



Luis  
Joyanes  
Aguilar

# Fondamenti di programmazione in C++

Algoritmi, strutture dati e oggetti

II edizione

*Edizione italiana a cura di*  
Aldo Franco Dragoni

Mc  
Graw  
Hill

Luis Joyanes Aguilar



# Fondamenti di programmazione in C++

Algoritmi, strutture dati e oggetti

*Seconda edizione*

Edizione italiana a cura di  
Aldo Franco Dragoni, Università Politecnica delle Marche



**Titolo originale:** Luis Joyanes Aguilar, *Programación en C++, Algoritmos, Estructuras de Datos y Objetos*, 2<sup>a</sup> Edición (Edizione revisionata) Copyright © 2006  
**L'autorizzazione per la seconda edizione italiana è stata concordata con McGraw Hill Interamericana de España, S.L.**



Copyright © 2021, 2008 McGraw-Hill Education (Italy) S.r.l.  
Corso Vercelli, 40 - 20145 Milano (MI)  
Tel 02535718.1 - [www.mheducation.it](http://www.mheducation.it)

I diritti di traduzione, riproduzione, memorizzazione elettronica e adattamento totale e parziale con qualsiasi mezzo sono riservati per tutti i Paesi.

Date le caratteristiche intrinseche di Internet, l'Editore non è responsabile per eventuali variazioni negli indirizzi e nei contenuti dei siti Internet riportati.

L'Editore ha fatto quanto possibile per contattare gli aventi diritto delle immagini che compaiono nel testo e resta a disposizione di chi non è stato possibile contattare.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

Le fotocopie *per uso personale* del lettore possono essere effettuate nei limiti del 15% di ciascun volume/fascicolo di periodico dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941 n. 633.

Le riproduzioni effettuate per finalità di carattere professionale, economico o commerciale o comunque *per uso diverso da quello personale* possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEAREdI, Corso di Porta Romana 108, 20122 Milano, e-mail [info@clearedi.org](mailto:info@clearedi.org) e sito web [www.clearedi.org](http://www.clearedi.org)

Portfolio Manager: Barbara Ferrario

Fotocomposizione e realizzazione editoriale: Fotocompos srl, Gussago (BS)

Grafica di copertina: Feel Italia, Milano

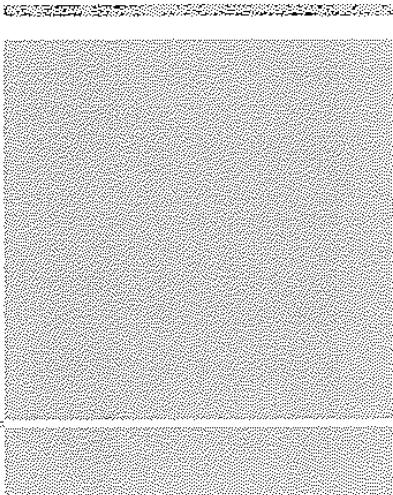
Immagine di copertina: CdTosh/Shutterstock

Stampa: Rodona Industria Gráfica, S.L.

ISBN 9788838699375

0123456789 25 24 23 22 21

# Indice breve



## **Parte I Fondamenti di programmazione**

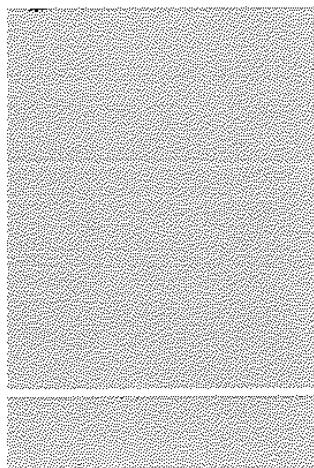
<b>Capitolo 1</b>	Introduzione all'informatica e alla programmazione	3
<b>Capitolo 2</b>	Il linguaggio C++. Elementi base	33
<b>Capitolo 3</b>	Operatori ed espressioni	65
<b>Capitolo 4</b>	La programmazione strutturata	89
<b>Capitolo 5</b>	Funzioni	123
<b>Capitolo 6</b>	Vettori e strutture	161
<b>Capitolo 7</b>	Puntatori e Riferimenti	193
<b>Capitolo 8</b>	Allocazione dinamica della memoria	221
<b>Capitolo 9</b>	Stringhe	235
<b>Capitolo 10</b>	Flussi e file:libreria standard di I/O	255

