spostamento a EOF
        app,          // scrittura alla fine del file
        trunc,        // troncamento del file a lunghezza 0
        binary,       // i/o in binario, il default è text mode
        nocreate,     // fallisce se il file non esiste
        noreplace    // fallisce se il file esiste
    };
    ...
};
```

Le modalità di apertura di uno stream su file possono essere combinate tramite l'OR bitweise (cioè bit a bit). Per default, gli oggetti di `ifstream` sono aperti in lettura mentre quelli di `ofstream` sono aperti in scrittura. Un `fstream` può essere aperto sia in lettura che in scrittura. Per creare un file con un oggetto `fstream` per I/O è necessario usare la modalità `ios::trunc`. Vediamo alcuni esempi.

```
fstream file("dati.txt", ios::in|ios::out);
if (!file) cout << "Errore in apertura\n";
```

Questo frammento di codice apre il file `dati.txt` in i/o testuale.

```
ofstream file("dati.txt", ios::app|ios::nocreate|ios::binary);
if (file.bad()) cout << "Il file non esiste\n";
```

La modalità `ios::nocreate` comporta un errore non recuperabile (cioè `bad()` ritorna 1) se il file `dati.txt` non esiste. Inoltre, il file viene aperto per output binario con modalità di append alla fine.

Per quanto concerne la chiusura di un file, il metodo `close()` chiude esplicitamente un file: esso viene automaticamente invocato dal distruttore dello stream.

Il posizionamento in una cella di uno stream associato ad un file si effettua tramite i seguenti metodi.

```

class istream: public virtual ios {
public:
    long tellg();
    istream& seekg(long posizione);
    ...
};

class ostream: public virtual ios {
public:
    long tellp();
    ostream& seekp(long posizione);
    ...
};

```

Le costanti `ios::beg`, `ios::cur` e `ios::end` sono delle posizioni definite in `ios`:

`ios::beg` = posizione iniziale dello stream, cioè vale 0.

`ios::cur` = posizione corrente.

`ios::end` = posizione finale dello stream, cioè la cella successiva all'ultimo byte dello stream (cioè EOF).

I metodi `tellg()` e `tellp()` restituiscono il valore del contatore di posizione nello stream, mentre i metodi `seekg(int pos)` e `seekp(int pos)` portano lo stream nella posizione `pos`. Vediamo un esempio.

```

fstream f("dati.txt", ios::trunc|ios::in|ios::out);
// crea il file dati.txt se non esisteva
if (!f) cout << "Errore in apertura\n";
f << "Pippo";
cout << f.tellp() << endl; // stampa: 5
f.seekp(ios::beg);
f << "Topolino"; cout << f.tellp() << endl; // stampa: 8
f << " Pluto"; cout << f.tellp() << endl; // stampa: 14
f.seekg(ios::beg);
char c; while (f.get(c)) cout << c; // stampa: Topolino Pluto
cout << endl; f.clear(); f.seekg(ios::beg);
while (f >> c) cout << c; cout << endl; // stampa: TopolinoPluto
f.clear(); f.seekg(6);
f >> c; cout << c << endl; // stampa: n

```

Stream di stringhe

Si possono definire stream associati a stringhe, ossia sequenze di caratteri memorizzate in RAM (si parla anche di I/O in memoria). Il carattere nullo di terminazione gioca il ruolo di

8 LA GERARCHIA DI CLASSI PER L'I/O

marcatore di fine stream. Le classi da utilizzare sono: `istringstream`, `ostringstream` e `stringstream`, il file header che le dichiara è `<sstream>`. I costruttori sono i seguenti.

```
istringstream(const char*, int = ios::in);
ostringstream(int = ios::out);
ostringstream(string, int = ios::out);
stringstream(int = ios::in|ios::out);
stringstream(const char* s, int = ios::in|ios::out);
```

I metodi di scrittura/lettura sono quelli ereditati da `istream`, `ostream` e `iostream`. Il metodo `str()` applicato ad uno stream di stringhe ritorna la stringa associata allo stream. Vediamo un esempio.

```
stringstream ss;
ss << 236 << ' ' << 3.14 << " pippo ";
cout << ss.tellp() << ' ' << ss.tellg() << endl; // stampa: 17 0
// la stringa in memoria è: "236 3.14 pippo "
// la posizione di output è avanzata alla fine ios::end
// la posizione di input è ancora a ios::beg
int i; ss >> i; cout << i << endl; // stampa: 236
double d; ss >> d; cout << d << endl; // stampa: 3.14
string s; ss >> s; cout << "*" << s << "*\n"; // stampa: *pippo*
```

Capitolo 9

Libreria Qt

Un utilizzo professionale del linguaggio C++ necessariamente si appoggia sull'utilizzo di librerie di classi, oltre alle librerie definite dallo standard C++ quali STL e libreria di input/output che abbiamo già considerato. Esistono numerose librerie per C++, per vari ambiti e scopi applicativi. Alcuni esempi di librerie ad ampia diffusione sono: Boost, gtkmm, POCO C++ Libraries, Microsoft Foundation Class Library, etc. In questo capitolo tratteremo la libreria Qt come importante esempio di libreria non standard per il C++.

Qt è una libreria “cross-platform” diffusamente utilizzata in ambito professionale per lo sviluppo di interfacce utente grafiche (graphical user interface, GUI). Ad esempio, Qt è usata per lo sviluppo di software ad amplissima diffusione quale: Adobe Photoshop, Skype, VirtualBox e Mathematica, e da multinazionali dell'ICT quali: Google, HP, Samsung, Research In Motion. Qt è un software libero ed open source, distribuito con licenza GNU LGPL. Qt è una libreria nativa per il linguaggio C++, ma ci sono binding per Python (è il binding di maggior qualità), C, Java, PHP, C#, etc. Qt è una libreria sviluppata aderendo completamente al paradigma della programmazione ad oggetti ed alla programmazione guidata dagli eventi (event driven). Qt estende il C++ con i cosiddetti segnali e slot utilizzando il Meta Object Compiler (moc), che aggiunge al codice C++ della “meta information” usata da Qt per implementare il sistema di segnali e slot non disponibile nativamente in C++.

Questo capitolo intende fornire una rapida e semplificata introduzione a Qt prendendo in considerazione solamente gli aspetti più basilari di Qt mediante l'utilizzo di semplici esempi. Tutorial e documentazione ufficiali vengono forniti assieme al framework Qt. In particolare, la documentazione delle classi di Qt è molto chiara e dettagliata e deve fungere da riferimento principale nell'utilizzo di Qt. Ulteriore abbondante materiale introduttivo e di approfondimento si trova liberamente disponibile on-line. Si segnala che il framework Qt fornisce anche il tool *Qt designer* che permette di progettare GUIs in Qt assemblando componenti in maniera grafica seguendo il principio what-you-see-is-what-you-get e quindi producendo automaticamente il corrispondente codice C++. Fornisce inoltre l'IDE *Qt*

Creator che costituisce un ambiente di sviluppo specifico per applicazioni Qt.

9.1 Introduzione a Qt

Con l'ausilio di Qt, C++ raggiunge un livello superiore di semplicità, usabilità ed uniformità sulle varie piattaforme. La libreria Qt consiste di vari moduli di classi, di cui QtGUI è il modulo che riguarda le GUI. Tra gli altri moduli si segnalano:

- QtCore: il nucleo di classi per tutto il framework Qt.
- QtMultimedia: classi per la riproduzione e gestione di audio e video.
- QtNetwork: classi per le programmazione di rete, come QTcpSocket, QFtp, QNetworkRequest, etc.
- QtOpenCL, QDom, QSql, QtWebKit: interfacce per le più diffuse tecnologie software.

Qt include un insieme di classi (e template di classi) che offrono dei tipi di dato alternativi a quelli del C++ standard, in generale di più semplice utilizzo per lo sviluppo di una applicazione Qt. Possiamo menzionare: QChar, QString, QVector<T>, QVectorIterator, QList<T>, QLinkedList<T>, QSet<T>, QMap<Key, T>, QDir, QFile, QTextStream, QDate, QTime, QPoint, etc.

QObject è la classe alla base del modello ad oggetti di Qt. In particolare, QObject incorpora le funzionalità del meccanismo ad eventi signal/slot che richiedono il moc per generare codice C++. Signal e slot sono utilizzati per la comunicazione tra gli oggetti delle classi Qt. Il meccanismo di signal/slot è un elemento centrale di Qt ed è la caratteristica che lo differenzia maggiormente da altri framework per lo sviluppo di GUI. Nella programmazione ad eventi di una GUI, quando cambia lo stato di un qualche componente di una GUI, detto in generale *widget*, spesso si vuole che ciò provochi la notifica di tale cambiamento ad un altro widget. Più in generale, si vogliono che gli oggetti di qualsiasi tipo siano in grado di comunicare tra loro. Ad esempio, se un utente clicca un qualche pulsante “Close”, probabilmente ci si aspetta che venga invocata la funzione close() della finestra che contiene quel pulsante.

Alcuni framework implementano questa tipologia di comunicazione tra oggetti tramite la tecnica dei cosiddetti “callback”, che sono dei puntatori a funzione. Tuttavia il meccanismo dei callback presenta vari difetti che ostacolano una implementazione semplice ed efficace della programmazione ad eventi. In Qt, i signal/slot formano un meccanismo di comunicazione alternativo alla tecnica dei callback. Un widget emette un particolare segnale quando si verifica un corrispondente evento particolare nel widget. I widget Qt hanno molti segnali predefiniti, precisamente elencati e descritti nella documentazione delle classi Qt, ed è sempre possibile definire dei widget sottotipo per aggiungere ulteriori segnali. Uno slot è una funzione che viene chiamata in risposta ad un particolare segnale che è stato emesso.

I widget Qt hanno molti slot predefiniti, ed è pratica comune definire dei widget sottotipo per aggiungere altri slot. Segnali e slot sono debolmente accoppiati: un oggetto che emette un segnale non conosce né si preoccupa degli slot che eventualmente riceveranno tale segnale. Il meccanismo di connessione mediante una connect tra signal e slot assicura che se si collega un segnale ad uno slot, lo slot sarà chiamato con i parametri che caratterizzano il segnale al momento giusto.

Tutte le classi che ereditano da `QObject` o dalle sue sottoclassi (ad esempio, `QWidget`) possono contenere segnali e slot. I segnali vengono emessi dagli oggetti quando essi cambiano stato in un modo che può essere di interesse per altri oggetti che intendono reagire a tale cambiamento di stato. L'oggetto non fa altro che comunicare il suo cambiamento di stato e non sa nulla se qualche altro oggetto sta ricevendo i suoi segnali. Ciò permette quindi un netto ed efficace encapsulamento dell'informazione. Gli slot possono essere utilizzati per ricevere segnali, ma sono anche dei metodi ordinari di una classe. Proprio come un oggetto non sa se qualche altro oggetto riceve i suoi segnali, uno slot non sa se ha dei segnali ad esso collegati. Ciò assicura che le componenti Qt siano realmente indipendenti tra loro. È possibile collegare un qualsiasi numero di segnali per un singolo slot, ed un segnale può essere collegato ad un qualsiasi numero di slot. È anche possibile collegare un segnale direttamente ad un secondo segnale in modo che il secondo segnale emetta immediatamente il suo signal quando viene emesso il primo.

9.2 Hello world

Vediamo un esempio del programma Qt più semplice che consiste di una finestra minimale contenente una stringa.

