NoiStudiamo

DISPENSA DI LABORATORIO

Programmazione II

Programmi completi in C++: classi, gerarchie, template, liste, pile, code, alberi e grafi.

A CURA DI

Damiano La Pusata

Ottobre 2026

INDICE

Indice dei capitoli

I dieci capitoli della dispensa, dagli strumenti di lavoro agli esercizi completi.

1. Gli strumenti di lavoro 3

2. Progettare una classe 6

3. Gerarchie e polimorfismo 9

4. Operatori e template 12

5. Liste concatenate 16

6. Pile e code 19

7. Alberi binari di ricerca 23

8. Grafi 29

9. Due esercizi completi 32

10. Metodo di lavoro ed errori frequenti 38

CAPITOLO 1

Gli strumenti di lavoro

Prima di scrivere programmi più grandi conviene padroneggiare gli strumenti: il compilatore g++ usato da riga di comando, la struttura di un programma C++, l'input e l'output su console e su file, la formattazione dei numeri. Il capitolo si chiude con un primo programma completo.

Come leggere i listati di questa dispensa

Ogni capitolo è costruito attorno a uno o più **programmi completi**: si possono copiare, compilare ed eseguire così come sono, e sotto ciascuno è riportato l'output effettivamente prodotto. Tutti i programmi sono stati compilati con g++ attivando gli avvisi e verificati con uno strumento che controlla la memoria, in modo da non contenere né errori né perdite di memoria.

DA RICORDARE

Il segno meno nei listati

Per ragioni tipografiche, nei listati e negli output il trattino è stampato come segno meno (−). Quando si ricopia il codice va digitato il normale trattino della tastiera: è il carattere che il compilatore si aspetta nell'operatore freccia `p−>succ`, nel decremento `i−−`, nei numeri negativi e nelle opzioni del compilatore come `−Wall`. Un segno meno copiato e incollato dal PDF produce un errore di compilazione del tipo «stray in program».

Compilare da riga di comando

Un ambiente di sviluppo nasconde i passaggi della compilazione dietro un pulsante. Lavorare dal terminale è più istruttivo e funziona su qualunque macchina: basta un editor di testo e il compilatore **g++**. Il comando fondamentale prende uno o più file sorgente e produce un file eseguibile.

TERMINALE

```
$ g++ −Wall programma.cpp −o programma    # compila: crea l'eseguibile "programma"
$ ./programma                              # lo esegue
$ ./programma < numeri.txt                 # lo esegue leggendo l'input da un file
$ g++ −Wall a.cpp b.cpp main.cpp −o prog   # più file sorgente, un solo eseguibile
```

L'opzione `−o` sceglie il nome dell'eseguibile; senza di essa il file si chiama `a.out`. L'opzione `−Wall` attiva gli **avvisi** (warning): il programma viene compilato lo stesso, ma g++ segnala costrutti sospetti come una variabile usata senza essere inizializzata, un confronto tra interi con e senza segno, una funzione che non restituisce un valore. Gli avvisi vanno sempre letti, perché molto spesso indicano un errore logico. Il simbolo < del terminale collega lo standard input a un file: il programma legge con `cin` come se i dati fossero digitati da tastiera, ed è il modo più comodo per provare più volte lo stesso input.

Gli errori di compilazione si leggono **dal primo**: un solo errore, per esempio un punto e virgola mancante dopo la definizione di una classe, ne provoca spesso decine di altri a cascata. Ogni messaggio indica il file, la riga e la colonna in cui il compilatore si è accorto del problema, che a volte si trova nella riga precedente.

La struttura di un programma

Un programma C++ è formato dalle direttive `#include` che importano le librerie, dalle definizioni di classi e funzioni e dalla funzione `main`, da cui parte l'esecuzione. Le classi e le funzioni devono essere **dichiarate prima di essere usate**: per questo, in un file unico, l'ordine naturale è prima le classi di base, poi le derivate, poi le strutture dati, infine il `main`.

HEADER	CHE COSA FORNISCE	ESEMPI
`<iostream>`	input e output su console	`cin`, `cout`, `cerr`, `endl`
`<fstream>`	input e output su file	`ifstream`, `ofstream`
`<string>`	la classe string	`getline`, `size()`, `stoi`, `stod`
`<sstream>`	flussi su stringhe	`stringstream` per spezzare una riga
`<iomanip>`	formattazione dell'output	`setw`, `setprecision`, `fixed`

Leggere i dati

Gli oggetti `cin` (tastiera) e `ifstream` (file) si usano allo stesso modo, con l'operatore di estrazione `>>`, che salta gli spazi e legge un valore del tipo della variabile. L'espressione `in >> x` vale **vero** se la lettura è riuscita e **falso** se è fallita, perché il file è finito o perché il dato non era del tipo atteso. Il modo corretto di leggere fino alla fine è quindi mettere la lettura **nella condizione del ciclo**.

LETTURA FINO ALLA FINE DEL FILE

```
while (in >> nome >> voto) {    // corretto: entra solo se la lettura è riuscita
    ...
}

while (!in.eof()) {             // sbagliato: eof diventa vero solo DOPO una lettura fallita
    in >> nome >> voto;         // l'ultimo record viene elaborato due volte
    ...
}
```

Quando i campi di una riga sono separati da un carattere diverso dallo spazio, per esempio una virgola, si legge l'intera riga con `getline` e la si spezza con uno `stringstream` e `getline(ss, campo, ',')`; le funzioni `stoi` e `stod` convertono poi il testo in numero. L'esempio completo si trova nel capitolo 7.

Un primo programma completo

Il programma seguente legge da un file nomi e voti, li stampa incolonnati, calcola la media e trova il voto più alto. Contiene già quasi tutti gli strumenti del capitolo: apertura del file con controllo, ciclo di lettura corretto, conversione esplicita per evitare la divisione intera, formattazione dei decimali.

C1_VOTI.CPP

```
// Legge da file nome e voto di ogni studente e stampa la media della classe
#include <iostream>
#include <fstream>
#include <iomanip>     // fixed, setprecision
#include <string>
using namespace std;

int main() {
```

```
    ifstream in("voti.txt");                // apre il file in lettura
    if (!in) {                              // il file non esiste o non si apre
        cerr << "Impossibile aprire voti.txt" << endl;
        return 1;
    }
    string nome;
    int voto;
    int quanti = 0, somma = 0, massimo = 0;
    string migliore;
    while (in >> nome >> voto) {            // si ferma quando una lettura fallisce
        cout << left << setw(10) << nome << voto << endl;
        somma += voto;
        quanti++;
        if (voto > massimo) { massimo = voto; migliore = nome; }
    }
    in.close();
    if (quanti == 0) { cout << "Nessun voto letto" << endl; return 0; }
    double media = (double) somma / quanti;   // cast: altrimenti divisione intera
    cout << fixed << setprecision(2);
    cout << "Studenti: " << quanti << ", media: " << media << endl;
    cout << "Voto piu' alto: " << massimo << " (" << migliore << ")" << endl;
    return 0;
}
```

VOTI.TXT

```
Anna 28
Bruno 24
Carla 30
Dario 27
```

OUTPUT

```
Anna      28
Bruno     24
Carla     30
Dario     27
Studenti: 4, media: 27.25
Voto piu' alto: 30 (Carla)
```

Alcuni dettagli meritano attenzione. Il test `if (!in)` intercetta il file mancante: senza di esso il programma andrebbe avanti senza leggere nulla. L'espressione `(double) somma / quanti` converte la somma prima della divisione; scrivendo `somma / quanti` tra due interi il risultato sarebbe 27 e non 27.25. I manipolatori `fixed` e `setprecision(2)` fissano due cifre dopo la virgola per tutte le stampe successive, mentre `left` e `setw(10)` allineano i nomi in una colonna di dieci caratteri. Infine il programma controlla il caso del file vuoto, in cui la media non sarebbe definita.

ESERCIZIO

Da provare

Modificare il programma perché scriva su un file `esito.txt` (con un `ofstream`) il nome di ogni studente seguito da «sufficiente» se il voto è almeno 18, «insufficiente» altrimenti. Il file di output si apre come quello di input, `ofstream out("esito.txt");`, e si scrive con `out << ...`.

CAPITOLO 2

Progettare una classe

Scrivere una classe significa decidere quali dati nascondere, quali operazioni offrire e quali regole far rispettare. Il capitolo segue la progettazione di una classe completa, un conto corrente, e fissa uno schema da riusare per qualunque classe.

Lo schema di base

Quasi tutte le classi di un esercizio hanno la stessa struttura: **attributi privati**, un **costruttore** che li inizializza con la lista di inizializzazione, i metodi **get** per leggerli, eventualmente i metodi **set** o le operazioni che li modificano rispettando dei vincoli, un metodo **stampa**. I metodi che non modificano l'oggetto si dichiarano `const`: è un'informazione utile per chi legge, ed è indispensabile per poterli chiamare su oggetti costanti o ricevuti per riferimento costante.

DEFINIZIONE

Invariante di classe

Un **invariante** è una proprietà che ogni oggetto della classe deve rispettare in qualunque momento, per esempio «il saldo non è mai negativo». Il costruttore la stabilisce e ogni metodo pubblico la conserva. Rendere privati gli attributi serve proprio a questo: nessuno, dall'esterno, può violare l'invariante assegnando direttamente un valore sbagliato.

Il conto corrente

Si vuole una classe `ContoCorrente` con intestatario e saldo. Ogni conto riceve un numero progressivo al momento della creazione; si può depositare, prelevare se il saldo è sufficiente, fare un bonifico verso un altro conto; si vuole sapere quanti conti sono stati aperti.