## **Parte II Fondamenti di programmazione orientata agli oggetti**

<b>Capitolo 11</b>	Classi e oggetti	289
<b>Capitolo 12</b>	Classi derivate: ereditarietà e polimorfismo	315
<b>Capitolo 13</b>	Template	341
<b>Capitolo 14</b>	Sovraccaricamento degli operatori	359
<b>Capitolo 15</b>	Eccezioni	389

## **Parte III Algoritmi e strutture dati astratte**

<b>Capitolo 16</b>	Ordinamento e ricerca	407
<b>Capitolo 17</b>	Liste	423
<b>Capitolo 18</b>	Pile e code	439
<b>Capitolo 19</b>	Ricorsione	457
<b>Capitolo 20</b>	Alberi	475



# Indice

Prefazione alla seconda edizione italiana	XI
Guida alla lettura	XII
<b>Parte I Fondamenti di programmazione</b>	1
<b>Capitolo 1 Introduzione all'informatica e alla programmazione</b>	3
Introduzione	3
1.1 Cos'è un computer?	3
1.2 Organizzazione fisica di un computer (hardware)	4
1.3 Rappresentazione dell'informazione nei computer	9
1.4 Concetto di algoritmo	12
1.5 Programmazione strutturata	14
1.6 Programmazione orientata agli oggetti	16
1.7 Il sistema operativo	22
1.8 Linguaggi di programmazione	23
1.9 Il linguaggio C	26
1.10 Il linguaggio C++	27
1.11 C versus C++	28
1.12 Il linguaggio di modellazione unificato (UML 2.0)	29
Concetti chiave	31
<b>Capitolo 2 Il linguaggio C++. Elementi base</b>	33
Introduzione	33
2.1 Costruzione di un programma in C++	33
2.2 Struttura generale di un programma C++	37
2.3 Debugging di un programma in C++	45
2.4 Elementi di un programma in C++	48
2.5 Tipi di dato predefiniti	49
2.6 Variabili	51
2.7 Durata e visibilità di una variabile	53
2.8 Istruzione di assegnamento	56
2.9 Costanti	57

<b>2.10</b> Input/output da console	60
Sommario	63
Concetti chiave	63
Esercizi	63
<b>Capitolo 3 Operatori ed espressioni</b>	65
Introduzione	65
<b>3.1</b> Operatori ed espressioni	65
<b>3.2</b> Operatore di assegnamento	70
<b>3.3</b> Operatori aritmetici	71
<b>3.4</b> Operatori di incremento e decremento	73
<b>3.5</b> Operatori relazionali	75
<b>3.6</b> Operatori logici	77
<b>3.7</b> Operatori di manipolazione dei bit	79
<b>3.8</b> Operatore condizionale	80
<b>3.9</b> Operatore virgola	81
<b>3.10</b> Operatore sizeof	81
<b>3.11</b> Conversioni di tipo	82
Sommario	85
Concetti chiave	85
Esercizi	86
<b>Capitolo 4 La programmazione strutturata</b>	89
Introduzione	89
<b>4.1</b> Strutture di controllo	89
<b>4.2</b> Istruzione if	90
<b>4.3</b> Istruzione condizionale doppia: if else	92
<b>4.4</b> Istruzioni if else annidate	93
<b>4.5</b> Istruzione switch	96
<b>4.6</b> if else e operatore condizionale ( ? : )	99
<b>4.7</b> Frequenti errori di programmazione	100
<b>4.8</b> Istruzione while	101
<b>4.9</b> Istruzione for	106
<b>4.10</b> Precauzioni nell'uso del for	109
<b>4.11</b> Istruzione do while	111
<b>4.12</b> Confronto fra while, for e do while	112
<b>4.13</b> Progetto di un'istruzione ciclica	113
<b>4.14</b> Cicli annidati	117
Sommario	119
Concetti chiave	119
Esercizi	120
<b>Capitolo 5 Funzioni</b>	123
Introduzione	123
<b>5.1</b> Concetto di funzione	124