```
#include <QApplication>
#include < QLabel >

int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    QPushButton hello("Hello world!");
    hello.show();
    return QApplication::exec();
}
```

Commentiamo le righe di questo programma:

- Il file header `QApplication` va sempre incluso, mentre `QLabel` è il file header corrispondente al widget Qt usata dal programma, che permette di mostrare un testo o una immagine.
- Ogni programma Qt deve costruire un oggetto `QApplication`, al cui costruttore sono passati gli argomenti `argc` e `argv` del metodo `main` del programma C++.

- hello è un widget di tipo QLabel creato mediante una chiamata al costruttore

```
QLabel(const QString& t, QWidget* parent = 0, Qt::WindowFlags f = 0)
```

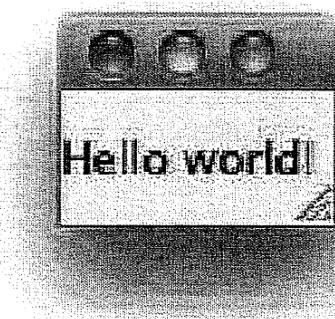
La stringa Hello world è convertibile al tipo QString. Il widget hello non ha parent widget, lui stesso è una window con frame e title bar.

- L'invocazione hello.show() rende visibile un widget, che altrimenti non è visibile di default.
- L'invocazione del metodo statico QApplication::exec() cede il controllo dal main a Qt.

Supponiamo di memorizzare il programma nel file main.cpp memorizzato in una folder con nome hello. La compilazione ed il linking del programma avvengono invocando la seguente sequenza di comandi da shell:

```
qmake -project  
qmake  
make
```

qmake è un programma fornito dal framework Qt che permette di generare un opportuno Makefile per il comando make che permette la corretta compilazione del programma. L'invocazione qmake -project genera il file hello.pro che a sua volta viene usato dall'invocazione qmake per generare il Makefile necessario alla compilazione e linking vero e proprio mediante il comando make. Questi passi di compilazione generano un programma eseguibile che mostra la GUI in figura (naturalmente l'effettiva GUI dipende dal sistema operativo sottostante).



9.3 Segnali e slot

Consideriamo il seguente programma.

9.3 Segnali e slot

```
#include <QApplication>
#include <QFont>
#include <QPushButton>
#include <QWidget>

int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    QWidget window;
    window.setWindowTitle("I'm a QWidget");
    window.resize(200, 120);
    QPushButton quit("Quit", &window);
    quit.setFont(QFont("Times", 18, QFont::Bold));
    quit.setGeometry(10, 40, 180, 40);
    QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
    window.show();
    return app.exec();
}
```

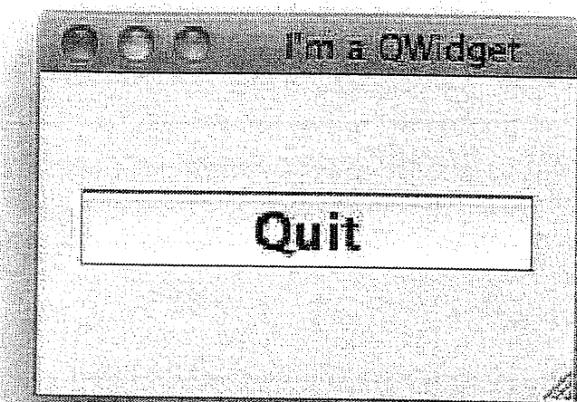
- QWidget è la classe base di tutti i widget. Una QWidget è un atomo di una GUI: riceve eventi dal sistema con cui interagisce (mouse, keyboard, touch screen, etc), e rappresenta se stessa sullo schermo. Una QWidget è detenuta dal suo cosiddetto parent. Una QWidget senza parent viene detta una independent window (con frame e taskbar). La posizione iniziale è controllata dal sistema.
- L'invocazione del metodo setWindowTitle() (che è un public slot di QWidget) permette di definire il titolo di window mentre resize() ridimensiona window.
- quit è un QPushButton, un widget con funzionalità di pulsante. quit ha come parent window, ovvero quit è figlio di window. Un figlio è sempre mostrato nell'area del suo parent, per default al top-left corner alla posizione (0,0).
- Il metodo setFont() permette di definire le proprietà delle font del pulsante quit, mentre l'invocazione quit.setGeometry(x, y, w, h) definisce (x, y) come coordinate del top-left corner di quit e (w, h) come dimensione di base ed altezza di quit.
- L'invocazione di connect(), metodo statico di QObject, la classe base di ogni oggetto Qt, stabilisce una connessione tra due QObject. Ogni QObject, e quindi ogni QWidget, può avere dei cosiddetti *signal* per spedire dei messaggi e dei cosiddetti (public) *slot* per ricevere dei messaggi. La documentazione Qt fornisce la lista dei segnali/slot disponibili per ogni classe. Il segnale clicked() emesso dal pulsante quit viene quindi connesso allo slot quit() di app:

```
QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
```

La connect permette di ottenere il seguente effetto: quando si clicka il pulsante quit l'applicazione app termina.

- L'invocazione `window.show()` invoca il metodo `show()` anche su tutti i figli di `window`.

La GUI del programma è nella seguente figura.



La precedente QWidget può anche essere progettata come una classe definita dal programmatore che eredita da QWidget nel seguente modo:

```
class MyWidget : public QWidget {
public:
    MyWidget(QWidget* parent = 0) : QWidget(parent) {
        setWindowTitle("I'm a QWidget");
        resize(200, 120);
        QPushButton* quit = new QPushButton(tr("Quit"), this);
        quit->setFont(QFont("Times", 18, QFont::Bold));
        quit->setGeometry(10, 40, 180, 30);
        connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    }
};

int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    MyWidget widget;
    widget.show();
    return app.exec();
}
```

9.4 Layout manager

Notiamo quindi che definiamo un costruttore con argomento il `QWidget` parent, dove il valore di default 0 significa essere a top-level (senza parent). Il corpo di questo costruttore include le chiamate a `setWindowTitle()` e `resize()`. `MyWidget` definisce un `QPushButton` come figlio mediante la chiamata del costruttore

```
QPushButton(tr("Quit"), this);
```

dove `tr("Quit")` marca la stringa `Quit` per possibili traduzioni di lingua a run-time. Nella connect

```
connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
```

`qApp` è una variabile globale dichiarata nel file header `QApplication` che punta all'unica istanza di `QApplication` del programma. Inoltre `quit` è una variabile locale, e non un campo dati. Naturalmente il `QPushButton` viene automaticamente distrutto quando viene distrutto `MyWidget`. Pertanto `MyWidget` non necessita della definizione esplicita di un distruttore.

9.4 Layout manager

Consideriamo la seguente classe `MyWidget`.

```
class MyWidget : public QWidget {
public:
    MyWidget(QWidget* parent = 0) : QWidget(parent) {
        QPushButton* quit = new QPushButton(tr("Quit"));
        quit->setFont(QFont("Times", 18, QFont::Bold));
        QLCDNumber* lcd = new QLCDNumber(2);
        lcd->setSegmentStyle(QLCDNumber::Filled);
        QSlider* slider = new QSlider(Qt::Horizontal);
        slider->setRange(0, 99); slider->setValue(0);
        connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
        connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
        QVBoxLayout* layout = new QVBoxLayout;
        layout->addWidget(quit);
        layout->addWidget(lcd);
        layout->addWidget(slider);
        setLayout(layout);
    }
};
```

- `QLCDNumber` è un widget che mostra dei numeri, in questo caso 2, con un look-and-feel LCD. L'invocazione `setSegmentStyle(QLCDNumber::Filled)` rende questo widget LCD più leggibile.

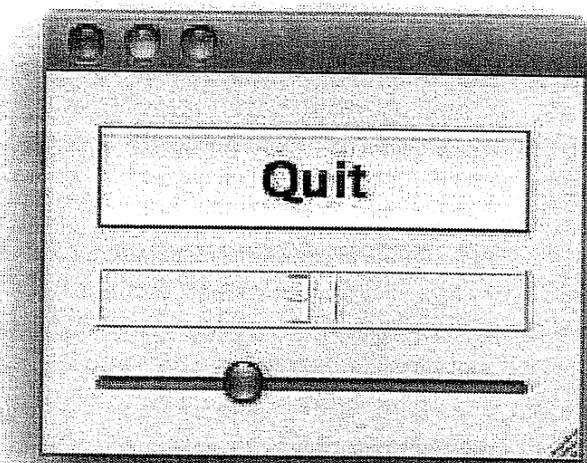
- Il widget `QSlider` permette di selezionare un valore intero in un range di valori. Nel nostro esempio, `slider` è un `QSlider` orizzontale con valori in [0, 99] e valore iniziale 0.
- Con la connect

```
connect (slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
```

il segnale `valueChanged(int)` del widget `slider` viene connesso allo slot `display()` del widget `lcd`. `slider` emette il segnale `valueChanged()` quando cambia il suo valore, quindi lo slot `display` è chiamato quando viene emesso questo segnale.

- `QVBoxLayout` è un cosiddetto *layout manager* (LM), ovvero un gestore della geometria dei figli di un widget. Inoltre il LM di un widget `w` gestisce il ridimensionamento dei figli di `w` quando viene ridimensionato `w`. Un `QLayout` (supertipo di `QVBoxLayout`), non è sottotipo di `QWidget` e quindi un LM non ha mai un parent. Con il metodo `addWidget()` aggiungo dei widget al controllo di un LM, mentre `QWidget::setLayout(layout)` installa `layout` come LM di `this` e rende `layout` figlio di `this`, in particolare tutti i widget sotto controllo di `layout` diventano figli di `this`.

La GUI del programma è nella seguente figura.



9.5 Connessioni tra segnali

Consideriamo il seguente programma composto dalle classi `LCDRange` e `MyWidget`.

9.5 Connessioni tra segnali

```
// lcdrange.h
#ifndef LCDRANGE_H
#define LCDRANGE_H

#include <QWidget>

class QSlider; // dichiarazione incompleta

class LCDRange : public QWidget {
    Q_OBJECT
public:
    LCDRange(QWidget* parent = 0);
    int value() const;
public slots:
    void setValue(int value);
signals:      // implicitamente protected
    void valueChanged(int newValue);
private:
    QSlider* slider;
};

#endif
```

```
// lcdrange.cpp
#include <QLCDNumber>
#include <QSlider>
#include <QVBoxLayout>
#include "lcdrange.h"

LCDRange::LCDRange(QWidget* parent) : QWidget(parent) {
    QLCDNumber* lcd = new QLCDNumber(2);
    lcd->setSegmentStyle(QLCDNumber::Filled);
    slider = new QSlider(Qt::Horizontal);
    slider->setRange(0, 99); slider->setValue(0);
    connect(slider, SIGNAL(valueChanged(int)),
            lcd, SLOT(display(int)));
    connect(slider, SIGNAL(valueChanged(int)),
            this, SIGNAL(valueChanged(int)));
    QVBoxLayout* layout = new QVBoxLayout;
    layout->addWidget(lcd); layout->addWidget(slider);
    setLayout(layout);
}

int LCDRange::value() const {
    return slider->value();
```

9 LIBRERIA QT

```
}

void LCDRange::setValue(int value) {
    slider->setValue(value);
}
```

```
// mywidget.h
#ifndef MYWIDGET_H
#define MYWIDGET_H

#include <QWidget>
class MyWidget : public QWidget {
public:
    MyWidget(QWidget* parent = 0);
};

#endif
```

```
// mywidget.cpp
#include <QApplication>
#include <QFont>
#include <QGridLayout>
#include <QPushButton>
#include <QWidget>
#include "lcdrange.h"
#include "mywidget.h"

MyWidget::MyWidget(QWidget* parent) : QWidget(parent) {
    QPushButton* quit = new QPushButton(tr("Quit"));
    quit->setFont(QFont("Times", 18, QFont::Bold));
    connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));
    QGridLayout* grid = new QGridLayout;
    LCDRange* previousRange = 0;
    for (int row = 0; row < 3; ++row) {
        for (int column = 0; column < 3; ++column) {
            LCDRange* lcdRange = new LCDRange;
            grid->addWidget(lcdRange, row, column);
            if (previousRange)
                connect(lcdRange, SIGNAL(valueChanged(int)),
                        previousRange, SLOT(setValue(int)));
            previousRange = lcdRange;
        }
    }
    QVBoxLayout* vblayout = new QVBoxLayout;
    vblayout->addWidget(quit); vblayout->addLayout(grid);
```

9.5 Connessioni tra segnali

```
setLayout(vblayout);  
}
```

```
// main.cpp  
#include <QApplication>  
#include "mywidget.h"  
  
int main(int argc, char* argv[]) {  
    QApplication app(argc, argv);  
    MyWidget widget;  
    widget.show();  
    return app.exec();  
}
```

- Nella dichiarazione della classe `LCDRange` notiamo la presenza della macro `Q_OBJECT`, che deve essere inclusa in tutte le classi che definiscono dei segnali o delle slot.
- La classe `LCDRange` include come public slot il metodo `void setValue(int)` che definisce il valore intero del `QSlider slider`. Inoltre, la classe `LCDRange` dichiara come signal, che è sempre marcato implicitamente come protected, un metodo `void valueChanged(int)`. Tale signal definito dal programmatore è automaticamente implementato dal Meta Object Compiler nel moc file.
- Nel costruttore di `LCDRange` è inclusa la connect