C2_CONTO.CPP

```
// Classe ContoCorrente: incapsulamento, costruttori, metodi const, membro statico
#include <iostream>
#include <string>
using namespace std;
class ContoCorrente {
private:
    string intestatario;
    double saldo;
    int numero;                    // numero progressivo assegnato alla creazione
    static int contatore;          // condiviso da tutti i conti
public:
    // costruttore con lista di inizializzazione e parametro di default
    ContoCorrente(string nome, double iniziale = 0)
        : intestatario(nome), saldo(iniziale), numero(++contatore) {}
    string getIntestatario() const { return intestatario; }
    double getSaldo() const { return saldo; }
    int getNumero() const { return numero; }
    static int quantiConti() { return contatore; }
    void deposita(double importo) {
        if (importo > 0) saldo += importo;          // ignora importi non validi
    }
```

```
    // restituisce false se il saldo non basta: il chiamante decide cosa fare
    bool preleva(double importo) {
        if (importo <= 0 || importo > saldo) return false;
        saldo -= importo;
        return true;
    }
    // bonifico verso un altro conto, passato per riferimento
    bool bonifico(ContoCorrente& destinatario, double importo) {
        if (!preleva(importo)) return false;
        destinatario.deposita(importo);
        return true;
    }
    void stampa() const {
        cout << "Conto " << numero << " di " << intestatario
             << ": saldo " << saldo << " euro" << endl;
    }
};

int ContoCorrente::contatore = 0;   // definizione del membro statico
int main() {
    ContoCorrente a("Anna", 500), b("Bruno");
    a.deposita(250);
    if (!b.preleva(100)) cout << "Prelievo rifiutato a " << b.getIntestatario() << endl;
    a.bonifico(b, 300);
    a.stampa();
    b.stampa();
    cout << "Conti aperti: " << ContoCorrente::quantiConti() << endl;
    return 0;
}
```

OUTPUT

```
Prelievo rifiutato a Bruno
Conto 1 di Anna: saldo 450 euro
Conto 2 di Bruno: saldo 300 euro
Conti aperti: 2
```

LE SCELTE DEL PROGETTO

Il costruttore usa un **parametro di default**: `ContoCorrente b("Bruno")` apre un conto con saldo zero. La lista di inizializzazione assegna anche il numero, incrementando prima il contatore: il primo conto riceve 1, il secondo 2.

Il contatore è un attributo `static`: ne esiste una sola copia, condivisa da tutti gli oggetti. Va **definito** una volta fuori dalla classe (`int ContoCorrente::contatore = 0;`), altrimenti il collegamento fallisce con un errore di riferimento non definito. Il metodo `quantiConti` è anch'esso statico e si chiama con il nome della classe, senza bisogno di un oggetto.

Il prelievo non stampa messaggi di errore: restituisce `false` e lascia al chiamante la decisione su cosa fare. È una buona abitudine, perché la stessa classe può essere usata in programmi diversi, con messaggi diversi o senza messaggi. Il bonifico riusa `preleva` e `deposita` invece di modificare direttamente i saldi, così i controlli restano in un unico punto.

Il bonifico riceve il conto destinatario **per riferimento**. Se lo ricevesse per valore lavorerebbe su una copia: la copia riceverebbe i soldi e verrebbe distrutta alla fine del metodo, mentre il conto originale resterebbe invariato. Lo stesso metodo può accedere agli attributi privati di `destinatario`, perché la protezione in C++ vale per classe e non per oggetto.

Quando servono il distruttore e il costruttore di copia

La classe `ContoCorrente` non alloca memoria con `new`: gli attributi sono una `string` e due numeri, che si distruggono e si copiano da soli. Il distruttore e il costruttore di copia generati dal compilatore vanno quindi bene. La situazione cambia quando un attributo è un **puntatore a memoria allocata dalla classe**: in quel caso serve il distruttore che la libera e, se gli oggetti vengono copiati, un costruttore di copia e un operatore di assegnamento che duplicano la memoria invece di condividerla. È la **regola dei tre**, che si incontra di nuovo nel capitolo 4.

ATTRIBUTI DELLA CLASSE	DISTRUTTORE	COPIA
solo tipi semplici, `string`, oggetti	non serve	quella automatica va bene
puntatore a memoria allocata con `new`	serve: `delete` o `delete[]`	da scrivere o da vietare
puntatore a un oggetto di cui non si è proprietari	non deve liberarlo	quella automatica va bene

ESERCIZIO

Da provare

Aggiungere un metodo `applicaInteressi(double tasso)` che aumenta il saldo della percentuale indicata, e un metodo statico che restituisce il conto con saldo maggiore tra due conti ricevuti per riferimento costante. Verificare che il compilatore rifiuti una chiamata a `deposita` su un conto dichiarato `const`.

CAPITOLO 3

Gerarchie e polimorfismo

Molti problemi coinvolgono oggetti simili ma non identici: dipendenti pagati in modi diversi, elementi di una collezione di tipo diverso. Il capitolo mostra come organizzarli in una gerarchia e come trattarli in modo uniforme attraverso puntatori alla classe base.

Il problema

Un'azienda ha **impiegati**, con uno stipendio mensile fisso, e **consulenti**, pagati in base alle ore lavorate. Tutti hanno nome e matricola. Si vuole stampare l'elenco del personale con lo stipendio di ciascuno, calcolare il totale degli stipendi e il numero complessivo di ore di consulenza.

Gli attributi comuni vanno in una classe base `Dipendente`; ogni tipo di dipendente è una classe derivata che aggiunge i propri attributi. Il calcolo dello stipendio esiste per tutti ma è diverso per ciascuno: è il caso tipico di una **funzione virtuale**. Poiché non esiste un dipendente «generico», la funzione è **virtuale pura** e la classe base è **astratta**.

C3_PERSONALE.CPP

```
// Gerarchia Dipendente -> Impiegato, Consulente: polimorfismo e array di puntatori
#include <iostream>
#include <string>
using namespace std;
class Dipendente {                                  // classe astratta
protected:
    string nome;
    int matricola;
public:
    Dipendente(string n, int m) : nome(n), matricola(m) {}
    virtual ~Dipendente() {}                        // virtuale: si cancella tramite base
    virtual double stipendio() const = 0;           // virtuale pura
    virtual void stampa() const {
        cout << matricola << " " << nome << ", stipendio " << stipendio();
    }
};

class Impiegato : public Dipendente {
private:
    double mensile;
public:
    Impiegato(string n, int m, double s) : Dipendente(n, m), mensile(s) {}
    double stipendio() const override { return mensile; }
    void stampa() const override {
        Dipendente::stampa();                       // riusa la versione della base
        cout << " (impiegato)" << endl;
    }
};

class Consulente : public Dipendente {
private:
    int ore;
    double pagaOraria;
public:
    Consulente(string n, int m, int h, double p)
```

```
        : Dipendente(n, m), ore(h), pagaOraria(p) {}
    double stipendio() const override { return ore * pagaOraria; }
    int getOre() const { return ore; }
    void stampa() const override {
        Dipendente::stampa();
        cout << " (consulente, " << ore << " ore)" << endl;
    }
};
int main() {
    const int N = 4;
    Dipendente* staff[N] = {                        // puntatori alla classe base
        new Impiegato("Rossi", 101, 1800),
        new Consulente("Verdi", 102, 60, 35),
        new Impiegato("Bianchi", 103, 2100),
        new Consulente("Neri", 104, 20, 50)
    };
    double totale = 0;
    int oreConsulenze = 0;
    for (int i = 0; i < N; i++) {
        staff[i]->stampa();                         // binding dinamico
        totale += staff[i]->stipendio();
        // dynamic_cast restituisce nullptr se l'oggetto non è un Consulente
        Consulente* c = dynamic_cast<Consulente*>(staff[i]);
        if (c != nullptr) oreConsulenze += c->getOre();
    }
    cout << "Totale stipendi: " << totale << endl;
    cout << "Ore di consulenza: " << oreConsulenze << endl;
    for (int i = 0; i < N; i++) delete staff[i];    // chiama il distruttore giusto
    return 0;
}
```

OUTPUT

```
101 Rossi, stipendio 1800 (impiegato)
102 Verdi, stipendio 2100 (consulente, 60 ore)
103 Bianchi, stipendio 2100 (impiegato)
104 Neri, stipendio 1000 (consulente, 20 ore)
Totale stipendi: 7000
Ore di consulenza: 80
```

Che cosa rende il codice polimorfo

Il `main` non conosce i tipi effettivi: lavora con un array di `Dipendente*`. Quando chiama `staff[i]->stipendio()` viene eseguita la versione della classe a cui appartiene davvero l'oggetto puntato, perché il metodo è `virtual` (binding dinamico). Aggiungere un nuovo tipo di dipendente, per esempio uno stagista con un rimborso fisso, richiede una nuova classe derivata, ma **nessuna modifica** al ciclo del `main`.

1. **La parola chiave** `override` dice al compilatore che il metodo intende ridefinirne uno virtuale della base. Se la firma non coincide (manca un `const`, cambia un parametro) si ottiene un errore invece di un nuovo metodo che nessuno chiamerà mai.
2. **La stampa della derivata riusa quella della base** con `Dipendente::stampa()` e poi aggiunge la propria parte. L'operatore di risoluzione dell'ambito evita la chiamata virtuale, che altrimenti richiamerebbe la stessa funzione all'infinito.
3. **Il distruttore della base è virtuale.** Gli oggetti vengono distrutti con `delete staff[i]`, cioè tramite un puntatore alla base: senza `virtual` verrebbe eseguito solo il distruttore di `Dipendente`.

4. **Gli attributi comuni sono** `protected`, quindi accessibili alle derivate ma non dall'esterno. In alternativa possono restare privati, e le derivate useranno i metodi get della base.

QUANDO SERVE IL TIPO EFFETTIVO

Le ore di consulenza esistono solo per i consulenti: non avrebbe senso aggiungere un metodo `getOre` alla classe base. Il `dynamic_cast<Consulente*>` prova a convertire il puntatore e restituisce `nullptr` se l'oggetto non è un consulente; funziona solo su classi con almeno una funzione virtuale. È uno strumento da usare con misura: se il `main` si riempie di `dynamic_cast`, probabilmente manca una funzione virtuale nella base.

DA RICORDARE

Perché un array di puntatori e non di oggetti

Un array `Dipendente staff[4]` non si può nemmeno dichiarare, perché la classe è astratta. Ma anche con una base concreta, copiare un oggetto derivato in una variabile del tipo base ne taglia la parte derivata (*slicing*): resta un oggetto della base, e il polimorfismo si perde. Il polimorfismo in C++ funziona solo attraverso **puntatori** o **riferimenti**.