<b>5.2</b>	Struttura di una funzione	125
<b>5.3</b>	Prototipi delle funzioni	130
<b>5.4</b>	Passaggio di parametri alla funzione	132
<b>5.5</b>	Argomenti di default	136
<b>5.6</b>	Funzioni in linea ( <i>inline</i> )	138
<b>5.7</b>	Visibilità e "storage classes" in C++	139
<b>5.8</b>	Specificatore di accesso <i>auto</i>	142
<b>5.9</b>	Funzioni di libreria	144
<b>5.10</b>	Compilazione modulare	150
<b>5.11</b>	Ricorsione	151
<b>5.12</b>	Sovraccaricamento delle funzioni	153
<b>5.13</b>	Template di funzioni	156
	Sommario	159
	Concetti chiave	159
	Esercizi	159
<b>Capitolo 6 Vettori e strutture</b>		161
	Introduzione	161
<b>6.1</b>	Array	161
<b>6.2</b>	Inizializzazione di un array	166
<b>6.3</b>	Array di caratteri e stringhe di testo	168
<b>6.4</b>	Array multidimensionali	169
<b>6.5</b>	Passaggio di vettori come parametri	172
<b>6.6</b>	Strutture	176
<b>6.7</b>	Accesso ai singoli campi delle strutture	179
<b>6.8</b>	Strutture annidate	180
<b>6.9</b>	Array di strutture	183
<b>6.10</b>	Utilizzazione di strutture come parametri	184
<b>6.11</b>	Funzioni membri di strutture	186
<b>6.12</b>	Unioni	187
	Sommario	188
	Concetti chiave	189
	Esercizi	189
<b>Capitolo 7 Puntatori e Riferimenti</b>		193
	Introduzione	193
<b>7.1</b>	Riferimenti	193
<b>7.2</b>	Puntatori	196
<b>7.3</b>	Puntatori null	199
<b>7.4</b>	Puntatore a puntatore	199
<b>7.5</b>	Puntatori e array	200
<b>7.6</b>	Array di puntatori	202
<b>7.7</b>	Puntatori a stringhe	202
<b>7.8</b>	Aritmetica dei puntatori	204

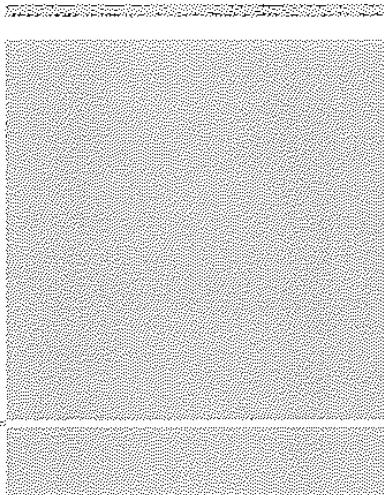
7.9	Puntatori "costanti" e puntatori "a costanti"	206
7.10	Puntatori come argomenti di funzioni	208
7.11	Puntatori a funzioni	210
7.12	Puntatori a strutture	215
Sommario		216
Concetti chiave		217
Esercizi		217
<b>Capitolo 8 Allocazione dinamica della memoria</b>		221
Introduzione		221
8.1	Gestione dinamica della memoria	221
8.2	L'operatore new	223
8.3	L'operatore delete	225
8.4	Esempi di new e delete	226
8.5	Gestione dell'overflow della memoria	230
8.6	Tipi di memoria in C++	231
Sommario		232
Concetti chiave		232
Esercizi		232
<b>Capitolo 9 Stringhe</b>		235
Introduzione		235
9.1	Concetto di stringa	235
9.2	Lettura di stringhe	237
9.3	Array e stringhe come parametri di funzioni	243
9.4	La libreria <code>cstring</code>	245
9.5	Conversione di stringhe a numeri	251
Sommario		251
Concetti chiave		252
Esercizi		252
<b>Capitolo 10 Flussi e file:libreria standard di I/O</b>		255
Introduzione		255
10.1	Flussi ( <code>stream</code> )	255
10.2	La libreria <code>iostream</code>	256
10.3	La classe <code>istream</code>	259
10.4	La classe <code>ostream</code>	264
10.5	Formattazione dell'output	267
10.6	Indicatori di formato	271
10.7	I/O da file	273
10.8	I/O binario	280
10.9	Accesso diretto	282
Sommario		285
Concetti chiave		285
Esercizi		286