```
connect(slider, SIGNAL(valueChanged(int)),  
        this, SIGNAL(valueChanged(int)));
```

che permette di fare una connessione tra signal: quando `slider` di tipo `QSlider` emette il signal predefinito `void valueChanged(int)` viene anche emesso il signal `valueChanged(int)` dall'oggetto `this` di tipo `LCDRange`, che il programmatore aveva dichiarato nella classe `LCDRange`.

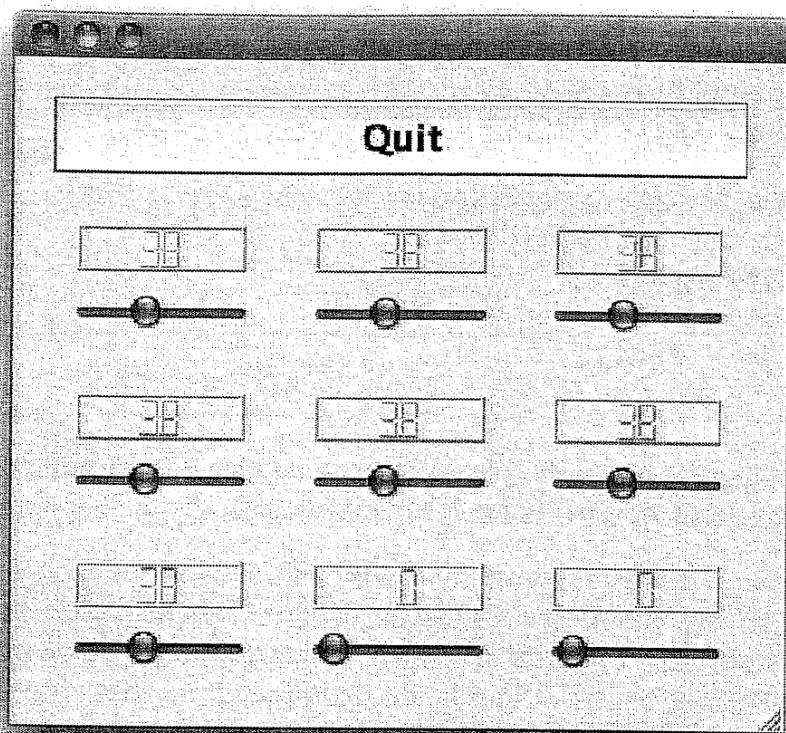
- Nel costruttore di `MyWidget`, all'interno del ciclo for è inclusa la connect

```
connect(lcdRange, SIGNAL(valueChanged(int)),  
        previousRange, SLOT(setValue(int)));
```

che permette di connettere il signal `void valueChanged(int)` di ogni `LCDRange` costruito con il public slot `void setValue(int)` del `LCDRange` costruito all'iterazione precedente. In questo modo riusciamo a definire una catena di segnali e slot tra i 9 `LCDRange` della GUI in modo tale che una variazione del valore di qualche `LCDRange` della griglia cambia il valore di tutti i `LCDRange` che lo precedono nella griglia.

9 LIBRERIA QT

La GUI del programma è nella seguente figura.



Capitolo 10

C++11

Bjarne Stroustrup, il creatore del C++, ha dichiarato che “C++11 appare come un nuovo linguaggio, in realtà semplicemente i pezzi ora si incastrano meglio”. In effetti, possiamo dire che in C++11 il nucleo del C++ è stato oliato in alcuni ingranaggi che prima provocavano qualche disturbo al programmatore. Tra le maggiori nuove caratteristiche, il C++11 ora supporta le lambda espressioni, i riferimenti rvalue, l’inferenza automatica di tipo, la sintassi di inizializzazione uniforme, le chiamate di costruttori nel corpo di costruttori, le funzioni default e delete, la keyword `nullptr`. La Standard Library di C++11 è stata rinnovata con nuovi algoritmi, classi contenitore, espressioni regolari, nuove classi di puntatori smart `shared_ptr` e `unique_ptr`, e, soprattutto, con una libreria che supporta la programmazione concorrente mediante multithreading e sincronizzazione.

Nel seguito analizzeremo brevemente i cambiamenti più significativi introdotti dallo standard C++11. A partire dalla versione 4.7 rilasciata a marzo 2012, il compilatore `g++` fornisce un supporto sperimentale al nuovo standard C++11, che può essere abilitato invocando il compilatore con l’opzione `-std=c++11`.

10.1 Lambda espressioni

Una lambda espressione, anche detta lambda funzione o funzione anonima, consente di definire delle funzioni a livello locale, dove esse sono invocate, senza definirne un identificatore. Una lambda espressione ha la seguente sintassi:

```
[capture list] (lista parametri) ->return-type { corpo }
```

dove `(lista parametri)` e `->return-type` sono opzionali. Se non vi sono parametri, le parentesi `()` possono essere omesse.

Il tipo di ritorno può essere omesso se il tipo di ritorno è `void` oppure se il corpo consiste di una unica istruzione `return` o se tutte le istruzioni di `return` ritornano espressioni dello stesso tipo.

```
[ ] (int x, int y) {return x+y;} // tipo di ritorno implicito: int
[ ] (int& x) {++x;}           // tipo di ritorno implicito: void
```

Una lambda espressione può usare variabili le dichiarate al suo esterno, e l'insieme di tali variabili usate (altresì dette catturate) viene detto una **chiusura** (closure in inglese: tale terminologia deriva da un concetto generale dei linguaggi di programmazione, in particolare nasce originariamente nei linguaggi funzionali). Il costrutto [capture list] elenca la lista delle variabili della closure:

[]	\\" nessuna variabile esterna catturata
[x, &y]	\\" x catturata per valore, y per riferimento
[&]	\\" tutte le variabili esterne catturate per riferimento
[=]	\\" tutte le variabili esterne catturate per valore
[&, x]	\\" tutte le variabili per riferimento, tranne x per valore

Vediamo alcuni esempi di utilizzo di lambda espressioni. Si supponga di voler contare quante lettere maiuscole occorrono in una stringa. Utilizzando il nuovo costrutto C++11 `for_each` per scorrere gli elementi di una collezione come un contenitore STL o un array di caratteri, definiamo una lambda espressione che determina se una lettera è in maiuscolo. Per ogni lettera maiuscola trovata, la lambda espressione incrementa la variabile `Uppercase` definita esternamente. La lambda espressione non ha tipo di ritorno esplicito, che è quindi implicitamente `void`.

```
int main() {
    char s [] = "Hello World";
    int Uppercase = 0; // modificata dalla lambda espressione
    for_each(s, s+sizeof(s), [&Uppercase] (char c) {
        if (isupper(c)) Uppercase++;
    });
    cout << Uppercase << " lettere maiuscole in: " << s << endl;
}
```

Si tratta quindi della definizione di una funzione anonima (senza nome) definita dentro un'altra funzione. La notazione `[&Uppercase]` ottiene un riferimento alla variabile esterna `Uppercase` che permette quindi alla lambda espressione di modificarla. Vediamo un ulteriore esempio.

```
vector<int> v = {50, -10, 20, -30};

// sort di default
std::sort(v.begin(), v.end());
```

10.2 Riferimenti rvalue

```
// v diventa { -30, -10, 20, 50 }

// sort per valore assoluto
std::sort(v.begin(), v.end(), [](int a, int b) {
    return abs(a)<abs(b); });
// v diventa { -10, 20, -30, 50 }
```

Infine consideriamo il seguente esempio.

```
std::vector<int> v = {6, 2, -1, 0, 6};
int totale = 0; int a = 5;
std::for_each(v.begin(), v.end(), [&, a, this](int x) {
    totale += x * a * this->some_function();
});
```

Si noti quindi che `totale` può essere modificata perché catturata per riferimento nella chiusura, mentre la variabile `a` è (giustamente) catturata per valore. Osserviamo inoltre che il puntatore `this` viene catturato per valore, e questo è l'unico modo in cui è possibile catturarlo in una lambda espressione. Ovviamente `this` può essere catturato se la lambda espressione viene definita nel corpo di qualche metodo di istanza, ossia in un blocco dove è definito il puntatore `this`.

10.2 Riferimenti rvalue

In C++, una variabile di tipo riferimento non costante può essere associata solamente ad un lvalue, mentre un riferimento costante può essere associato sia a lvalue che rvalue. Il C++11 introduce un nuovo tipo riferimento le cui variabili sono chiamate riferimenti rvalue e la cui sintassi per un tipo `T` è `T&&`. I riferimenti rvalue (non costanti) possono essere associati solamente a rvalues, ad esempio oggetti temporanei anonimi e valori di tipi predefiniti.

```
T a;
T f();
T& r1 = a; // OK: a è un lvalue
T& r2 = f(); // ILLEGALE: f() è un rvalue

T&& rr1 = f(); // OK
T&& rr2 = a; // ILLEGALE: a è un lvalue
```

Il motivo principale per l'aggiunta di questi riferimenti rvalue viene dalla semantica di una operazione di “move”. Diversamente dalla tradizionale copia, una operazione di move vuole trasferire il contenuto di un oggetto sorgente ad un qualche oggetto target, lasciando quindi l'oggetto sorgente in uno stato di “vuoto”. In alcuni casi è possibile effettuare una

operazione di move piuttosto che fare una costosa operazione di copia. Consideriamo ad esempio una operazione di swap tra stringhe.

```
template<class T> swap(T& a, T& b) {
    T tmp(a); // abbiamo due copie di a
    a = b;     // abbiamo due copie di b
    b = tmp;   // abbiamo due copie di tmp (ovvero a)
}
```

Evidentemente questa implementazione dello swap è piuttosto costosa, comportando tre operazioni di copia e l'allocazione in memoria di un oggetto temporaneo. Se T è un tipo per cui la copia è costosa, come un vector o una stringa, tale operazione di swap può diventare inaccettabilmente costosa. In C++11 possiamo definire dei costruttori "move" e delle assegnazioni "move" per trasferire piuttosto che copiare i loro argomenti sfruttando il tipo riferimento rvalue.

```
template<class T> class vector {
    // ...
    vector(const vector&); // costruttore di copia
    vector(vector&&);   // costruttore move
    vector& operator=(const vector&); // assegnazione di copia
    vector& operator=(vector&&);   // assegnazione move
};
```

L'idea di una assegnazione move $a = b$; è che invece che a diventi una copia di b, a dovrebbe semplicemente essere un nuovo identificatore per la rappresentazione di b mentre b viene "svuotato", ovvero diventa un identificatore per un default poco costoso del proprio tipo. Ad esempio, per le stringhe, l'assegnazione move $s1=s2$; potrebbe fare la delete della stringa puntata da s1, quindi far puntare s1 alla stringa puntata da s2, e rendere s2 la stringa vuota.

La richiesta di invocare costruttori move ed assegnazioni move viene fatta al compilatore nel seguente modo:

```
template<class T> void swap(T& a, T& b) { // swap basato su move
    T tmp(std::move(a)); // costruttore move
    a = std::move(b);   // assegnazione move
    b = std::move(tmp); // assegnazione move
}
```

`move(x)` informalmente significa "`x` può essere trattato come un rvalue". La funzione template `move` è definita nel namespace standard `std` ed il suo compito è semplicemente quello di accettare un qualsiasi argomento lvalue o rvalue e ritornarlo come un riferimento rvalue.

La libreria standard del C++11 Standard Library usa ripetutamente tali operazioni di move in molti contenitori ed algoritmi generici.

10.3 Inferenza automatica di tipo

In C++ è sempre obbligatorio specificare esplicitamente il tipo di una variabile nel momento in cui la si dichiara. Tuttavia, in molti casi la dichiarazione di una variabile include anche una inizializzazione. C++11 sfrutta la dichiarazione con simultanea inizializzazione permettendo di dichiarare variabili senza specificarne il tipo usando la keyword `auto`.