ESERCIZIO

Da provare

Aggiungere la classe `Stagista`, con un rimborso mensile fisso, e inserirne uno nell'array. Poi scrivere una funzione `Dipendente* piuPagato(Dipendente* v[], int n)` che restituisce il dipendente con lo stipendio maggiore, e stamparlo.

CAPITOLO 4

Operatori e template

L'overloading degli operatori permette di usare gli oggetti con la stessa sintassi dei tipi predefiniti; i template permettono di scrivere una funzione o una classe una sola volta per qualunque tipo. Il capitolo li mette all'opera su una classe di frazioni e su un array dinamico generico.

Una classe Frazione

Una frazione è una coppia numeratore e denominatore, sempre ridotta ai minimi termini e con il segno al numeratore. Le si vuole sommare e moltiplicare con + e *, confrontare con < e ==, stampare con `cout <<`.

C4_FRAZIONE.CPP

```
// Classe Frazione con overloading degli operatori e funzione template
#include <iostream>
using namespace std;
int mcd(int a, int b) {                     // algoritmo di Euclide
    while (b != 0) { int r = a % b; a = b; b = r; }
    return a < 0 ? -a : a;
}
class Frazione {
private:
    int num, den;
    void semplifica() {
        int d = mcd(num, den);
        num /= d; den /= d;
        if (den < 0) { num = -num; den = -den; }   // segno sempre al numeratore
    }
public:
    Frazione(int n = 0, int d = 1) : num(n), den(d) { semplifica(); }
    // operatori membro: l'operando sinistro è *this
    Frazione operator+(const Frazione& f) const {
        return Frazione(num * f.den + f.num * den, den * f.den);
    }
    Frazione operator*(const Frazione& f) const {
        return Frazione(num * f.num, den * f.den);
    }
    bool operator<(const Frazione& f) const { return num * f.den < f.num * den; }
    bool operator==(const Frazione& f) const { return num == f.num && den == f.den; }
    // operator<< non può essere membro: a sinistra c'è un ostream
    friend ostream& operator<<(ostream& os, const Frazione& f) {
        os << f.num;
        if (f.den != 1) os << "/" << f.den;
        return os;
    }
};
// funziona con qualunque tipo che abbia l'operatore <
template <typename T>
T massimo(const T v[], int n) {
    T m = v[0];
```

```
    for (int i = 1; i < n; i++)
        if (m < v[i]) m = v[i];
    return m;
}
int main() {
    Frazione a(1, 2), b(3, 4), c(6, 8);
    cout << a << " + " << b << " = " << a + b << endl;
    cout << a << " * " << b << " = " << a * b << endl;
    cout << "3/4 == 6/8? " << (b == c ? "si" : "no") << endl;
    Frazione v[] = {Frazione(2, 3), Frazione(5, 4), Frazione(-1, 2), Frazione(7, 6)};
    int interi[] = {4, 9, 2, 7};
    cout << "massimo frazioni: " << massimo(v, 4) << endl;
    cout << "massimo interi: " << massimo(interi, 4) << endl;
    return 0;
}
```

OUTPUT

```
1/2 + 3/4 = 5/4
1/2 * 3/4 = 3/8
3/4 == 6/8? si
massimo frazioni: 5/4
massimo interi: 9
```

Gli operatori aritmetici e di confronto sono **metodi**: in `a + b` l'operando sinistro è l'oggetto su cui si chiama il metodo, il destro è il parametro. Sono `const` perché non modificano né `a` né `b`, e restituiscono una **nuova** frazione per valore. Il costruttore chiama `semplifica`, quindi ogni frazione creata, anche come risultato di un'operazione, è già ridotta: per questo == può limitarsi a confrontare numeratori e denominatori.

L'operatore << non può essere un metodo di `Frazione`, perché il suo operando sinistro è un `ostream`. Si scrive come funzione esterna, dichiarata `friend` per accedere agli attributi privati. Riceve lo stream per riferimento e lo **restituisce**: è questo che permette di concatenare più stampe nella stessa istruzione.

IL TEMPLATE DI FUNZIONE

La funzione `massimo` è scritta una volta sola e funziona con gli interi e con le frazioni. Il compilatore genera una versione per ogni tipo con cui viene chiamata, deducendo `T` dagli argomenti. L'unico requisito sul tipo è implicito: deve esistere l'operatore <. Chiamandola con una classe che non lo definisce si ottiene un errore di compilazione all'interno del template.

Un array dinamico generico

Gli array del C++ hanno dimensione fissa. Un **array dinamico** alloca un blocco di memoria e, quando è pieno, ne alloca uno di **dimensione doppia**, copia gli elementi e libera il vecchio. Scritto come template di classe, funziona per qualunque tipo.

C4_VETTORE.CPP

```
// Template di classe: array dinamico che raddoppia quando è pieno
#include <iostream>
#include <string>
using namespace std;
template <typename T>
class Vettore {
private:
```

```
    T* dati;
    int n;          // elementi presenti
    int cap;        // capacità dell'array allocato
    void raddoppia() {
        cap *= 2;
        T* nuovo = new T[cap];
        for (int i = 0; i < n; i++) nuovo[i] = dati[i];
        delete[] dati;                          // libera il vecchio array
        dati = nuovo;
    }
public:
    Vettore(int c = 2) : dati(new T[c]), n(0), cap(c) {}
    ~Vettore() { delete[] dati; }
    Vettore(const Vettore&) = delete;           // copia vietata: vedi testo
    Vettore& operator=(const Vettore&) = delete;
    void aggiungi(const T& x) {
        if (n == cap) raddoppia();              // O(n) solo ogni tanto
        dati[n++] = x;
    }
    int size() const { return n; }
    int capacita() const { return cap; }
    T& operator[](int i) { return dati[i]; }    // accesso come un array
};
int main() {
    Vettore<int> v;
    for (int i = 1; i <= 5; i++) {
        v.aggiungi(i * i);
        cout << "dopo " << i << " inserimenti: capacita' " << v.capacita() << endl;
    }
    for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
    cout << endl;
    Vettore<string> parole;
    parole.aggiungi("pila"); parole.aggiungi("coda"); parole.aggiungi("lista");
    parole[0] = "albero";                       // operator[] restituisce T&
    for (int i = 0; i < parole.size(); i++) cout << parole[i] << " ";
    cout << endl;
    return 0;
}
```

OUTPUT

```
dopo 1 inserimenti: capacita' 2
dopo 2 inserimenti: capacita' 2
dopo 3 inserimenti: capacita' 4
dopo 4 inserimenti: capacita' 4
dopo 5 inserimenti: capacita' 8
1 4 9 16 25
albero coda lista
```

Il raddoppio costa O(n), ma avviene sempre più di rado: partendo da capacità 2, le copie avvengono dopo 2, 4, 8, 16 inserimenti, e il costo totale di n inserimenti resta proporzionale a n. In media, quindi, ogni inserimento costa O(1). Aumentare la capacità di una quantità fissa, per esempio di 10 celle alla volta, porterebbe invece a un costo totale quadratico.

L'operatore `[]` restituisce un **riferimento** `T&`: per questo `parole[0] = "albero"` modifica l'elemento dentro il vettore. Se restituisse `T` per valore, l'assegnamento cambierebbe una copia temporanea.

DA RICORDARE

La copia è vietata

La classe possiede un array allocato con `new`: il distruttore lo libera. Se un `Vettore` venisse copiato con il costruttore di copia automatico, le due copie avrebbero lo stesso puntatore `dati` e alla loro distruzione lo stesso array verrebbe liberato due volte. Le due righe con `= delete` **vietano** la copia e l'assegnamento: un tentativo di copia diventa un errore di compilazione invece di un errore durante l'esecuzione. L'alternativa è scrivere un costruttore di copia che allochi un nuovo array e copi gli elementi.

ESERCIZIO

Da provare

Aggiungere a `Vettore` un metodo `rimuovi(int i)` che elimina l'elemento in posizione i spostando indietro i successivi, e un operatore << template che stampa il vettore tra parentesi quadre.

CAPITOLO 5

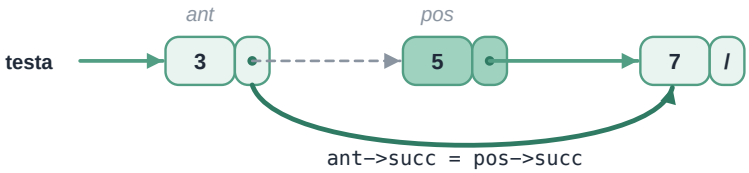
Liste concatenate

Una lista concatenata si costruisce un nodo alla volta e si percorre seguendo i puntatori. Il capitolo realizza una lista ordinata di interi completa di inserimento, rimozione, ricerca, funzioni ricorsive e distruttore, con un metodo per scrivere codice sui puntatori senza errori.

Il metodo del disegno

Le operazioni sulle liste sono brevi ma piene di casi particolari. Prima di scrivere una funzione conviene **disegnare** la lista con tre o quattro nodi, eseguire l'operazione a mano spostando le frecce, e solo dopo tradurre ogni freccia spostata in un assegnamento. Poi si ripete il disegno per i casi limite, che sono quasi sempre gli stessi.

- **Lista vuota**: `testa` vale `nullptr`; ogni accesso a `testa->...` è un errore.
- **Operazione sul primo nodo**: cambia `testa`, quindi va trattata a parte.
- **Operazione sull'ultimo nodo**: il successivo è `nullptr`, e il ciclo deve fermarsi in tempo.
- **Elemento assente**: il ciclo arriva alla fine senza trovarlo, e la funzione deve accorgersene.



Rimozione di un nodo: il precedente viene collegato al successivo, poi il nodo viene liberato.

Una lista ordinata

La lista mantiene i valori in ordine crescente. Ogni inserimento cerca la posizione giusta; la ricerca e la rimozione possono fermarsi appena superano il valore cercato, perché oltre ci sono solo valori più grandi.

C5_LISTA.CPP

```
// Lista concatenata ordinata di interi
#include <iostream>
using namespace std;
struct Nodo {
    int valore;
    Nodo* succ;
    Nodo(int v, Nodo* s = nullptr) : valore(v), succ(s) {}
};

class ListaOrdinata {
private:
    Nodo* testa;
    int n;
    int sommaRic(Nodo* p) const {                    // ricorsiva: caso base lista vuota
        if (p == nullptr) return 0;
        return p->valore + sommaRic(p->succ);
    }
    void stampaRic(Nodo* p) const {                  // stampa al ritorno
        if (p == nullptr) return;
```

```
        stampaRic(p->succ);
        cout << p->valore << " ";
    }
public:
    ListaOrdinata() : testa(nullptr), n(0) {}
    ~ListaOrdinata() {                                  // libera tutti i nodi
        while (testa != nullptr) {
            Nodo* tmp = testa;
            testa = testa->succ;
            delete tmp;
        }
    }
    // inserisce mantenendo l'ordine crescente: O(n)
    void inserisci(int x) {
        if (testa == nullptr || x <= testa->valore) {   // in testa
            testa = new Nodo(x, testa);
        } else {
            Nodo* p = testa;                            // p: ultimo nodo con valore < x
            while (p->succ != nullptr && p->succ->valore < x) p = p->succ;
            p->succ = new Nodo(x, p->succ);
        }
        n++;
    }
    // rimuove la prima occorrenza di x; restituisce false se non c'è
    bool rimuovi(int x) {
        Nodo* prec = nullptr;
        Nodo* p = testa;
        while (p != nullptr && p->valore < x) { prec = p; p = p->succ; }
        if (p == nullptr || p->valore != x) return false;
        if (prec == nullptr) testa = p->succ;           // era la testa
        else prec->succ = p->succ;                      // scavalca p
        delete p;
        n--;
        return true;
    }
    bool cerca(int x) const {                           // si ferma appena supera x
        for (Nodo* p = testa; p != nullptr && p->valore <= x; p = p->succ)
            if (p->valore == x) return true;
        return false;
    }
    void stampaDecrescente() const {                    // ricorsione: prima il resto
        stampaRic(testa);
        cout << endl;
    }
    int size() const { return n; }
    int somma() const { return sommaRic(testa); }
    void stampa() const {
        cout << "[ ";
        for (Nodo* p = testa; p != nullptr; p = p->succ) cout << p->valore << " ";
        cout << "]" << endl;
    }
};
int main() {
    ListaOrdinata l;
    int dati[] = {7, 3, 9, 1, 5, 3};
    for (int x : dati) l.inserisci(x);
    l.stampa();
    cout << "elementi: " << l.size() << ", somma: " << l.somma() << endl;
    cout << "cerca 5: " << l.cerca(5) << ", cerca 6: " << l.cerca(6) << endl;
```

```
    l.rimuovi(1); l.rimuovi(9); l.rimuovi(4);
    l.stampa();
    l.stampaDecrescente();
    return 0;
}
```

OUTPUT

```
[ 1 3 3 5 7 9 ]
elementi: 6, somma: 28
cerca 5: 1, cerca 6: 0
[ 3 3 5 7 ]
7 5 3 3
```

INSERIMENTO

Se la lista è vuota o il nuovo valore non supera quello della testa, il nodo diventa la nuova testa: `new Nodo(x, testa)` crea il nodo già collegato alla vecchia testa. Altrimenti il puntatore `p` avanza finché il nodo **successivo** ha un valore minore di `x`: ci si ferma sul nodo **dopo il quale** inserire, perché in una lista semplice non si può tornare indietro. La condizione `p->succ != nullptr` viene controllata per prima: grazie alla valutazione in cortocircuito di `&&`, `p->succ->valore` non viene mai letto su un puntatore nullo.

RIMOZIONE

Per scavalcare un nodo serve il suo **precedente**. Il ciclo avanza quindi due puntatori, `prec` e `p`, sempre a un passo di distanza. Alla fine ci sono tre situazioni: `p` è nullo o contiene un valore diverso (elemento assente), `prec` è nullo (si rimuove la testa), oppure il caso generale. Il nodo va liberato con `delete` **dopo** averlo staccato dalla lista. Lo stesso ordine vale nel distruttore, che salva la testa in un puntatore temporaneo, avanza la testa e solo dopo libera il nodo salvato: con `delete testa; testa = testa->succ;` si leggerebbe un nodo già liberato.

RICORSIONE SULLE LISTE

Una lista è una struttura ricorsiva: è vuota, oppure è un nodo seguito da una lista. Molte funzioni si scrivono quindi in modo naturale con la ricorsione, con caso base `p == nullptr`. La somma restituisce il valore del nodo più la somma del resto. La stampa decrescente sfrutta l'ordine delle chiamate: prima si stampa ricorsivamente il resto della lista, poi il nodo corrente, quindi i valori escono dal fondo verso la testa. Spostando la `cout` prima della chiamata ricorsiva si otterrebbe l'ordine normale.

Le funzioni ricorsive sono **private** e ricevono un `Nodo*`; quelle pubbliche, senza parametri, le avviano dalla testa. Così chi usa la classe non vede i nodi, che restano un dettaglio interno.

ESERCIZIO

Da provare

Aggiungere un metodo `rimuoviDuplicati()` che lascia una sola copia di ogni valore (nella lista ordinata i duplicati sono consecutivi) e un metodo ricorsivo `int conta(int x)` che restituisce il numero di occorrenze di `x`.

CAPITOLO 6

Pile e code

Pile e code si realizzano in due modi: su una lista concatenata o su un array. Il capitolo ne mostra una per tipo, entrambe come template, e le usa per due problemi classici: il controllo delle parentesi e una coda con priorità costruita usando solo le operazioni di coda.

Una pila per controllare le parentesi

Un'espressione ha le parentesi corrette se ogni parentesi chiusa corrisponde all'ultima aperta e non ancora chiusa, e se alla fine non resta nulla di aperto. «L'ultima aperta» è proprio l'elemento in cima a una **pila**: si scorre il testo, si impila ogni parentesi aperta e, a ogni chiusa, si estrae la cima e si controlla che sia del tipo giusto.

C6_PARENTESI.CPP

```
// Pila template su lista concatenata, usata per controllare le parentesi
#include <iostream>
#include <string>
using namespace std;
template <typename T>
class Pila {
private:
    struct Elem {
        T dato;
        Elem* sotto;                                // l'elemento sottostante
    };
    Elem* cima;
public:
    Pila() : cima(nullptr) {}
    ~Pila() { while (!isEmpty()) pop(); }
    bool isEmpty() const { return cima == nullptr; }
    void push(const T& x) { cima = new Elem{x, cima}; }   // O(1)
    T pop() {                                       // precondizione: pila non vuota
        Elem* e = cima;
        T x = e->dato;
        cima = e->sotto;
        delete e;
        return x;
    }
    T top() const { return cima->dato; }
};
// restituisce la posizione del primo errore, oppure -1 se le parentesi sono corrette
int controlla(const string& s) {
    Pila<int> aperte;                               // posizioni delle parentesi aperte
    for (int i = 0; i < (int) s.size(); i++) {
        char c = s[i];
        if (c == '(' || c == '[' || c == '{') aperte.push(i);
        else if (c == ')' || c == ']' || c == '}') {
            if (aperte.isEmpty()) return i;         // chiusa senza aperta
            char a = s[aperte.pop()];
            if ((c == ')' && a != '(') || (c == ']' && a != '[') ||
                (c == '}' && a != '{'))
                return i;                           // tipo sbagliato
```

```
        }
    }
    if (!aperte.isEmpty()) return aperte.top();     // aperta mai chiusa
    return -1;
}

int main() {
    string prove[] = {"a[i] = (b + c) * {d}", "f(x[2)]", "((a + b)", "x + y)"};
    for (const string& s : prove) {
        int e = controlla(s);
        if (e == -1) cout << s << "  ->  corretta" << endl;
        else cout << s << "  ->  errore in posizione " << e << endl;
    }
    return 0;
}
```

OUTPUT

```
a[i] = (b + c) * {d}  ->  corretta
f(x[2)]  ->  errore in posizione 5
((a + b)  ->  errore in posizione 0
x + y)  ->  errore in posizione 5
```

La pila è realizzata su una lista: `push` crea un nodo in testa e `pop` lo toglie, entrambe in tempo O(1). Il nodo `Elem` è una struttura **interna** alla classe, invisibile dall'esterno. La pila contiene le **posizioni** delle parentesi invece dei caratteri: con la posizione si risale al carattere (`s[aperte.pop()]`) e si può anche segnalare dove si trova l'errore. I tre possibili errori sono una chiusa con la pila vuota, una chiusa di tipo diverso dall'ultima aperta e una aperta che resta nella pila alla fine.

DA RICORDARE

Precondizioni

`pop` e `top` non controllano che la pila sia vuota: è una **precondizione**, cioè una condizione che il chiamante deve garantire. Il programma la rispetta controllando `isEmpty()` prima di ogni estrazione. È una scelta comune nelle strutture dati didattiche; in alternativa il metodo può restituire un valore di errore o un `bool`.

Una coda su array circolare

Una coda su array deve inserire dal fondo ed estrarre dalla testa. Se la testa restasse sempre in posizione 0, ogni estrazione richiederebbe di spostare tutti gli elementi. L'**array circolare** evita il problema: la testa avanza, e quando un indice raggiunge la fine dell'array riparte da 0 grazie all'operatore resto `%`. Bastano l'indice della testa e il numero di elementi n: la prima cella libera è `(testa + n) % cap`.

n = 3: gli elementi in ordine di arrivo sono A, B, C; il prossimo enqueue scrive nella cella 1

Una coda circolare di capacità 6 con tre elementi: il fondo ha già fatto il giro dell'array.

Il problema da risolvere è la sala d'attesa di un ambulatorio. I pazienti urgenti devono passare davanti a quelli normali, ma dopo gli urgenti arrivati prima di loro; i normali vanno in fondo. La difficoltà è che si possono usare **soltanto** `enqueue` e `dequeue`.