```
auto x = 0;      // x ha tipo int perché 0 è un valore di tipo int
auto c = 'f';   // char
auto d = 0.7;   // double
auto debito_nazionale = 200000000000L; // long int
auto y = qt_obj.qt_fun(); // y ha il tipo di ritorno di qt_fun
```

L'inferenza automatica di tipo può essere principalmente sfruttata quando il tipo della variabile è effettivamente una inutile verbosità o quando è automaticamente generato nei template. Ad esempio, in

```
void fun(const vector<int> &vi) {
    vector<int>::const_iterator ci=vi.begin();
    ...
}
```

possiamo invece dichiarare un iteratore semplicemente come:

```
auto ci=vi.begin();
```

Inoltre la keyword `decltype` può essere usata per determinare staticamente (cioè a compile-time) il tipo di qualche espressione.

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

L'uso di `auto` vs `decltype` è riassunto nel seguente esempio.

```
const std::vector<int> v(1);
auto a = v[0];           // a ha tipo int
decltype(v[1]) b = 1;   // b ha tipo const int&
auto c = 0;              // c ha tipo int
auto d = c;              // d ha tipo int
decltype(c) e;           // e ha tipo int
decltype(0) f;           // f ha tipo int
```

10.4 Inizializzazione uniforme

Il C++ ha almeno quattro diverse notazioni di inizializzazione, ed alcune di esse hanno dei tratti in comune.

1. std::string s("hello");
int m = int(); // inizializzazione di default
2. std::string s = "hello";
int x = 5;
3. int arr[4] = {0,1,2,3};
4. class X {
 int x;
 S(): x(0) {}
};

Questa proliferazione di sintassi per l'inizializzazione può generare confusione, non solamente tra i novizi del linguaggio. Per ovviare a ciò, il C++11 introduce una nuova notazione uniforme per l'inizializzazione aggregata basata su una lista di valori delimitata da parentesi graffe. Tale notazione è illustrata nei seguenti esempi:

```
struct S {
    int a;
    int b;
    // S(int x, int y): x(a), y(b) {}
};

S x{0,0}; // equivalente a: S x(0,0);

S fun() {
    return {1,2}; // equivalente a: return S(1,2);
}

// inizializzazione di array dinamico
int* a = new int[3] {1,2,0};

class X {
    int a[4];
public:
    X() : a{1,2,3,4} {} // inizializzazione di campo dati array
};
```

Per quanto riguarda i contenitori, in C++11 possiamo evitare l'usuale lunga lista di invocazioni di `push_back()` nel seguente modo:

10.5 default e delete

```
// inizializzazione di contenitori in C++11
std::vector<string> vs = {"first", "second", "third"};
std::map<string, string> singers =
{ {"LadyGaga", "347 0123456"}, {"Rihanna", "348 9876543"} };

void fun(std::list<double> l);
fun({0.34, -3.2, 5, 4.0});
```

In modo analogo, C++11 permette l'inizializzazione dei campi dati di una classe simultaneamente alla loro dichirazione:

```
class C {
    int a=7; // C++11 only
public:
    C();
};
```

10.5 default e delete

Sappiamo che per ogni classe sono sempre disponibili le seguenti funzioni: (1) il costruttore standard (che è di default), (2) il costruttore di copia standard, (3) l'assegnazione standard, (4) il distruttore standard. È possibile ridefinire le versioni standard di queste funzioni, mentre non vi è controllo sulla creazione di queste funzioni standard: ad esempio, abbiamo già osservato che per rendere gli oggetti di una classe non copiabili è necessario dichiarare come privati il costruttore di copia e l'assegnazione. In C++11, tali funzioni si possono rendere esplicitamente di default oppure non disponibili.

```
class A {
public:
    A(int) {} // costruttore ad 1 argomento
    A() = default; // costruttore altrimenti non disponibile
    virtual ~A() = default; // distruttore virtuale standard
};
```

La dichiarazione `=default` richiede al compilatore di generare l'implementazione di default per la funzione. Le funzioni di default hanno un vantaggio duplice: (1) sono più efficienti delle implementazioni di default definite dal programmatore; (2) dispensano il programmatore dal compito di fornire esplicitamente una funzione di default, qualora questo sia reso necessario dal codice.

Inoltre possiamo rendere tali funzioni non disponibili mediante la sintassi `=delete`. Le funzioni delete sono utili per evitare la copia di oggetti.

```
class NoCopy {
public:
    NoCopy& operator =(const NoCopy&) = delete;
    NoCopy (const NoCopy&) = delete;
};

NoCopy a,b;
NoCopy b(a); // errore in compilazione
b=a;          // errore in compilazione
```

Il modificatore `=delete` può anche essere usato per proibire la chiamata di una funzione qualsiasi. Questo può rivelarsi utile in situazioni simili alla seguente.

```
class OnlyDouble {
public:
    void f(double d);
    template<class T> void f(T) = delete;
};
```

10.6 Overriding esplicito

Nel seguente codice:

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

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

sappiamo che il metodo `D::m(int)` non è un overriding di `B::m(double)` ma è invece una seconda funzione virtuale per la classe derivata `D`. Una tale situazione può essere inavvertitamente provocata dal progettista della classe `D`, che intendeva invece definire l'overriding del metodo virtuale `B::m(double)`. Il C++11 aggiunge quindi il modificatore `override` per dichiarare esplicitamente quando una funzione virtuale deve necessariamente essere un overriding.

```
class B {
public:
```

10.7 nullptr

```
virtual void m(double);  
};  
  
class D: public B {  
public:  
    virtual void m(int) override; // ILLEGALE  
};
```

Quindi, se il compilatore non trova una funzione virtuale di cui viene effettivamente fatto l'overriding, segnala una illegalità.

Si segnala inoltre che C++11 permette di marcare una funzione virtuale `m()` dichiarata in qualche classe `C` con il modificatore `final`: questa marcatura proibisce alle classi che derivano da `C` di effettuare l'overriding di `m()`.

```
class B {  
public:  
    virtual void m(int) final;  
};  
  
class D: public B {  
public:  
    virtual void m(int); // ILLEGALE  
};
```

10.7 nullptr

In C++11, la keyword `nullptr` sostituisce la macro per il preprocessore `NUL` — che può introdurre bug nel porting tra piattaforme diverse, perché può essere espanso come `0` o come `((void*)0)` — ed il valore `0` che sono sempre stati usati come surrogati del puntatore nullo. Ora `nullptr` è fortemente tipizzato.

```
void f(int); // #1  
void f(char*); // #2  
  
f(0); // in C++: quale f è invocata?  
f(nullptr); // in C++11: chiamata non ambigua a f #2
```

È possibile usare `nullptr` per tutti le categorie di puntatori, inclusi i puntatori a funzione.

```
const char* pc = str.c_str();  
if (pc != nullptr) cout << pc << endl;  
void (*pmf)() = nullptr; // puntatore a funzione
```

`nullptr` è di tipo `nullptr_t`, che è implicitamente convertibile a qualsiasi tipo puntatore, mentre non è implicitamente convertibile ai tipi integrali, eccetto `bool`. Per compatibilità all'indietro, o rimane un valore valido per i puntatori nulli.

10.8 Chiamate di costruttori

In C++11 un costruttore può invocare un altro costruttore della stessa classe, un meccanismo noto come *delegation* e disponibile in linguaggi come Java e C#.

```
class C {
    int x, y;
    char* p;
public:
    C(int v, int w) : x(v), y(w), p(new char [5]) {}
    C(): C(0,0) {}
    C(int v): C(v,0) {}
};
```

Si noti che lo stesso effetto ottenuto con il meccanismo di delegation dei costruttori si poteva già ottenere mediante l'uso di argomenti di default nei costruttori. Con la delegation, gli argomenti di default fanno quindi parte dell'implementazione del costruttore piuttosto che della sua segnatura, e ciò può risultare un beneficio per i progettisti della classe.

Capitolo 11

Esercizi Riepilogativi

In questo capitolo conclusivo vengono presentati alcuni esercizi riepilogativi che affrontano gli argomenti trattati nel testo. Alcuni esercizi sono tratti da passati appelli d'esame. Le soluzioni degli esercizi sono proposte nell'Appendice A. L'ordine di presentazione degli esercizi non è significativo.

Esercizio 11.1. Definire una classe `IntMod` i cui oggetti rappresentano numeri interi modulo un dato intero. Devono essere disponibili gli operatori di somma e moltiplicazione tra oggetti di `IntMod`. Definire inoltre opportuni convertitori di tipo affiché gli oggetti di `IntMod` siano liberamente utilizzabili assieme ad espressioni di tipo primitivo `int` e valga la seguente condizione: quando in una espressione compaiono sia espressioni intere che oggetti di `IntMod` il tipo dell'espressione dovrà essere intero.

Esercizio 11.2. Il seguente programma compila ed esegue correttamente. Quali stampe produce in output la sua esecuzione?

```
class C {
private:
    int d;
public:
    C(string s="") : d(s.size()) {}
    explicit C(int n) : d(n) {}
    operator int() {return d;}
    C operator+(C x) {return C(d+x.d);}
};

int main() {
    C a, b("pipetto"), c(3);
    cout << a << ' ' << b << ' ' << c+4 << ' ' << c+b;
}
```

11 ESERCIZI RIEPILOGATIVI

Esercizio 11.3. Il seguente programma compila ed esegue correttamente. Quali stampe produce in output la sua esecuzione?

```
class It {
    friend class C;
public:
    bool operator<(It i) const {return index < i.index;}
    It operator++(int) { It t = *this; index++; return t; }
    It operator+(int k) {index = index + k; return *this; }
private:
    int index;
};

class C {
public:
    C(int k) {
        if (k>0) {dim=k; p = new int[k];}
        for(int i=0; i<k; i++) *(p+i)=i;
    }
    It begin() const { It t; t.index = 0; return t; }
    It end() const { It t; t.index = dim; return t; }
    int& operator[](It i) {return *(p + i.index);}
private:
    int* p;
    int dim;
};

int main() {
    C c1(4), c2(8);
    for(It i = c1.begin(); i < c1.end(); i++) cout << c1[i] << ' ';
    cout << "UNO\n";
    It i = c2.begin();
    for(int n=0; i < c2.end(); ++n, i = i+n) cout << c2[i] << ' ';
    cout << "DUE";
}
```

Esercizio 11.4. Si considerino le seguenti dichiarazioni e definizioni:

```
class Nodo {
private:
    Nodo(string st="***", Nodo* s=0, Nodo* d=0): info(st), sx(s),
                                                    dx(d) {}
    string info;
    Nodo* sx;
    Nodo* dx;
```

```

};

class Albero {
public:
    Albero(): radice(0) {}
    Albero(const Albero&); // dichiarazione costruttore di copia
private:
    Nodo* radice;
};

```

Quindi, gli oggetti della classe `Albero` rappresentano *alberi binari ricorsivamente definiti di stringhe*. Si ridefinisca il costruttore di copia di `Albero` in modo che esegua copie profonde.

Esercizio 11.5. Definire una classe `Data` i cui oggetti rappresentano una data con giorno della settimana (lun-mar...-dom). La classe `Data` deve rendere disponibili:

- opportuni costruttori;
- metodi di selezione per ottenere giorno della settimana, giorno, mese, anno di una data;
- l'operatore di output;
- l'operatore di uguaglianza;
- l'operatore relazionale < che ignori il giorno della settimana.

Esercizio 11.6. Definire una classe `Vettore` i cui oggetti rappresentano vettori di interi. La classe `Vettore` deve rendere disponibili un costruttore di default e gli operatori di uguaglianza, output e indicizzazione. Inoltre `Vettore` deve rendere disponibili un costruttore di copia profonda e l'assegnazione profonda.

Esercizio 11.7. Si considerino le seguenti definizioni:

```

class A {
public:
    virtual ~A() {}
};

class B: public A {};
class C: virtual public B {};
class D: virtual public B {};
class E: public C, public D {};

char F(A* p, C& r) {
    B* punt = dynamic_cast<B*>(p);
}

```

11 ESERCIZI RIEPILOGATIVI

```
try{
    E& s = dynamic_cast<E&>(r);
}
catch(bad_cast) {
    if(punt) return 'O';
    else return 'M';
}
if(punt) return 'R';
return 'A';
}
```

Si consideri inoltre il seguente `main()` incompleto, dove `?` denota una incognita:

```
int main() {
    A a; B b; C c; D d; E e;
    cout << F(?,?) << F(?,?) << F(?,?) << F(?,?);
}
```

Definire opportunamente le chiamate in tale `main()` usando gli oggetti locali `a`, `b`, `c`, `d`, `e` in modo tale che la sua esecuzione non provochi errori a run-time e produca in output la stampa ROMA.

Esercizio 11.8. Sia `B` una classe con distruttore pubblico e virtuale e sia `C` una sottoclasse di `B`. Definire una funzione `int Fun(vector<B*>& v)` con il seguente comportamento: in ogni invocazione `Fun(v)`,

- se `v` è vuoto allora ritorna 0;
- se `v` non è vuoto denotiamo con `T*` il tipo dinamico di `v[0]`; in questo caso ritorna il numero di elementi di `v` che hanno un tipo dinamico `T1*` tale che `T1` è un sottotipo di `C` diverso da `T`.

Ad esempio, il seguente programma deve compilare ed eseguire correttamente e la sua esecuzione deve provocare le stampe indicate.

```
class B {public: virtual ~B() {} };
class C: public B {};
class D: public B {};
class E: public C {};

int Fun(vector<B*> &v) { ... }

int main() {
    vector<B*> u, v, w;
    cout << Fun(u); // stampa 0
    cout << Fun(v); // stampa 1
    cout << Fun(w); // stampa 3
}
```

```

B b; C c; D d; E e; B *p = &e, *q = &c;
v.push_back(&c); v.push_back(&b); v.push_back(&d);
v.push_back(&c); v.push_back(&e); v.push_back(p);
cout << Fun(v); // stampa 2
w.push_back(p); w.push_back(&d); w.push_back(q);
w.push_back(&e);
cout << Fun(w); // stampa 1
}

```

Esercizio 11.9. Definire una superclasse Auto i cui oggetti rappresentano generiche automobili e due sue sottoclassi Benzina e Diesel, i cui oggetti rappresentano automobili alimentate, rispettivamente, a benzina e a diesel (ovviamente non esistono automobili non alimentate e si assume che ogni auto è o benzina o diesel). Ci interesserà l'aspetto fiscale delle automobili, cioè il calcolo del bollo auto. Queste classi devono soddisfare le seguenti specifiche:

- Ogni automobile è caratterizzata dal numero di cavalli fiscali. La tassa per cavallo fiscale è unica per tutte le automobili (sia benzina che diesel) ed è fissata in 5 euro. La classe Auto fornisce un metodo tassa() che ritorna la tassa di bollo fiscale per l'automobile di invocazione.
- La classe Diesel è dotata (almeno) di un costruttore ad un parametro intero x che permette di creare un'auto diesel di x cavalli fiscali. Il calcolo del bollo fiscale per un'auto diesel viene fatto nel seguente modo: si moltiplica il numero di cavalli fiscali per la tassa per cavallo fiscale e si somma una addizionale fiscale unica per ogni automobile diesel fissata in 100 euro.
- Un'auto benzina può soddisfare o meno la normativa europea Euro4. La classe Benzina è dotata di (almeno) un costruttore ad un parametro intero x e ad un parametro booleano b che permette di creare un'auto benzina di x cavalli fiscali che soddisfa Euro4 se b vale true altrimenti che non soddisfa Euro4. Il calcolo del bollo fiscale per un'auto benzina viene fatto nel seguente modo: si moltiplica il numero di cavalli fiscali per la tassa per cavallo fiscale; se l'auto soddisfa Euro4 allora si detrae un bonus fiscale unico per ogni automobile benzina fissato in 50 euro, altrimenti non vi è alcuna detrazione.

Si definisca inoltre una classe ACI i cui oggetti rappresentano delle filiali ACI addette all'incasso dei bolli auto. Ogni oggetto ACI è caratterizzato da un vector di puntatori ad auto, cioè un oggetto `vector<Auto*>`, che rappresenta la lista delle automobili gestite dalla filiale ACI. La classe ACI fornisce un metodo aggiungiAuto(const Auto& a) che aggiunge l'auto a alla lista gestita dalla filiale di invocazione. Inoltre, la classe ACI fornisce un metodo incassaBolli() che ritorna la somma totale dei bolli che devono pagare tutte le auto gestite dalla filiale di invocazione.

11 ESERCIZI RIEPILOGATIVI

Definire infine un esempio di `main()` in cui viene costruita una filiale ACI a cui vengono aggiunte quattro automobili in modo tale che l'incasso dei bolli ammonti a 1600 euro.

Esercizio 11.10. Si considerino le seguenti definizioni di template di classe.

```
template<class T> class D; // dichiarazione incompleta

template<class T1, class T2>
class C {
    friend class D<T1>;
private:
    T1 t1;
    T2 t2;
};

template<class T>
class D {
public:
    void m() {C<T,T> c; cout << c.t1 << c.t2;}
    void n() {C<int,T> c; cout << c.t1 << c.t2;}
    void o() {C<T,int> c; cout << c.t1 << c.t2;}
    void p() {C<int,int> c; cout << c.t1 << c.t2;}
    void q() {C<int,double> c; cout << c.t1 << c.t2;}
    void r() {C<char,double> c; cout << c.t1 << c.t2;}
};
```

Determinare se i seguenti `main()` compilano correttamente o meno.

- (1) int main() { D<char> d; d.m(); }
- (2) int main() { D<char> d; d.n(); }
- (3) int main() { D<char> d; d.o(); }
- (4) int main() { D<char> d; d.p(); }
- (5) int main() { D<char> d; d.q(); }
- (6) int main() { D<char> d; d.r(); }

Esercizio 11.11. Definire una superclasse `Biglietto` i cui oggetti rappresentano generici biglietti d'ingresso per uno spettacolo (ad esempio un concerto) e due sue sottoclassi `PostoNumerato` e `PostoNonNumerato`, i cui oggetti rappresentano, rispettivamente, biglietti per posti numerati e non numerati. Ci interesserà il costo di tali biglietti per un certo spettacolo. Queste classi devono soddisfare le seguenti specifiche:

- Ogni biglietto è caratterizzato dal nome dell'acquirente e dalla collocazione del posto in platea o galleria.

- Tutti i posti numerati sono in platea, mentre i posti non numerati possono essere sia in platea che in galleria (quindi in platea vi possono essere sia posti numerati che non numerati).
- Ogni biglietto per un posto numerato (quindi solo di platea) è caratterizzato dalla fila del posto.
- Come detto, ogni biglietto per un posto non numerato può essere di galleria o di platea. Inoltre, ogni biglietto per un posto non numerato è caratterizzato dall'essere a prezzo ridotto o pieno.

Definire inoltre una classe `Spettacolo` i cui oggetti rappresentano uno specifico spettacolo. La classe `Spettacolo` deve soddisfare le seguenti specifiche.

- Ogni spettacolo è caratterizzato da una base di prezzo B per i biglietti, da una addizionale di prezzo A , dal numero massimo di posti numerati, dal numero di fila n che caratterizza i posti numerati di prima fila, cioè tale che ogni posto numerato con un numero di fila $\leq n$ è un posto numerato di prima fila, ed infine dal numero di posti numerati venduti. Inoltre, ogni spettacolo è caratterizzato da una `list` di puntatori a biglietti, cioè un oggetto `list<Biglietto*>`, che memorizza la lista dei biglietti venduti per lo spettacolo.
- `Spettacolo` fornisce un metodo `void aggiungiBiglietto(const Biglietto&)` con il seguente comportamento: ogni invocazione `s.aggiungiBiglietto(b)` aggiunge il biglietto `b` alla lista dei biglietti venduti per lo spettacolo `s` secondo la seguente regola:
 - Se `b` è un biglietto per un posto non numerato allora `b` viene sempre aggiunto alla lista (si suppone che i posti non numerati siano illimitati).
 - Se `b` è un biglietto per un posto numerato allora `b` viene aggiunto alla lista se vi sono ancora posti numerati disponibili, ed in tal caso, naturalmente, viene anche aggiornato il numero di posti numerati venduti. Altrimenti, cioè quando non vi sono più posti numerati disponibili, `b` semplicemente non viene aggiunto alla lista dei biglietti venduti.
- La classe `Spettacolo` fornisce un metodo `double prezzo(const Biglietto&)` con il seguente comportamento: ogni invocazione `s.prezzo(b)` ritorna il prezzo del biglietto `b` per lo spettacolo `s` come segue:
 - un biglietto per un posto numerato di prima fila costa $2 * A + 2 * B$ mentre un posto numerato non di prima fila costa $2 * A$;
 - un biglietto per un posto non numerato di galleria costa B , di platea $B + A$, e se si tratta di un biglietto a prezzo ridotto in entrambi i casi viene detratto $A/2$.

11 ESERCIZI RIEPILOGATIVI

- Infine, la classe Spettacolo fornisce un metodo double incasso() con il seguente comportamento: ogni invocazione s.incasso() ritorna l'incasso per i biglietti finora venduti (cioè quelli memorizzati nella lista dei biglietti venduti).

Definire infine un esempio di main() in cui viene istanziato un oggetto s di Spettacolo che specifica i parametri di uno spettacolo, vengono costruiti quattro biglietti che vengono aggiunti allo spettacolo s e viene quindi calcolato e stampato l'incasso per quei biglietti.

Esercizio 11.12. Il seguente programma compila ed esegue correttamente.

```
class A {
    friend class C;
private:
    int k;
public:
    A(int x=2) : k(x) {}
    void m(int x=3) {k=x;}
};

class C {
private:
    A* p;
    int n;
public:
    C(int k=3) {if (k>0) {p = new A[k]; n=k;}}
    A* operator->() const {return p;}
    A& operator*() const {return *p;}
    A* operator+(int i) const {return p+i;}
    void F(int k, int x) {if (k<n) p[k].m(x);}
    void stampa() const {
        for(int i=0; i<n; i++) cout << p[i].k << ' ';
    }
};

int main() {
    C c1; c1.F(2,9);
    C c2(4); c2.F(0,8);
    *c1=*c2;
    (c2+3)->m(7);
    c1.stampa(); cout << "UNO\n";
    c2.stampa(); cout << "DUE\n";
    c1=c2;
    *(c2+1)=A(3);
    c1->m(1);
    *(c2+2)=*c1;
    c1.stampa(); cout << "TRE\n";
}
```

```
c2.stampa(); cout << "QUATTRO";  
}
```

Quali stampe produce in output la sua esecuzione?

Esercizio 11.13. Si ricorda che nella gerarchia di classi per l'I/O la classe base astratta `ios` ha il distruttore virtuale. Si definisca una classe `C` che soddisfa le seguenti specifiche.

1. Un oggetto della classe `C` è caratterizzato da un vector di puntatori a `ios` e dal numero massimo di puntatori che questo vector può contenere. Deve essere disponibile un costruttore ad un argomento intero `k`, con un valore di default positivo, che determina il numero massimo `k` di puntatori che il vector può contenere.
2. Deve essere disponibile un metodo `void insert(ios&)` con il seguente comportamento: una invocazione `c.insert(s)` inserisce nel vector di `c` un puntatore a `s` quando valgono entrambe le seguenti condizioni (altrimenti lascia inalterato il vector):
 - (a) il vector può contenere ancora elementi rispetto al numero massimo possibile;
 - (b) se `D&` è il tipo dinamico di `s` allora il tipo `D` è diverso sia da `fstream` che da `stringstream`.
3. Deve essere disponibile un template di metodo `int conta(T&)`, dove `T` è un parametro di tipo, con il seguente comportamento: ogni invocazione `c.conta(t)` ritorna il numero di puntatori del vector di `c` che hanno un tipo dinamico `D*` tale che il tipo `D` è un sottotipo del tipo del parametro attuale `t`.

Ad esempio, il seguente `main()` deve compilare ed eseguire correttamente provocando le stampe indicate:

```
int main() {  
    ifstream f("pippo"); ofstream g("mandrake");  
    fstream h("pluto"), i("zagor");  
    ostream* p = &g;  
    stringstream s;  
    C c(10);  
    c.insert(f); c.insert(g); c.insert(h);  
    c.insert(i); c.insert(*p); c.insert(s);  
    istream& r=f;  
    cout << c.conta(r); // stampa: 1 (è il puntatore all'oggetto f)  
}
```

Esercizio 11.14. Si considerino le seguenti definizioni.

11 ESERCIZI RIEPILOGATIVI