C6_PAZIENTI.CPP

```
// Coda template su array circolare e inserimento con priorità
#include <iostream>
#include <string>
using namespace std;
template <typename T>
class Coda {
private:
    T* dati;
    int cap, testa, n;
public:
    Coda(int c) : dati(new T[c]), cap(c), testa(0), n(0) {}
    ~Coda() { delete[] dati; }
    bool isEmpty() const { return n == 0; }
    bool isFull() const { return n == cap; }
    int size() const { return n; }
    bool enqueue(const T& x) {                      // O(1)
        if (isFull()) return false;
        dati[(testa + n) % cap] = x;                // il fondo si calcola
        n++;
        return true;
    }
    T dequeue() {                                   // O(1), precondizione: non vuota
        T x = dati[testa];
        testa = (testa + 1) % cap;                  // la testa avanza in cerchio
        n--;
        return x;
    }
};

struct Paziente {
    string nome;
    bool urgente;
};

// inserisce p dopo gli urgenti già presenti, usando solo enqueue e dequeue
template <typename T>
void enqueuePriorita(Coda<T>& c, const T& p) {
    if (!p.urgente) { c.enqueue(p); return; }
    int quanti = c.size();                          // salvato prima del giro
    int urgenti = 0;
    for (int i = 0; i < quanti; i++) {              // conta gli urgenti in testa
        T x = c.dequeue();
        if (x.urgente && urgenti == i) urgenti++;
        c.enqueue(x);
    }
    for (int i = 0; i < urgenti; i++) c.enqueue(c.dequeue());     // urgenti in fondo
    c.enqueue(p);                                                 // poi il nuovo
    for (int i = 0; i < quanti - urgenti; i++) c.enqueue(c.dequeue()); // poi gli altri
}

int main() {
    Coda<Paziente> sala(10);
    enqueuePriorita(sala, Paziente{"Anna", false});
    enqueuePriorita(sala, Paziente{"Bruno", false});
    enqueuePriorita(sala, Paziente{"Carla", true});
```

```
    enqueuePriorita(sala, Paziente{"Dario", false});
    enqueuePriorita(sala, Paziente{"Elena", true});
    cout << "Ordine di visita:";
    while (!sala.isEmpty()) {
        Paziente p = sala.dequeue();
        cout << " " << p.nome << (p.urgente ? "(U)" : "");
    }
    cout << endl;
    return 0;
}
```

OUTPUT

```
Ordine di visita: Carla(U) Elena(U) Anna Bruno Dario
```

LA ROTAZIONE

L'idea è far **ruotare** la coda: un `dequeue` seguito da un `enqueue` dello stesso elemento lo sposta dalla testa al fondo, e ripetendo l'operazione tante volte quanti sono gli elementi la coda torna com'era. Il primo giro conta gli urgenti, che grazie a questo stesso algoritmo stanno sempre all'inizio. Il secondo giro porta in fondo gli urgenti, poi si inserisce il nuovo paziente, infine il terzo giro porta in fondo i normali: il risultato è urgenti vecchi, nuovo urgente, normali. L'operazione costa O(n).

DA RICORDARE

Salvare la dimensione prima del ciclo

Il numero di giri va memorizzato **prima** di cominciare (`int quanti = c.size();`). Durante la rotazione la dimensione resta la stessa, ma in varianti in cui alcuni elementi vengono scartati e non reinseriti cambierebbe a ogni passo, e un ciclo scritto con `i < c.size()` salterebbe degli elementi.

ESERCIZIO

Da provare

Usando solo `push`, `pop` e `isEmpty`, scrivere una funzione template che inverte il contenuto di una coda con l'aiuto di una pila, e una che rimuove da una coda tutti i pazienti non urgenti conservando l'ordine degli altri.

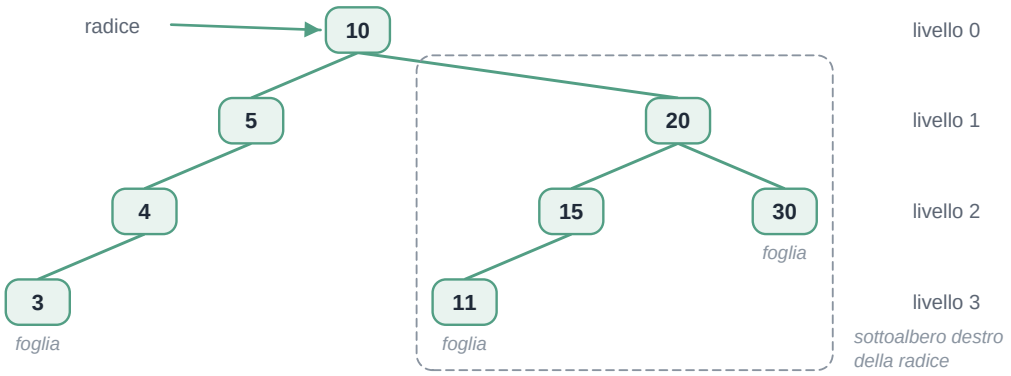
CAPITOLO 7

Alberi binari di ricerca

Un albero binario di ricerca si realizza con poche funzioni ricorsive, quasi tutte lunghe una manciata di righe. Il capitolo costruisce un BST di interi completo di visite e cancellazione, poi un BST di oggetti caricato da file.

Il BST di interi

Il programma legge interi da standard input finché non incontra il valore −1, li inserisce in un BST e stampa le tre visite, il minimo, il massimo, l'altezza e il numero di foglie; poi cerca una chiave e rimuove la radice. L'input di prova è la sequenza 10 5 20 4 3 15 30 11, che produce l'albero della figura.



L'albero ottenuto inserendo 10 5 20 4 3 15 30 11.

C7_BST.CPP

```
// BST di interi: letti fino a −1, tre visite, minimo, massimo, ricerca, cancellazione
#include <iostream>
using namespace std;
struct Nodo {
    int chiave;
    Nodo* sx;
    Nodo* dx;
    Nodo(int k) : chiave(k), sx(nullptr), dx(nullptr) {}   // figli sempre inizializzati
};

class BST {
private:
    Nodo* radice;
    Nodo* inserisci(Nodo* r, int k) {              // restituisce la radice aggiornata
        if (r == nullptr) return new Nodo(k);
        if (k < r->chiave) r->sx = inserisci(r->sx, k);
        else r->dx = inserisci(r->dx, k);          // i duplicati vanno a destra
        return r;
    }
    void preorder(Nodo* r) const {
        if (r == nullptr) return;
        cout << r->chiave << " ";                  // R S D
        preorder(r->sx); preorder(r->dx);
```

```
    }
    void inorder(Nodo* r) const {
        if (r == nullptr) return;
        inorder(r->sx);                          // S R D
        cout << r->chiave << " ";
        inorder(r->dx);
    }
    void postorder(Nodo* r) const {
        if (r == nullptr) return;
        postorder(r->sx); postorder(r->dx);      // S D R
        cout << r->chiave << " ";
    }
    int altezza(Nodo* r) const {                 // albero vuoto: -1, una foglia: 0
        if (r == nullptr) return -1;
        int a = altezza(r->sx), b = altezza(r->dx);
        return 1 + (a > b ? a : b);
    }
    int foglie(Nodo* r) const {
        if (r == nullptr) return 0;
        if (r->sx == nullptr && r->dx == nullptr) return 1;
        return foglie(r->sx) + foglie(r->dx);
    }
    Nodo* rimuovi(Nodo* r, int k) {
        if (r == nullptr) return nullptr;        // chiave assente
        if (k < r->chiave) r->sx = rimuovi(r->sx, k);
        else if (k > r->chiave) r->dx = rimuovi(r->dx, k);
        else if (r->sx == nullptr || r->dx == nullptr) {   // casi 1 e 2
            Nodo* figlio = (r->sx != nullptr) ? r->sx : r->dx;
            delete r;
            return figlio;                       // il figlio prende il suo posto
        } else {                                 // caso 3: due figli
            Nodo* s = r->dx;
            while (s->sx != nullptr) s = s->sx;  // successore: minimo del destro
            r->chiave = s->chiave;               // copia la chiave del successore
            r->dx = rimuovi(r->dx, s->chiave);   // e lo elimina dal destro
        }
        return r;
    }
    void distruggi(Nodo* r) {                    // postorder: prima i figli
        if (r == nullptr) return;
        distruggi(r->sx); distruggi(r->dx);
        delete r;
    }
public:
    BST() : radice(nullptr) {}
    ~BST() { distruggi(radice); }
    void inserisci(int k) { radice = inserisci(radice, k); }
    void rimuovi(int k) { radice = rimuovi(radice, k); }
    bool vuoto() const { return radice == nullptr; }
    bool cerca(int k) const {                    // iterativa: O(h)
        Nodo* p = radice;
        while (p != nullptr && p->chiave != k) p = (k < p->chiave) ? p->sx : p->dx;
        return p != nullptr;
    }
    int minimo() const {                         // precondizione: albero non vuoto
        Nodo* p = radice;
        while (p->sx != nullptr) p = p->sx;
        return p->chiave;
    }
    int massimo() const {
```

```
        Nodo* p = radice;
        while (p->dx != nullptr) p = p->dx;
        return p->chiave;
    }
    void visite() const {
        cout << "preorder:  "; preorder(radice);  cout << endl;
        cout << "inorder:   "; inorder(radice);   cout << endl;
        cout << "postorder: "; postorder(radice); cout << endl;
    }
    int altezza() const { return altezza(radice); }
    int foglie() const { return foglie(radice); }
};
int main() {
    BST t;
    int x;
    while (cin >> x && x != -1) t.inserisci(x);    // -1 termina l'input
    if (t.vuoto()) { cout << "albero vuoto" << endl; return 0; }
    t.visite();
    cout << "minimo " << t.minimo() << ", massimo " << t.massimo()
         << ", altezza " << t.altezza() << ", foglie " << t.foglie() << endl;
    cout << "cerca 15: " << (t.cerca(15) ? "trovato" : "assente") << endl;
    t.rimuovi(10);                                  // la radice, che ha due figli
    cout << "dopo aver rimosso 10:" << endl;
    t.visite();
    return 0;
}
```

NUMERI.TXT

```
10 5 20 4 3 15 30 11 -1
```

OUTPUT DI ./C7_BST < NUMERI.TXT

```
preorder:  10 5 4 3 20 15 11 30
inorder:   3 4 5 10 11 15 20 30
postorder: 3 4 5 11 15 30 20 10
minimo 3, massimo 30, altezza 3, foglie 3
cerca 15: trovato
dopo aver rimosso 10:
preorder:  11 5 4 3 20 15 30
inorder:   3 4 5 11 15 20 30
postorder: 3 4 5 15 30 20 11
```

LO SCHEMA RICORSIVO

Quasi tutte le funzioni private hanno la stessa forma: **caso base** sull'albero vuoto (`r == nullptr`), poi una combinazione dei risultati sui due sottoalberi. Le tre visite differiscono solo per la posizione della stampa rispetto alle due chiamate ricorsive: prima (preorder), in mezzo (inorder), dopo (postorder). L'inorder di un BST produce sempre le chiavi in ordine crescente, ed è il controllo più rapido che l'albero sia stato costruito bene.

L'inserimento ricorsivo **restituisce la radice del sottoalbero aggiornato**, e il chiamante la riassegna: `r->sx = inserisci(r->sx, k)`. Quando la ricorsione arriva a un sottoalbero vuoto, il nuovo nodo restituito viene così agganciato al padre senza bisogno di tenerne traccia. La stessa tecnica rende semplice la cancellazione.

LA CANCELLAZIONE

Trovato il nodo, ci sono tre casi. Se è una **foglia** o ha **un solo figlio**, il nodo viene liberato e al suo posto sale il figlio (eventualmente `nullptr`): i due casi si gestiscono con lo stesso codice. Se ha **due figli**, si cerca il **successore**, cioè il minimo del sottoalbero destro, se ne copia la chiave nel nodo e lo si elimina dal sottoalbero destro, dove ha al più un figlio. Nell'esempio, rimuovendo 10 il successore 11 sale alla radice.

COSTI

Ricerca, inserimento, minimo, massimo e cancellazione percorrono un solo cammino dalla radice: costano O(h), dove h è l'altezza. Le visite, il conteggio delle foglie, l'altezza e il distruttore toccano tutti i nodi: costano O(n). Il distruttore lavora in **postorder**, perché un nodo può essere liberato solo dopo i suoi figli.

Un BST di oggetti letti da file

In molte applicazioni le chiavi sono un campo di un oggetto. Il programma seguente carica degli studenti da un file con i campi separati da virgole, li organizza in un BST con chiave la matricola, li stampa in ordine, cerca due matricole e conta gli studenti con media almeno 27.

C7_STUDENTI.CPP

```
// BST di studenti ordinato per matricola, letto da un file con campi separati da virgole
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
using namespace std;

class Studente {
private:
    int matricola;
    string nome;
    double media;
public:
    Studente(int m, string n, double md) : matricola(m), nome(n), media(md) {}
    int getMatricola() const { return matricola; }
    double getMedia() const { return media; }
    void stampa() const { cout << matricola << " " << nome << " " << media << endl; }
};

class Archivio {                                    // BST con chiave la matricola
private:
    struct Nodo {
        Studente s;
        Nodo* sx;
        Nodo* dx;
        Nodo(const Studente& x) : s(x), sx(nullptr), dx(nullptr) {}
    };
    Nodo* radice;
    void inorder(Nodo* r) const {
        if (r == nullptr) return;
        inorder(r->sx); r->s.stampa(); inorder(r->dx);
    }
    int contaMedia(Nodo* r, double soglia) const {  // visita completa: O(n)
        if (r == nullptr) return 0;
        int qui = (r->s.getMedia() >= soglia) ? 1 : 0;
        return qui + contaMedia(r->sx, soglia) + contaMedia(r->dx, soglia);
```

```
    }
    Nodo* inserisci(Nodo* r, const Studente& x) {   // come per gli interi
        if (r == nullptr) return new Nodo(x);
        if (x.getMatricola() < r->s.getMatricola()) r->sx = inserisci(r->sx, x);
        else r->dx = inserisci(r->dx, x);
        return r;
    }
    void distruggi(Nodo* r) {
        if (r == nullptr) return;
        distruggi(r->sx); distruggi(r->dx); delete r;
    }
public:
    Archivio() : radice(nullptr) {}
    ~Archivio() { distruggi(radice); }
    void inserisci(const Studente& x) { radice = inserisci(radice, x); }
    const Studente* cerca(int m) const {            // nullptr se assente
        Nodo* p = radice;
        while (p != nullptr && p->s.getMatricola() != m)
            p = (m < p->s.getMatricola()) ? p->sx : p->dx;
        return p ? &p->s : nullptr;
    }
    void stampaOrdinato() const { inorder(radice); }
    int contaMedia(double soglia) const { return contaMedia(radice, soglia); }
};
int main() {
    ifstream in("studenti.txt");
    if (!in) { cerr << "file non trovato" << endl; return 1; }
    Archivio a;
    string riga, campo;
    while (getline(in, riga)) {                     // una riga per studente
        stringstream ss(riga);
        getline(ss, campo, ','); int m = stoi(campo);
        string nome; getline(ss, nome, ',');
        getline(ss, campo, ','); double media = stod(campo);
        a.inserisci(Studente(m, nome, media));
    }
    a.stampaOrdinato();
    int cercate[] = {1042, 1100};
    for (int m : cercate) {
        const Studente* s = a.cerca(m);
        if (s) { cout << "trovato: "; s->stampa(); }
        else cout << "matricola " << m << " assente" << endl;
    }
    cout << "studenti con media almeno 27: " << a.contaMedia(27) << endl;
    return 0;
}
```

STUDENTI.TXT

```
1042,Rossi,26.5
1017,Verdi,28.0
1080,Bianchi,24.5
1003,Neri,29.5
1066,Gallo,27.0
```

OUTPUT

```
1003 Neri 29.5
1017 Verdi 28
1042 Rossi 26.5
1066 Gallo 27
1080 Bianchi 24.5
trovato: 1042 Rossi 26.5
matricola 1100 assente
studenti con media almeno 27: 3
```

Il nodo contiene una **copia** dello studente; per crearla si usa il costruttore di copia automatico di `Studente`, sufficiente perché la classe non possiede memoria dinamica. La ricerca restituisce un puntatore all'oggetto trovato, oppure `nullptr`: il chiamante deve **controllarlo** prima di usarlo. Il conteggio con la soglia non può sfruttare l'ordine dell'albero, perché l'ordine riguarda la matricola e non la media: deve visitare tutti i nodi.

ESERCIZIO

Da provare

Aggiungere al BST di interi un metodo che stampa le chiavi comprese tra due valori a e b, visitando solo i sottoalberi che possono contenerle, e un metodo che stampa i nodi livello per livello usando la coda del capitolo 6.

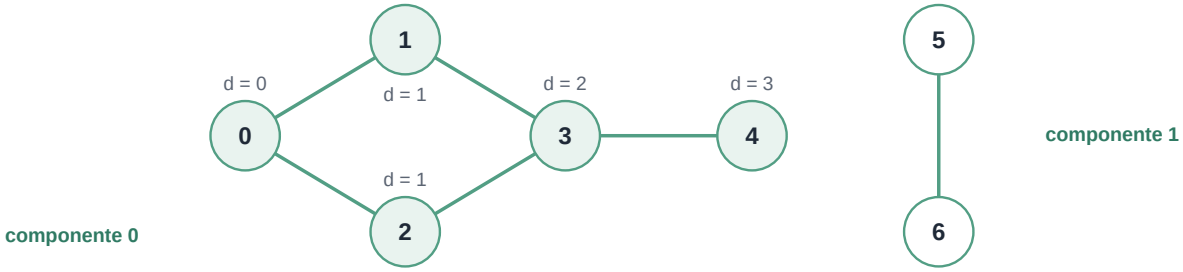
CAPITOLO 8

Grafi

Un grafo si memorizza con un array di liste di adiacenza e si esplora con due visite: in ampiezza, che calcola le distanze, e in profondità, che permette di trovare le componenti connesse. Il capitolo le realizza in un unico programma.

Rappresentazione e visite

Il grafo è **non orientato**: ogni arco {u, v} viene inserito due volte, nella lista di u e in quella di v. L'array `adj` contiene, per ogni nodo, il puntatore alla testa della sua lista di vicini; gli archi si aggiungono in testa, in tempo O(1). Lo spazio occupato è O(V + E).



Il grafo d'esempio, con le distanze dal nodo 0 calcolate dalla BFS.

C8_GRAFO.CPP

```
// Grafo non orientato con liste di adiacenza: BFS con distanze, DFS, componenti connesse
#include <iostream>
using namespace std;
class Grafo {
private:
    struct Arco { int dest; Arco* succ; };
    int n;
    Arco** adj;                                     // adj[u]: lista dei vicini di u
    void aggiungiInTesta(int u, int v) { adj[u] = new Arco{v, adj[u]}; }
    void dfsVisita(int u, bool* visitato) const {
        visitato[u] = true;
        cout << u << " ";
        for (Arco* a = adj[u]; a != nullptr; a = a->succ)
            if (!visitato[a->dest]) dfsVisita(a->dest, visitato);
    }
public:
    Grafo(int nodi) : n(nodi), adj(new Arco*[nodi]) {
        for (int i = 0; i < n; i++) adj[i] = nullptr;
    }
    ~Grafo() {
        for (int i = 0; i < n; i++)
            while (adj[i] != nullptr) { Arco* a = adj[i]; adj[i] = a->succ; delete a; }
        delete[] adj;
    }
    void aggiungiArco(int u, int v) {               // non orientato: due inserimenti
        aggiungiInTesta(u, v);
        aggiungiInTesta(v, u);
```

```
    }
    void bfs(int s) const {                         // O(V + E)
        int* dist = new int[n];
        int* coda = new int[n];                     // ogni nodo entra al più una volta
        int testa = 0, fondo = 0;
        for (int i = 0; i < n; i++) dist[i] = -1;   // -1: non ancora scoperto
        dist[s] = 0;
        coda[fondo++] = s;
        while (testa < fondo) {
            int u = coda[testa++];
            for (Arco* a = adj[u]; a != nullptr; a = a->succ)
                if (dist[a->dest] == -1) {
                    dist[a->dest] = dist[u] + 1;
                    coda[fondo++] = a->dest;
                }
        }
        for (int i = 0; i < n; i++) {
            cout << "distanza " << s << " -> " << i << ": ";
            if (dist[i] == -1) cout << "irraggiungibile" << endl;
            else cout << dist[i] << endl;
        }
        delete[] dist; delete[] coda;
    }
    int componenti() const {                        // una DFS per ogni nodo non visitato
        bool* visitato = new bool[n];
        for (int i = 0; i < n; i++) visitato[i] = false;
        int c = 0;
        for (int u = 0; u < n; u++)
            if (!visitato[u]) {
                cout << "componente " << c << ": ";
                dfsVisita(u, visitato);
                cout << endl;
                c++;
            }
        delete[] visitato;
        return c;
    }
};

int main() {
    Grafo g(7);
    int archi[][2] = {{0, 1}, {0, 2}, {1, 3}, {2, 3}, {3, 4}, {5, 6}};
    for (auto& a : archi) g.aggiungiArco(a[0], a[1]);
    g.bfs(0);
    int c = g.componenti();                         // stampa le componenti
    cout << "numero di componenti: " << c << endl;
    return 0;
}
```