```
class Z {
public:
    Z() {cout << "Z() ";}
    Z(const Z& x) {cout << "Zc ";}
};

class A {
private:
    Z w;
public:
    A() {cout << "A() ";}
    A(const A& x) {cout << "Ac ";}
};

class B: virtual public A {
private:
    Z z;
public:
    B() {cout << "B() ";}
    B(const B& x) {cout << "Bc ";}
};

class C: virtual public A {
private:
    Z z;
public:
    C() {cout << "C() ";}
};

class D: public B, public C {
public:
    D() {cout << "D() ";}
    D(const D& x): C(x) {cout << "Dc ";}
};

int main() {
    B b1; C c1; cout << "***0\n";
    B b2 = b1; cout << "***1\n";
    C c2 = c1; cout << "***2\n";
    D d1; cout << "***3\n";
    D d2 = d1; cout << "***4";
}
```

Il precedente programma compila ed esegue correttamente. Si scrivano le stampe prodotte

dalla sua esecuzione.

Esercizio 11.15.

Si consideri la seguente realtà. Un supermercato prevede la possibilità per i clienti di dotarsi di fidelity card e diventare quindi clienti "fedeli" al supermercato. La fidelity card permette di partecipare alla raccolta punti cosicché al raggiungimento di una certa soglia punti i clienti fedeli vinceranno un premio. Inoltre i clienti fedeli godono di uno sconto incondizionato del 5% sulla spesa. Si chiede di modellare tale realtà aderendo alle seguenti specifiche.

1. Definire una classe `Prodotto` i cui oggetti rappresentano un prodotto in vendita nel supermercato. Un prodotto è quindi caratterizzato dal suo prezzo di vendita.
2. Definire una classe `Cliente` i cui oggetti rappresentano un generico cliente del supermercato. Un cliente è caratterizzato dal "carrello della spesa", cioè dalla lista dei prodotti acquistati nel corso di una spesa, rappresentabile tramite un vettore di oggetti `Prodotto`.
 - (a) La classe `Cliente` rende disponibile un metodo `double spesaTotale()` con il seguente comportamento: `c.spesaTotale()` ritorna il prezzo totale da pagare per il carrello della spesa del cliente `c`.
3. Definire una classe `ClienteFedele` derivata da `Cliente` i cui oggetti rappresentano un cliente fedele, ovvero dotato di fidelity card. Ogni cliente fedele è caratterizzato dal saldo dei punti accumulati. D'altra parte la soglia punti `SP` per ottenere la vincita del premio è fissata per tutti i clienti fedeli a 100, mentre lo sconto incondizionato `Sc` è fissato per tutti i clienti fedeli al 5%.
 - (a) Il comportamento di `spesaTotale()` per un cliente fedele è aggiornato nel seguente modo: `cf.spesaTotale()` ritorna il prezzo totale da pagare per il carrello della spesa del cliente `cf` dove ogni prodotto nel carrello della spesa gode dello sconto incondizionato `Sc`.
 - (b) La classe `ClienteFedele` fornisce un metodo `void accreditaPunti(int)` con il seguente comportamento: `cf.accreditaPunti(n)` incrementa di `n` il saldo punti del cliente fedele `cf`; inoltre se dopo l'aggiornamento il saldo punti è maggiore o uguale alla soglia punti `SP` per ottenere il premio, decrementa il saldo punti di `SP` e quindi lancia una eccezione di tipo `RitiroPremio`; si chiede anche di definire tale classe `RitiroPremio`.
4. Definire una classe `GestioneGiornaliera` i cui oggetti permettono la gestione giornaliera della cassa e dei punti fedeltà del supermercato. Un oggetto della classe `GestioneGiornaliera` è caratterizzato dalla lista giornaliera dei clienti del supermercato, rappresentabile tramite un vettore.

11 ESERCIZI RIEPILOGATIVI

- (a) GestioneGiornaliera fornisce un metodo double chiudi Cassa () con il seguente comportamento: g.chiudiCassa () deve ritornare l'incasso giornaliero totale del supermercato, ovvero per la lista giornaliera dei clienti rappresentata da g, ed accreditare a tutti i clienti fedeli della giornata i punti fedeltà accumulati secondo la seguente regola: per un cliente fedele con una spesa totale di s euro vengono accreditati $s/10$ punti (dove $s/10$ denota il troncamento intero della divisione).
- (b) GestioneGiornaliera fornisce un metodo int saldoPuntiGiornaliero () con il seguente comportamento: g.saldoPuntiGiornaliero () ritorna il totale dei saldi punti per tutti i clienti fedeli della lista giornaliera dei clienti rappresentata da g.

Esercizio 11.16. Definire un template di classe $C<T, size>$ con parametro di tipo T e parametro valore size di tipo intero che soddisfi le seguenti specifiche:

1. MultiInfo<T> è un template di classe associato ed annidato nel template $C<T, size>$. Un oggetto di MultiInfo<T> rappresenta un oggetto di tipo T, detto *informazione*, con una certa *molteplicità* $m \geq 0$.
2. Un oggetto di $C<T, size>$ rappresenta un array allocato dinamicamente di dimensione size di oggetti di MultiInfo<T>.
3. $C<T, size>$ rende disponibile un costruttore $C(\text{const } T\&, \text{ int})$ con il seguente comportamento: una invocazione $C(t, k)$ costruisce un oggetto di $C<T, size>$ il cui array contiene in ogni posizione un oggetto di MultiInfo<T> con informazione t e quando $k \geq 1$ con molteplicità k, altrimenti (cioè quando $k < 1$) con molteplicità 0.
4. Nel template $C<T, size>$ il costruttore di copia, l'assegnazione e il distruttore devono essere "profondi", cioè la costruzione di copia e l'assegnazione di copia non devono provocare alcuna condivisione di memoria mentre la distruzione deve provocare anche la deallocazione di tutta la memoria dinamica.
5. $C<T, size>$ fornisce l'overloading dell'operatore $T^* \text{ operator } [] (\text{int})$ con il seguente comportamento: se $0 \leq k < \text{size}$ allora una invocazione $c[k]$ ritorna un puntatore all'informazione di tipo T memorizzata nell'array di c in posizione k, altrimenti ritorna il puntatore nullo.
6. $C<T, size>$ fornisce un metodo int occorrenze (const T&) con il seguente comportamento: una invocazione $c.\text{occorrenze}(t)$ ritorna la somma delle molteplicità di tutte le occorrenze dell'informazione t nell'array memorizzato in c.
7. Deve essere disponibile l'overloading dell'operatore di output per oggetti di $C<T, size>$ che permette di stampare tutte le informazioni di tipo T con relativa molteplicità memorizzate nell'array di un oggetto di $C<T, size>$.

Esercizio 11.17. Si considerino le seguenti dichiarazioni di classi di qualche libreria grafica, dove gli oggetti delle classi Container, Component, Button e MenuItem sono chiamati, rispettivamente, contenitori, componenti, pulsanti ed entrate di menu.

```
class Component;

class Container {
public:
    virtual ~Container();
    vector<Component*> getComponents() const;
};

class Component: public Container {};

class Button: public Component {
public:
    vector<Container*> getContainers() const;
};

class MenuItem: public Button {
public:
    void setEnabled(bool b = true);
};

class NoButton {};
```

Assumiamo i seguenti fatti.

1. Il comportamento del metodo `getComponents()` della classe `Container` è il seguente: `c.getComponents()` ritorna un `vector` di puntatori a tutte le componenti inserite nel contenitore `c`; se `c` non ha alcuna componente allora ritorna un `vector` vuoto.
2. Il comportamento del metodo `getContainers()` della classe `Button` è il seguente: `b.getContainers()` ritorna un `vector` di puntatori a tutti i contenitori che contengono il pulsante `b`; se `b` non appartiene ad alcun contenitore allora ritorna un `vector` vuoto.
3. Il comportamento del metodo `setEnabled()` della classe `MenuItem` è il seguente: `mi.setEnabled(b)` abilita (con `b==true`) o disabilita (con `b==false`) l'entrata di menu `mi`.

Definire una funzione `Button** Fun(const Container&)` con il seguente comportamento: in ogni invocazione `Fun(c)`

1. Se `c` contiene almeno una componente `Button` allora

ritorna un puntatore alla prima cella di un array dinamico di puntatori a pulsanti contenente tutti e soli i puntatori ai pulsanti che sono componenti del contenitore `c` ed in cui tutte le componenti che sono una entrata di menu e sono contenute in almeno 2 contenitori vengono disabilitate.

2. Se invece `c` non contiene nessuna componente `Button` allora solleva una eccezione di tipo `NoButton`.

Esercizio 11.18. Si consideri il seguente modello di realtà concernente i file audio memorizzati in un riproduttore audio digitale `iZod©`.

(A) Definire la seguente gerarchia di classi.

1. Definire una classe base polimorfa astratta `FileAudio` i cui oggetti rappresentano un file audio memorizzabile in un `iZod`. Ogni `FileAudio` è caratterizzato dal titolo (una stringa) e dalla propria dimensione in MB. La classe è astratta in quanto prevede i seguenti **metodi virtuali puri**:

- un metodo di “clonazione”: `FileAudio* clone()`.
- un metodo `bool qualita()` con il seguente contratto: `f->qualita()` ritorna true se il file audio `*f` è considerato di qualità, altrimenti ritorna false.

2. Definire una classe concreta `Mp3` derivata da `FileAudio` i cui oggetti rappresentano un file audio in formato mp3. Ogni oggetto `Mp3` è caratterizzato dal proprio bitrate espresso in Kbit/s. La classe `Mp3` implementa i metodi virtuali puri di `FileAudio` come segue:

- per ogni puntatore `p` a `Mp3`, `p->clone()` ritorna un puntatore ad un oggetto `Mp3` che è una copia di `*p`.
- per ogni puntatore `p` a `Mp3`, `p->qualita()` ritorna true se il bitrate di `*p` è ≥ 192 Kbit/s, altrimenti ritorna false.

3. Definire una classe concreta `WAV` derivata da `FileAudio` i cui oggetti rappresentano un file audio in formato WAV. Ogni oggetto `WAV` è caratterizzato dalla propria frequenza di campionamento espressa in kHz e dall’essere lossless oppure no (cioè con compressione senza perdita oppure con perdita). La classe `WAV` implementa i metodi virtuali puri di `FileAudio` come segue:

- per ogni puntatore `p` a `WAV`, `p->clone()` ritorna un puntatore ad un oggetto `WAV` che è una copia di `*p`.
- per ogni puntatore `p` a `WAV`, `p->qualita()` ritorna true se la frequenza di campionamento di `*p` è ≥ 96 kHz, altrimenti ritorna false.

(B) Definire una classe `iZod` i cui oggetti rappresentano i brani memorizzati in un `iZod`. La classe `iZod` deve soddisfare le seguenti specifiche:

1. È definita una classe annidata Brano i cui oggetti rappresentano un brano memorizzato nell'iZod. Ogni oggetto Brano è rappresentato da un puntatore polimorfo ad un FileAudio.

- La classe Brano deve essere dotata di un opportuno costruttore Brano (FileAudio*) con il seguente comportamento: Brano (p) costruisce un oggetto Brano il cui puntatore polimorfo punta ad una copia dell'oggetto *p.
- La classe Brano ridefinisce costruttore di copia profonda, assegnazione profonda e distruttore profondo.

2. Un oggetto di izod è quindi caratterizzato da un vector di oggetti di tipo Brano che contiene tutti i brani memorizzati nell'iZod.

3. La classe izod rende disponibili i seguenti metodi:

- Un metodo `vector<Mp3> mp3(double, int)` con il seguente comportamento: una invocazione `iz.mp3(dim, br)` ritorna un vector di oggetti Mp3 contenente tutti e soli i file audio in formato mp3 memorizzati nell'iZod iz che: (i) hanno una dimensione $\geq dim$ e (ii) hanno un bitrate $\geq br$.
- Un metodo `vector<FileAudio*> braniQual()` con il seguente comportamento: una invocazione `iz.braniQual()` ritorna il vector dei puntatori ai FileAudio memorizzati nell'iZod iz che: (i) sono considerati di qualità e (ii) se sono dei file audio WAV allora devono essere lossless.
- Un metodo `void insert(Mp3*)` con il seguente comportamento: una invocazione `iz.insert(p)` inserisce il nuovo oggetto Brano (p) nel vector dei brani memorizzati nell'iZod iz se il file audio mp3 *p non è già memorizzato in iz, mentre se il file audio *p risulta già memorizzato non provoca alcun effetto.

11 ESERCIZI RIEPILOGATIVI

Appendice A

Soluzioni degli Esercizi

A.1 Esercizi del Capitolo 11

Soluzione (Possibile) Esercizio 11.1.