OUTPUT

```
distanza 0 -> 0: 0
distanza 0 -> 1: 1
distanza 0 -> 2: 1
distanza 0 -> 3: 2
distanza 0 -> 4: 3
distanza 0 -> 5: irraggiungibile
distanza 0 -> 6: irraggiungibile
componente 0: 0 2 3 4 1
componente 1: 5 6
numero di componenti: 2
```

LA VISITA IN AMPIEZZA

La BFS usa una coda: estrae un nodo, esamina i suoi vicini e accoda quelli non ancora scoperti, assegnando loro la distanza del nodo corrente più uno. Al posto dei colori si usa `dist[v] == -1` per dire «non ancora scoperto». Poiché ogni nodo entra in coda al più una volta, la coda può essere un semplice array di n posizioni con due indici, senza bisogno di renderla circolare. I nodi 5 e 6 non vengono mai raggiunti e restano a distanza −1.

LA VISITA IN PROFONDITÀ E LE COMPONENTI

La DFS ricorsiva visita un nodo, lo marca e si richiama su ogni vicino non visitato: quando termina, ha visitato **esattamente** i nodi raggiungibili dal nodo di partenza, cioè la sua componente connessa. Per contare le componenti basta quindi avviare una nuova DFS da ogni nodo non ancora visitato, contando gli avvii. Il costo complessivo resta O(V + E), perché ogni nodo e ogni arco vengono esaminati un numero costante di volte.

L'ordine in cui la DFS stampa i nodi (0 2 3 4 1) dipende dall'ordine delle liste di adiacenza: poiché gli archi sono inseriti in testa, ogni lista contiene i vicini nell'ordine inverso di inserimento.

ESERCIZIO

Da provare

Modificare la BFS perché memorizzi anche il predecessore di ogni nodo e stampi il cammino minimo da 0 a un nodo dato, risalendo i predecessori con una funzione ricorsiva.

CAPITOLO 9

Due esercizi completi

Gli esercizi di programmazione più completi mettono insieme una gerarchia di classi, una struttura dati realizzata a mano e alcune operazioni da svolgere su di essa. Il capitolo ne svolge due per intero, spiegando le scelte di progetto prima del codice.

Studenti e borse di studio in una coda

IL TESTO

Si consideri la classe **Studente** con attributi privati matricola, nome, cognome e media, e metodi pubblici costruttore, get degli attributi e stampa. Si consideri la classe **BorsaDiStudio** con attributi privati importo e durata. La classe **StudenteBorsista** deriva da Studente e ha un attributo privato aggiuntivo, **un puntatore a BorsaDiStudio**, e i metodi pubblici **get_importo_borsa** e **stampa**, che aggiunge l'importo della borsa. Il programma crea una **coda di dimensione 10** e vi inserisce sei studenti, tre dei quali borsisti; poi **rimuove gli studenti con media inferiore a 25**, stampando un avviso; infine stampa il contenuto della coda e il **totale degli importi delle borse**.

LE SCELTE

1. **La gerarchia.** `StudenteBorsista` deriva pubblicamente da `Studente`; `stampa` è virtuale nella base e ridefinita nella derivata, che chiama `Studente::stampa()` e aggiunge la borsa.
2. **Il totale.** `get_importo_borsa` è dichiarato virtuale già in `Studente`, dove restituisce 0: il totale si calcola con un solo ciclo polimorfo, senza chiedersi il tipo di ogni studente. L'alternativa è il `dynamic_cast` del capitolo 3.
3. **La borsa.** È un oggetto separato collegato da un puntatore. Lo studente borsista la riceve già allocata e ne diventa proprietario: la libera nel proprio distruttore. Per questo il distruttore della base deve essere virtuale.
4. **La coda.** La dimensione fissa suggerisce l'array circolare del capitolo 6, qui con elementi di tipo `Studente*`: i puntatori alla base permettono di mettere nella stessa coda studenti e borsisti.
5. **La rimozione.** Si usa la rotazione: si estraggono tanti elementi quanti ne contiene la coda e ciascuno viene cancellato oppure reinserito. Alla fine restano solo gli studenti sufficienti, nello stesso ordine. Lo stesso giro serve per la stampa e per il totale.
6. **Il formato.** Per stampare 28.0 e non 28 si usano `fixed` e `setprecision(1)`.

IL CODICE

C9_BORSE.CPP

```
// Studenti e borse di studio in una coda circolare di dimensione 10
#include <iostream>
#include <iomanip>      // fixed, setprecision: per stampare 28.0 invece di 28
#include <string>
using namespace std;

// ===== Studente
class Studente {
public:
    Studente(int m, const string& n, const string& c, double med)
        : matricola(m), nome(n), cognome(c), media(med) {}
    virtual ~Studente() {}                      // virtuale: si cancella tramite Studente*
    int getMatricola() const { return matricola; }
    const string& getNome() const { return nome; }
    const string& getCognome() const { return cognome; }
    double getMedia() const { return media; }
    virtual int get_importo_borsa() const { return 0; }   // uno studente normale: 0
    virtual void stampa() const {                  // "matricola: nome cognome, media M"
        cout << matricola << ": " << nome << " " << cognome << ", media "
             << fixed << setprecision(1) << media;
    }
private:
    int matricola;
    string nome, cognome;
    double media;
};

// ===== BorsaDiStudio
class BorsaDiStudio {
public:
    BorsaDiStudio(int imp, int dur = 12) : importo(imp), durata(dur) {}
    int getImporto() const { return importo; }
    int getDurata() const { return durata; }
private:
    int importo;
    int durata;                                    // in mesi
};

// ===== StudenteBorsista
class StudenteBorsista : public Studente {
public:
    StudenteBorsista(int m, const string& n, const string& c, double med,
                     BorsaDiStudio* b)
        : Studente(m, n, c, med), borsa(b) {}
    ~StudenteBorsista() { delete borsa; }          // la borsa appartiene allo studente
    int get_importo_borsa() const override {
        return (borsa != nullptr) ? borsa->getImporto() : 0;
    }
    void stampa() const override {
        Studente::stampa();                        // parte comune
        if (borsa != nullptr)
            cout << " [borsa di " << borsa->getImporto() << " euro]";
    }
private:
    BorsaDiStudio* borsa;                          // composizione tramite puntatore
};
```

```
// ===== Coda (array circolare)
class Coda {
public:
    Coda(int capacita) : dim(capacita), testa(0), n(0) { v = new Studente*[dim]; }
    ~Coda() { delete[] v; }                          // libera l'array (gli studenti no)
    bool isEmpty() const { return n == 0; }
    bool isFull() const { return n == dim; }
    int size() const { return n; }
    bool enqueue(Studente* s) {
        if (isFull()) return false;
        v[(testa + n) % dim] = s;                // la posizione dopo l'ultimo, circolare
        n++;
        return true;
    }
    Studente* dequeue() {                            // precondizione: coda non vuota
        Studente* s = v[testa];
        testa = (testa + 1) % dim;
        n--;
        return s;
    }
private:
    Studente** v;                                    // array di puntatori a Studente
    int dim, testa, n;
};

// ===== main
int main() {
    Coda coda(10);
    coda.enqueue(new Studente(1001, "Mario", "Rossi", 25.5));
    coda.enqueue(new StudenteBorsista(1002, "Anna", "Verdi", 28.0,
                                      new BorsaDiStudio(880)));
    coda.enqueue(new Studente(1003, "Luca", "Bianchi", 26.0));
    coda.enqueue(new Studente(1004, "Sara", "Neri", 27.5));
    coda.enqueue(new StudenteBorsista(1005, "Marco", "Gialli", 24.0,
                                      new BorsaDiStudio(500)));
    coda.enqueue(new StudenteBorsista(1006, "Laura", "Marroni", 29.0,
                                      new BorsaDiStudio(600)));

    // rimozione degli insufficienti usando solo dequeue/enqueue: un giro completo
    int quanti = coda.size();
    for (int i = 0; i < quanti; i++) {
        Studente* s = coda.dequeue();
        if (s->getMedia() < 25) {
            cout << "Avviso: rimosso " << s->getNome() << " " << s->getCognome()
                 << " (matricola " << s->getMatricola() << ")" << endl;
            delete s;                            // chiama il distruttore giusto (virtual)
        } else {
            coda.enqueue(s);                     // torna in fondo: l'ordine si conserva
        }
    }
    // stampa del contenuto e totale delle borse (polimorfismo)
    cout << "Contenuto della coda:" << endl;
    int totale = 0;
    quanti = coda.size();
    for (int i = 0; i < quanti; i++) {
        Studente* s = coda.dequeue();
        s->stampa();                                 // versione scelta a esecuzione
        cout << endl;
```

```
        totale += s->get_importo_borsa();
        coda.enqueue(s);
    }
    cout << "Totale degli importi delle borse di studio: " << totale << " euro" << endl;
    while (!coda.isEmpty()) delete coda.dequeue();      // libera tutti gli studenti
    return 0;
}
```

OUTPUT

```
Avviso: rimosso Marco Gialli (matricola 1005)
Contenuto della coda:
1001: Mario Rossi, media 25.5
1002: Anna Verdi, media 28.0 [borsa di 880 euro]
1003: Luca Bianchi, media 26.0
1004: Sara Neri, media 27.5
1006: Laura Marroni, media 29.0 [borsa di 600 euro]
Totale degli importi delle borse di studio: 1480 euro
```

VERIFICA

Perché il distruttore di Studente è virtuale?

Perché gli studenti vengono distrutti tramite puntatori `Studente*`. Senza `virtual` verrebbe eseguito solo il distruttore di `Studente` e la borsa di uno studente borsista non sarebbe mai liberata.

Perché la coda contiene Studente* e non Studente?

Per il polimorfismo: un `StudenteBorsista` copiato in una variabile `Studente` perderebbe la parte derivata e `stampa` userebbe sempre la versione della base.

Quanto costano enqueue e dequeue? E la rimozione degli insufficienti?

O(1) le prime due, perché l'array circolare non sposta elementi; O(n) la rimozione, che fa un giro completo della coda.

Il programma si adatta facilmente a varianti dello stesso problema. Con una **pila** al posto della coda si sostituisce la struttura dati (e l'ordine di stampa si inverte). Con la **lettura da file** si usa il ciclo `while (in >> matricola >> nome >> cognome >> media >> importo)` e si crea un borsista solo se l'importo è diverso da zero. Con un **BST** si riusa la classe del capitolo 7 con la matricola come chiave.

Una mediateca

IL TESTO

Una mediateca contiene elementi multimediali con titolo e anno. Gli elementi sono **libri**, con autore e numero di pagine, e **DVD**, con la durata in minuti. La mediateca memorizza gli elementi in un **array allocato dinamicamente** che cresce quando è pieno, e offre l'inserimento, l'elenco completo e la ricerca per titolo.

LE SCELTE

`Elemento` è una classe astratta con `stampa` virtuale pura; `Libro` e `Dvd` la implementano. La mediateca contiene un array di `Elemento*` e ne è **proprietaria**: il suo distruttore libera prima ogni elemento, poi l'array dei puntatori. Quando l'array è pieno se ne alloca uno di capacità doppia e si copiano i **puntatori**, non gli oggetti, che restano dove sono. La ricerca restituisce un puntatore, oppure `nullptr` se il titolo non è presente.

C9_MEDIATECA.CPP

```
// Mediateca: gerarchia di elementi multimediali in un array dinamico di puntatori
#include <iostream>
#include <string>
using namespace std;
class Elemento {
protected:
    string titolo;
    int anno;
public:
    Elemento(string t, int a) : titolo(t), anno(a) {}
    virtual ~Elemento() {}
    string getTitolo() const { return titolo; }
    virtual void stampa() const = 0;               // ogni tipo si stampa a modo suo
};
class Libro : public Elemento {
    string autore;
    int pagine;
public:
    Libro(string t, int a, string au, int p) : Elemento(t, a), autore(au), pagine(p) {}
    void stampa() const override {
        cout << "Libro: " << titolo << " (" << anno << "), " << autore
             << ", " << pagine << " pagine" << endl;
    }
};

class Dvd : public Elemento {
    int minuti;
public:
    Dvd(string t, int a, int m) : Elemento(t, a), minuti(m) {}
    void stampa() const override {
        cout << "DVD: " << titolo << " (" << anno << "), " << minuti << " minuti" << endl;
    }
};
class Mediateca {
private:
    Elemento** v;                                  // array dinamico di puntatori
    int n, cap;
public:
    Mediateca(int c = 2) : v(new Elemento*[c]), n(0), cap(c) {}
    ~Mediateca() {                                 // possiede gli elementi
        for (int i = 0; i < n; i++) delete v[i];
        delete[] v;
    }
    void aggiungi(Elemento* e) {
        if (n == cap) {                            // pieno: raddoppia la capacità
            Elemento** nuovo = new Elemento*[cap * 2];
            for (int i = 0; i < n; i++) nuovo[i] = v[i];   // si copiano i puntatori
            delete[] v;
            v = nuovo;
            cap *= 2;
        }
        v[n++] = e;
    }
    void elenco() const {
        for (int i = 0; i < n; i++) v[i]->stampa();     // binding dinamico
    }
    Elemento* cerca(const string& t) const {       // ricerca sequenziale: O(n)
```

```
        for (int i = 0; i < n; i++)
            if (v[i]->getTitolo() == t) return v[i];
        return nullptr;
    }
};

int main() {
    Mediateca m;
    m.aggiungi(new Libro("Il nome della rosa", 1980, "Umberto Eco", 503));
    m.aggiungi(new Dvd("Nuovo Cinema Paradiso", 1988, 155));
    m.aggiungi(new Libro("Il Gattopardo", 1958, "Tomasi di Lampedusa", 330));
    m.aggiungi(new Dvd("La vita e' bella", 1997, 116));
    m.elenco();
    string cercati[] = {"Il Gattopardo", "Amarcord"};
    for (const string& t : cercati) {
        Elemento* e = m.cerca(t);
        cout << "Ricerca di \"" << t << "\": ";
        if (e != nullptr) e->stampa();
        else cout << "non presente" << endl;
    }
    return 0;
}
```

OUTPUT

```
Libro: Il nome della rosa (1980), Umberto Eco, 503 pagine
DVD: Nuovo Cinema Paradiso (1988), 155 minuti
Libro: Il Gattopardo (1958), Tomasi di Lampedusa, 330 pagine
DVD: La vita e' bella (1997), 116 minuti
Ricerca di "Il Gattopardo": Libro: Il Gattopardo (1958), Tomasi di Lampedusa, 330 pagine
Ricerca di "Amarcord": non presente
```

DA RICORDARE

Due livelli di delete

La riga `delete[] v` libera solo l'array dei puntatori. Gli oggetti puntati sono stati allocati uno per uno con `new` e vanno liberati uno per uno con `delete v[i]`. Dimenticare il ciclo non provoca errori visibili, ma lascia in memoria tutti gli elementi.

CAPITOLO 10

Metodo di lavoro ed errori frequenti

Scrivere un programma di qualche centinaio di righe in un tempo limitato richiede un metodo: procedere per piccoli passi, compilare spesso, riconoscere al volo gli errori più comuni. Il capitolo raccoglie queste indicazioni e una lista di controllo finale.

Procedere per piccoli passi

1. **Leggere tutto il testo** prima di scrivere codice. Si individuano i nomi di classi, attributi e metodi (vanno usati quelli richiesti), la gerarchia («deriva da», «è un»), le composizioni («ha un», «un puntatore a»), la struttura dati e come deve essere realizzata, il formato esatto dell'output.
2. **Scrivere lo scheletro e compilarlo.** Le classi con attributi, costruttori, get e stampa, e un `main` che crea due oggetti e li stampa. Da questo momento si ha sempre un programma che compila.
3. **Realizzare la struttura dati** e provarla da sola, con pochi elementi, stampando il contenuto dopo ogni operazione.
4. **Aggiungere le funzionalità una alla volta**: lettura da file, filtri, ricerche, totali. Dopo ciascuna si compila e si confronta l'output con quello atteso.
5. **Rifinire**: commenti, liberazione della memoria, un'ultima compilazione con `-Wall` e un'esecuzione completa.

Il principio è che ogni modifica sia **piccola** e venga compilata subito: se compare un errore, si trova quasi certamente nelle ultime righe scritte. Scrivere duecento righe e compilarle alla fine significa trovarsi davanti decine di errori intrecciati.

Gli errori più frequenti

MESSAGGIO DI G++ O SINTOMO	CAUSA PROBABILE	RIMEDIO
`expected ';' after class definition`	manca il punto e virgola dopo la graffa che chiude la classe	`};`
`'x' is private within this context`	accesso a un attributo privato dall'esterno o da una derivata	usare un get, `protected` o `friend`
`passing 'const X' as 'this' argument discards qualifiers`	metodo non `const` chiamato su un oggetto costante	dichiarare il metodo `const`
`no matching function for call to 'X::X()'`	manca il costruttore di default	parametri di default, o lista di inizializzazione nella derivata
`'Base' is an inaccessible base of 'Derivata'`	ereditarietà privata	`class Derivata : public Base`
`cannot declare variable to be of abstract type`	si istanzia una classe con funzioni virtuali pure	implementarle nella derivata o usare un puntatore
`undefined reference to vtable for X`	un metodo virtuale dichiarato ma mai definito	definirlo, o renderlo virtuale puro
`undefined reference to X::contatore`	attributo statico dichiarato ma non definito	definirlo fuori dalla classe
`'X' was not declared in this scope`	classe usata prima della definizione, o header mancante	riordinare il codice o aggiungere l'include
`stray '\342' in program`	carattere non ASCII copiato da un documento	riscrivere il carattere a mano
`Segmentation fault` **durante l'esecuzione**	accesso tramite un puntatore nullo o non inizializzato	inizializzare a `nullptr` e controllare prima dell'uso
l'ultimo dato letto è elaborato due volte	ciclo `while (!in.eof())`	mettere la lettura nella condizione del ciclo

Lista di controllo

DA RICORDARE

Prima di considerare finito un programma

Compila con `g++ -Wall` senza errori, e gli avvisi sono stati letti e risolti.

Esegue fino in fondo senza interruzioni e l'output coincide con quello atteso: ordine, separatori, cifre decimali.

Gli attributi sono privati o protetti; i costruttori usano la lista di inizializzazione e le derivate chiamano il costruttore della base.

I metodi da ridefinire sono `virtual` nella base, e il distruttore è virtuale se gli oggetti vengono cancellati tramite puntatori alla base.

Ogni `new` ha il suo `delete` (`delete[]` per gli array); le strutture dati hanno un distruttore che libera tutti i nodi.

I puntatori dei nodi sono inizializzati a `nullptr`, e ogni puntatore restituito da una ricerca viene controllato prima di essere usato.

Il codice è commentato quanto basta per spiegare a voce ogni scelta.

DA RICORDARE

Controllare la memoria

Su Linux lo strumento `valgrind` esegue il programma sorvegliando ogni accesso alla memoria. Il comando `valgrind --leak-check=full ./programma` segnala letture di memoria già liberata, uso di variabili non inizializzate e, alla fine, i blocchi allocati e mai liberati, con la riga in cui sono stati allocati. È il modo più rapido per verificare che distruttori e `delete` siano completi.