```
// file "intmod.h"
#ifndef INTMOD_H
#define INTMOD_H

class IntMod {
public:
    explicit IntMod(int n=0); // impedisce la conversione
                            // implicita int->IntMod
    operator int() const; // conversione implicita IntMod->int
    static void set_modulo(int);
    IntMod operator+(const IntMod&) const;
    IntMod operator*(const IntMod&) const;
private:
    static int modulo;
    int val;
};

#endif
```

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

IntMod::IntMod(int n): val(n%modulo) {}

IntMod::operator int() const { return val; }
```

A SOLUZIONI DEGLI ESERCIZI

```
void IntMod::set_modulo(int mod) { modulo=mod; }

IntMod IntMod::operator+(const IntMod& m) const {
    return IntMod(val + m.val);
}

IntMod IntMod::operator*(const IntMod& m) const {
    return IntMod(val * m.val);
}

int IntMod::modulo=1;
```

```
//ESEMPIO DI UTILIZZO
#include <iostream>
#include "intmod.h"
using std::cout; using std::endl;

int main() {
    IntMod::set_modulo(4);
    IntMod a(3), b(2);
    cout << a+b << endl;           // stampa 1
    cout << a*b << endl;           // stampa 2
    cout << IntMod(4)+a << endl; // stampa 3
    cout << a+4 << endl;           // stampa 7
}
```

Soluzione Esercizio 11.2.

```
0 6 7 8
```

Soluzione Esercizio 11.3.

```
0 1 2 3 UNO
0 1 3 6 DUE
```

Soluzione (Possibile) Esercizio 11.4.

```
class Nodo {
friend class Albero;
private:
    Nodo(string st="***", Nodo* s=0, Nodo* d=0): info(st), sx(s),
                                                    dx(d) {}
    string info;
```

A.1 Esercizi del Capitolo 11

```
Nodo* sx;
Nodo* dx;
};

class Albero {
public:
    Albero(): radice(0) {}
    Albero(const Albero& t) {radice = copia(t.radice);}
private:
    Nodo* radice;
    static Nodo* copia(Nodo* p) {
        if (!p) return 0;
        else return new Nodo(p->info, copia(p->sx), copia(p->dx));
    }
};
```

Soluzione (Possibile) Esercizio 11.5.

```
// file "data.h"
#ifndef DATA_H
#define DATA_H
#include <iostream>
#include <string>
using std::string; using std::ostream;

class Data {
public:
    Data(string = "", int = 1, int = 1, int = 0);

    string get_gsett() const;
    int get_giorno() const;
    int get_mese() const;
    int get_anno() const;

    bool operator==(const Data&) const;
    bool operator<(const Data&) const;
    void aggiungi_anno();

private:
    int giorno, mese, anno;
    string gsett;
};

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

#endif
```

```
// file "data.cpp"
```

A SOLUZIONI DEGLI ESERCIZI

```
#include "data.h"

Data::Data(string s, int g, int m, int a):
    gsett(s), giorno(g), mese(m), anno(a) {}

string Data::get_gsett() const {return gsett;}

int Data::get_giorno() const {return giorno;}

int Data::get_mese() const {return mese;}

int Data::get_anno() const {return anno;}

bool Data::operator==(const Data& d) const {
    return ((gsett==d.gsett) && (giorno==d.giorno) &&
            (mese==d.mese) && (anno==d.anno));
}

bool Data::operator<(const Data& d) const {
    if(anno < d.anno) return true;
    if(anno==d.anno && mese<d.mese) return true;
    if ((anno == d.anno) && (mese==d.mese) && (giorno < d.giorno))
        return true;
    return false;
}

ostream& operator<<(ostream& os, const Data& d) {
    return os << d.get_gsett() << " " << d.get_giorno()
        << "/" << d.get_mese() << "/" << d.get_anno();}
```

```
// ESEMPIO DI UTILIZZO
#include "data.h"
#include <iostream>
using std::cout;

int main() {
    Data d("lun",21,10,2002), e("mar",22,10,2002);
    cout << d << endl;
    cout << (d<e) << " " << (e==d) << endl;
}
```

Soluzione (Possibile) Esercizio 11.6.

```
// file "vettore.h"
#ifndef VETTORE_H
```

A.1 Esercizi del Capitolo 11

```
#define VETTORE_H
#include<iostream>
using std::ostream;

class Vettore {
public:
    Vettore(int =1, int =0);
    Vettore(const Vettore& );
    Vettore& operator=(const Vettore& );
    int& operator[](int) const;
    bool operator==(const Vettore&) const;
    int dim() const { return size; } // definizione inline
private:
    int size; // dimensione del vettore
    int* v; // array dinamico che memorizza il vettore
};

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

```
// file "vettore.cpp"
#include "vettore.h"
using std::cout; using std::endl;

Vettore::Vettore(int sz, int x): size(sz), v(new int[sz]) {
    for(int i=0; i<size; i++) v[i] = x;
}

Vettore::Vettore(const Vettore& x): size(x.size),
                           v(new int[x.size]) {
    for(int i=0; i<size; i++) v[i]=x.v[i];
}

Vettore& Vettore::operator=(const Vettore& x) {
    if(this != &x) { // se siamo nel caso x=x non faccio nulla
        delete[] v; // dealloco
        size = x.size;
        v = new int[x.size];
        for(int i=0; i<size; i++) v[i]=x.v[i];
    }
    return *this;
}

int& Vettore::operator[](int i) const { return v[i]; }
```

A SOLUZIONI DEGLI ESERCIZI

```
bool Vettore::operator==(const Vettore& x) const {
    if(size != x.size) return false;
    for(int i=0; i<size; i++) if(v[i] != x.v[i]) return false;
    return true;
}

ostream& operator<<(ostream& ostr, const Vettore& x) {
    ostr << '(';
    for(int i=0; i<x.dim()-1; i++) {
        ostr << x[i] << ',';
        if((i+1)%10 == 0) cout << endl;
    }
    return ostr << x[x.dim()-1] << ")\n";
}
```

Soluzione (Possibile) Esercizio 11.7.

```
int main() {
    A a; B b; C c; D d; E e;
    cout << F(&b,e) << F(&b,c) << F(&a,c) << F(&a,e);
}
```

Soluzione (Possibile) Esercizio 11.8.

```
int Fun(vector<B*>& v) {
    int tot = 0;
    for(vector<B*>::iterator it = v.begin(); it!= v.end(); ++it)
        if(typeid(*v[0])!=typeid(*(*it)) && dynamic_cast<C*>(*it))
            ++tot;
    return tot;
}
```

Soluzione (Possibile) Esercizio 11.9.

```
class Auto {
private:
    int cavalliFiscali;
    static double tassaPerCF;
protected:
    Auto(int cf = 0): cavalliFiscali(cf) {}
    virtual ~A() {}
public:
    int getCavalliFiscali() const {return cavalliFiscali;}
    static double getTassaPerCF() {return tassaPerCF;}
    virtual double tassa() const = 0;
```

A.1 Esercizi del Capitolo 11

```
};

double Auto::tassaPerCF = 5;

class Diesel: public Auto {
private:
    static double tassaDiesel;
public:
    Diesel(int cf = 0): Auto(cf) {}
    virtual double tassa() const {
        return getCavalliFiscali() * getTassaPerCF() + tassaDiesel;
    }
};
double Diesel::tassaDiesel = 100;

class Benzina: public Auto {
private:
    static double bonusEco4;
    bool eco4;
public:
    Benzina(int cf = 0, bool x = true): Auto(cf), eco4(x) {}
    virtual double tassa() const {
        return (eco4 ? getCavalliFiscali()*getTassaPerCF()-bonusEco4;
                  : getCavalliFiscali()*getTassaPerCF());
    }
};
double Benzina::bonusEco4 = 50;

class ACI {
private:
    vector<Auto*> v;
public:
    double incassaBolli() const {
        double tot = 0;
        for(vector<Auto*>::const_iterator it=v.begin(); it!=v.end();
            ++it)
            tot += (*it)->tassa();
        return tot;
    }
    void aggiungiAuto(const Auto& a) {
        Auto& t = const_cast<Auto&>(a); v.push_back(&t);
    }
};

int main() {
    Benzina fiat(60, false), lancia(60, true);
```

A SOLUZIONI DEGLI ESERCIZI

```
Diesel bmw(90), audi(80);
ACI a;
a.aggiungiAuto(fiat); a.aggiungiAuto(lancia);
a.aggiungiAuto(bmw); a.aggiungiAuto(audi);
cout << a.incassaBolli() << endl; // stampa: 1600
}
```

Soluzione Esercizio 11.10.

- (1) COMPILA.
- (2) NON COMPILA.
- (3) COMPILA.
- (4) NON COMPILA.
- (5) NON COMPILA.
- (6) COMPILA.

Soluzione (Possibile) Esercizio 11.11.

```
class Biglietto {
private:
    string nome;
    bool galleria;
protected:
    Biglietto(const Biglietto& b) :
        nome(b.nome), galleria(b.galleria) {}
    Biglietto(string n, bool g=false): nome(n), galleria(g) {}
    virtual ~Biglietto() {}
public:
    bool isGalleria() const {return galleria;}
    string getNome() const {return nome;}
};

class PostoNumerato: public Biglietto {
private:
    int fila;
public:
    PostoNumerato(string nome, int f): Biglietto(nome), fila(f) {}
    int numFila() const {return fila;}
};

class PostoNonNumerato: public Biglietto {
private:
```

A.1 Esercizi del Capitolo 11

```
bool ridotto;
public:
    PostoNonNumerato(string nome, bool g=true, bool r=false) :
        Biglietto(nome, g), ridotto(r) {}
    bool Ridotto() const {return ridotto;}
};

class Spettacolo {
private:
    double prezzoBase, addizionale;
    int maxNumerati, maxFila;
    int numeratiVenduti;
    list<Biglietto*> l;
public:
    Spettacolo(double pb, double add, int maxN, int maxF, int nv=0) :
        prezzoBase(pb), addizionale(add), maxNumerati(maxN),
        maxFila(maxF), numeratiVenduti(nv) {}

    void aggiungiBiglietto(const Biglietto& b) {
        Biglietto* p = const_cast<Biglietto*>(&b);
        if(dynamic_cast<PostoNumerato*>(p))
            if(numeratiVenduti < maxNumerati)
                {numeratiVenduti++; v.push_back(p);}
            else cout << "Posti numerati esauriti\n";
        else if(dynamic_cast<PostoNonNumerato*>(p)) l.push_back(p);
    }

    double prezzo(const Biglietto& b) const {
        double s = 0;
        Biglietto* p = const_cast<Biglietto*>(&b);
        PostoNumerato* x = dynamic_cast<PostoNumerato*>(p);
        if(x)
            s = ((x->numFila() <= maxFila) ? 2*prezzoBase+2*addizionale
                                              : 2*prezzoBase);
        else {
            PostoNonNumerato* y = dynamic_cast<PostoNonNumerato*>(p);
            if(y) {
                s = prezzoBase;
                if(!y->isGalleria()) s += addizionale;
                if(y->Ridotto()) s -= addizionale/2;
            }
        }
        return s;
    }
    double incasso() const {
```

A SOLUZIONI DEGLI ESERCIZI

```
double tot=0;
for(list<Biglietto*>::const_iterator it = l.begin();
                                         it!= l.end(); ++it)
    tot += prezzo(**it);
return tot;
};

int main() {
    Spettacolo s(10,6,200,10);
    PostoNumerato a1("pippo",2);           // prezzo: 32
    PostoNumerato a2("pluto",17);          // prezzo: 20
    PostoNonNumerato b1("zagor");          // prezzo: 10
    PostoNonNumerato b2("pluto",false,true); // prezzo: 13
    s.aggiungiBiglietto(a1); s.aggiungiBiglietto(a2);
    s.aggiungiBiglietto(b1); s.aggiungiBiglietto(b2);
    cout << s.incasso();                  // stampa: 75
}
```

Soluzione Esercizio 11.12.

```
8 2 9 UNO
8 2 2 7 DUE
1 3 1 7 TRE
1 3 1 7 QUATTRO
```

Soluzione (Possibile) Esercizio 11.13.

```
class C {
private:
    vector<ios*> v;
    int max;
public:
    C(int k=10): max(k) {}
    void insert(ios& s) {
        if( v.size() < max && typeid(s)!=typeid(stringstream)
            && typeid(s)!=typeid(fstream) )
            v.push_back(&s);
    }
    template<class T>
    int conta(T& t) const {
        int x=0;
        for(vector<ios*>::const_iterator it=v.begin(); it!=v.end();
                                         ++it) {
            if(dynamic_cast<T*>(*it)) ++x;
        }
    }
};
```

A.1 Esercizi del Capitolo 11

```
    }
    return x;
}
};
```

Soluzione Esercizio 11.14.

```
Z() A() Z() B() Z() A() Z() C() **0
Z() A() Z() Bc **1
Z() Ac Zc **2
Z() A() Z() B() Z() C() D() **3
Z() A() Z() B() Zc Dc **4
```

Soluzione (Possibile) Esercizio 11.15.

```
class RitiraPremio {};

class Prodotto {
private:
    double prezzo;
public:
    Prodotto(double p): prezzo(p) {}
    double getPrezzo() const {return prezzo;}
    double setPrezzo(double p) {prezzo=p;}
};

class Cliente {
private:
    vector<Prodotto> carrello;
public:
    virtual ~Cliente() {}
    virtual double spesaTotale() const {
        double tot=0;
        for(int i=0; i<carrello.size(); i++)
            tot += carrello[i].getPrezzo();
        return tot;
    }
};

class ClienteFedele : public Cliente {
private:
    int punti; // saldo punti
    static int sogliaPuntiPremio; // punti richiesti per premio
    static int sconto; // sconto in percentuale
public:
```

A SOLUZIONI DEGLI ESERCIZI

```
ClienteFedele(int p=0): punti(p) {}
virtual double spesaTotale() const {
    double tot=Cliente::spesaTotale();
    tot *= 1-sconto/100;
    return tot;
}
void accreditaPunti(int n) throw(RitiraPremio) {
    punti +=n;
    if(punti>=sogliaPuntiPremio){
        punti -= sogliaPuntiPremio;
        throw RitiraPremio();
    }
}
int saldoPunti() const {return punti;}
};

int ClienteFedele::sogliaPuntiPremio=100;
int ClienteFedele::sogliaPuntiPremio=5;

class GestioneGiornaliera {
private:
    vector<Cliente*> v;
public:
    double chiudiCassa() const {
        double tot=0;
        for(int i=0; i<v.size(); ++i) {
            double s = v[i]->spesaTotale();
            tot += s;
            ClienteFedele* p = dynamic_cast<ClienteFedele*>(v[i]);
            try{p->accreditaPunti(static_cast<int>(s/10));}
            catch(RitiraPremio) {}
        }
        return tot;
    }
    int saldoPuntiGiornaliero() const {
        int tot=0;
        for(int i=0; i<v.size(); ++i) {
            ClienteFedele *p=dynamic_cast<ClienteFedele*>(v[i]);
            if(p) tot+= p->saldoPunti();
        }
        return tot;
    }
};
```

Soluzione (Possibile) Esercizio 11.16.

```
// dichiarazione incompleta
```

A.1 Esercizi del Capitolo 11

```
template <class T, int size> class C;

// dichiarazione incompleta
template <class T, int size>
ostream& operator<< (ostream&, const C<T,size>&);

template <class T=string, int size=1>
class C {
    friend ostream& operator<< <T,size>(ostream&, const C<T,size>&);
public:
    C(const T& t=T(), int k=1): array(new MultiInfo[size](t,k)) {}
    C(const C& x): array(new MultiInfo[size]) {
        for(int i=0; i<size; ++i) array[i] = x.array[i];
    }
    ~C() {delete[] array;}
    C& operator=(const C& x) {
        if (this != &x) {
            // delete[] array; // non serve!
            for(int i=0; i<size; ++i) array[i] = x.array[i];
        }
        return *this;
    }
    T* operator[](int k) const {
        return (0<=k && k<size)? &(array[k].info) : 0;
    }
    int occorrenze(const T& t) const {
        int sum=0;
        for(int i=0; i<size; ++i)
            if(array[i].info==t) sum+=array[i].mult;
        return sum;
    }
private:
    class MultiInfo {
public:
    MultiInfo(const T& x = T(), int z = 1) : info(x), mult(z) {
        if (mult<1) mult=1;
    }
    T info;
    int mult;
    };
    MultiInfo* array;
};

template <class T, int size>
ostream& operator<< (ostream& os, const C<T,size>& x) {
```

A SOLUZIONI DEGLI ESERCIZI

```
for(int i=0; i<size; ++i) os << " Valore: " <<  
    x.array[i].info << " Molteplicita': " << x.array[i].mult;  
return os;  
}
```

Soluzione (Possibile) Esercizio 11.17.

```
class NoButton {};  
  
Button** Fun(const Container& c) throw(NoButton) {  
    vector<Component*> v = c.getComponents();  
    int cont=0;  
    for(int i=0;i<v.size();++i)  
        if(dynamic_cast<Button*>(v[i])) ++cont;  
    if(cont==0) throw NoButton();  
    Button** a = new Button*[cont];  
    cont=0;  
    for(int i=0; i<v.size(); ++i) {  
        if(dynamic_cast<Button*>(v[i])) {  
            MenuItem* mi = dynamic_cast<MenuItem*>(v[i]);  
            if(mi&&mi->getContainers().size()>1) mi->setEnabled(false);  
            a[cont]=dynamic_cast<Button*>(v[i]);  
            ++cont;  
        }  
    }  
    return a;  
}
```

Soluzione (Possibile) Esercizio 11.18.

```
class FileAudio {  
public:  
    virtual FileAudio* clone() const =0;  
    virtual bool qualita() const =0;  
    double dimensione() const {return dim;}  
private:  
    double dim; // MB  
};  
  
class Mp3: public FileAudio {  
public:  
    virtual FileAudio* clone() const {  
        return new Mp3(*this);  
    }  
    virtual bool qualita() const {
```

A.1 Esercizi del Capitolo 11

```
    return BitRate >= 192;
}
bool operator==(const Mp3& x) const {
    return dimensione() == x.dimensione() &&
        bitrate() == x.bitrate();
}
int bitrate() const {return BitRate;}
private:
    int BitRate; // Kbits
};

class WAV: public FileAudio {
public:
    virtual FileAudio* clone() const {
        return new WAV(*this);
    }
    virtual bool qualita() const {
        return frequenzaCampionamento >= 96;
    }
    bool LossLess() const {return lossless;}
    double freq() const {return frequenzaCampionamento;}
private:
    bool lossless;
    double frequenzaCampionamento; // kHz
};

class iZod {
private:
    class Brano {
public:
    FileAudio* f;

    Brano(FileAudio* p): f(p->clone()) {}
    Brano(const Brano& x): f((x.f)->clone()) {}
    Brano& operator=(const Brano& x) {
        if(this != &x) {
            delete f;
            f=(x.f)->clone();
        }
        return *this;
    }
    ~Brano() {delete f;}
    };
};
```

A SOLUZIONI DEGLI ESERCIZI

```
vector<Brano> brani;

public:
    vector<Mp3> mp3(double dim, int br) const {
        vector<Mp3> ris;
        vector<brano>::const_iterator it = brani.begin();
        Mp3* p=0;
        for( ; it<brani.end(); ++it) {
            if( (*it).f->dimensione() >= dim) {
                p=dynamic_cast<Mp3*> ((*it).f);
                if(p && p->bitrate() > br)
                    ris.push_back(*p);
            }
        }
    }

    vector<FileAudio*> braniQual() const {
        vector<FileAudio*> ris;
        vector<Brano>::const_iterator it = brani.begin();
        for( ; it<brani.end(); ++it) {
            if( (*it).f->qualita() ) {
                WAV* q = dynamic_cast<WAV*> ((*it).f);
                if(q)
                    { if(q->LossLess()) ris.push_back(q); }
                else ris.push_back((*it).f);
            }
        }
    }

    void insert(Mp3* q) {
        vector<Brano>::iterator it = brani.begin();
        bool trovato = false;
        for( ; it<brani.end() && !trovato; ++it) {
            Mp3* p = dynamic_cast<Mp3*> ((*it).f);
            if(p && *p == *q)
                trovato = true;
        }
        if(!trovato) brani.push_back(brano(q));
    }
};
```

A.2 Altri esercizi del testo

Soluzione Esercizio 2.5.1.

A.2 Altri esercizi del testo

```
int main() {
    C x, y(1), z(2); const C v(2);
    z=x.F(y);           // (1): OK
    //v.F(y);           // (2): Illegale
    v.G(y);             // (3): OK
    (v.G(y)).F(x);    // (4): OK
    (v.G(y)).G(x);    // (5): OK
    //x.H(v);           // (6): Illegale
    //x.H(z.G(y));     // (7): Illegale (nota bene!)
    x.I(z.G(y));       // (8): OK (nota bene!)
    x.J(z.G(y));       // (9): OK
    v.J(z.G(y));       // (10): OK
}
```

Soluzione (Possibile) Esercizio 2.7.5.

```
#include <math.h>
#include <iostream>
using std::cout;

class Point {
private:
    double x_, y_, z_;
public:
    Point();
    Point(double, double, double);
    void negate();
    double norm() const;
    void print() const;
};

Point::Point() {x_=0; y_=0; z_=0;}

Point::Point(double x, double y, double z) {x_=x; y_=y; z_=z;}

void Point::negate() {x_ *= -1; y_ *= -1; z_ *= -1;}

double Point::norm() const {return sqrt(x_*x_ + y_*y_ + z_*z_);}

void Point::print() const {cout << '(' << x_ << ', ' << y_ << ', ' << z_ << ')' ;}
```

Soluzione (Possibile) Esercizio 2.10.2.

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

A SOLUZIONI DEGLI ESERCIZI

Soluzione Esercizio 2.10.3.

```
D01 Dc C0 UNO  
D01 C1 DUE  
Dc Cc TRE
```

Soluzione Esercizio 3.6.1.

```
PROGRAMMA UNO  
5Cd 5Cd uno  
5Cd 5Cd due  
5Cd 5Cd 5Cd 4Cd 3Cd
```

```
PROGRAMMA DUE  
5Cd uno  
5Cd due  
5Cd 5Cd 5Cd 4Cd 3Cd
```

```
PROGRAMMA TRE  
uno  
due  
5Cd 5Cd 5Cd 4Cd 3Cd
```

```
PROGRAMMA QUATTRO  
Bd Ad Cd Ad Cd Ad Bd Ad
```

Soluzione Esercizio 4.3.4.

```
0A() 2A()
```

Si noti che non stampa: 0A() 0A() 2A() !

Soluzione (Possibile) Esercizio 5.5.1.

```
template <class T>  
T Extract_Min(vector<T> &v) {  
    //NOTA: passaggio per riferimento non costante  
    if (v.empty()) {cerr << "Coda vuota!"; exit(1)};  
    for (vector<T>::iterator i = v.begin() + 1; i != v.end(); i++)  
        if (*i > *(i-1)) swap<T> (i, i-1);  
    T m = *(v.end() - 1);  
    v.pop_back();  
    return m;  
}
```

A.2 Altri esercizi del testo

```
template <class T>
void swap(vector<T>::iterator x, vector<T>::iterator y) {
    T t = *x; *x = *y; *y = t;
}
```

Soluzione Esercizio 6.2.2.

```
// PROGRAMMA UNO
1
12

// PROGRAMMA DUE
3.14 *
3.14 3.14 4
* 6.28 5
* 6.28 5
```

Soluzione Esercizio 6.3.2.

```
C1 C1 D0 E0 C0 D1 C1 C1 D0 E0 F0 C0 G0
```