<b>Parte II Fondamenti di programmazione orientata agli oggetti</b>	287
<b>Capitolo 11 Classi e oggetti</b>	289
<b>Introduzione</b>	289
<b>11.1 Classi e oggetti</b>	289
<b>11.2 Definizione di una classe</b>	290
<b>11.3 Costruttori</b>	301
<b>11.4 Distruttori</b>	305
<b>11.5 Overloading di funzioni membro</b>	307
<b>11.6 Errori frequenti di programmazione</b>	307
<b>Sommario</b>	310
<b>Concetti chiave</b>	311
<b>Esercizi</b>	311
<b>Capitolo 12 Classi derivate: ereditarietà e polimorfismo</b>	315
<b>Introduzione</b>	315
<b>12.1 Classi derivate</b>	315
<b>12.2 Tipi di ereditarietà</b>	317
<b>12.3 Distruttori</b>	325
<b>12.4 Ereditarietà multipla</b>	326
<b>12.5 Polimorfismo</b>	333
<b>12.6 Vantaggi del polimorfismo</b>	337
<b>Sommario</b>	338
<b>Concetti chiave</b>	339
<b>Esercizi</b>	339
<b>Capitolo 13 Template</b>	341
<b>Introduzione</b>	341
<b>13.1 Genericità</b>	341
<b>13.2 Template in C++</b>	342
<b>13.3 Template di funzione</b>	342
<b>13.4 Template di classe</b>	348
<b>13.5 Template e polimorfismo</b>	355
<b>Sommario</b>	356
<b>Concetti chiave</b>	356
<b>Esercizi</b>	356
<b>Capitolo 14 Sovraccaricamento degli operatori</b>	359
<b>Introduzione</b>	359
<b>14.1 Sovraccaricamento</b>	359
<b>14.2 Operatori unari</b>	361
<b>14.3 Sovraccaricamento degli operatori unari</b>	362
<b>14.4 Sovraccaricamento degli operatori binari</b>	368

14.5 Sovraccaricamento degli operatori di assegnamento	372
14.6 Sovraccaricamento degli operatori di inserimento ed estrazione	375
14.7 Sovraccaricamento di new e delete	379
14.8 Conversioni di dati e operatori di conversione forzata di tipi	381
14.9 Un'applicazione del sovraccaricamento degli operatori	383
Sommario	386
Concetti chiave	387
Esercizi	387
<b>Capitolo 15 Eccezioni</b>	389
Introduzione	389
15.1 Condizioni di errore nei programmi	389
15.2 Gestione delle eccezioni in C++	390
15.3 Specifica delle eccezioni	398
15.4 Esempi di gestione delle eccezioni	400
Sommario	403
Concetti chiave	403
Esercizi	404
<b>Parte III Algoritmi e strutture dati astratte</b>	405
<b>Capitolo 16 Ordinamento e ricerca</b>	407
Introduzione	407
16.1 Ricerca in vettori: ricerca sequenziale e binaria	407
16.2 Analisi degli algoritmi di ricerca	410
16.3 Algoritmi di ordinamento elementari	411
16.4 Ordinamento per scambio	411
16.5 Ordinamento per selezione	413
16.6 Ordinamento per inserimento	414
16.7 Ordinamento a bolle	417
16.8 Ordinamento Shell	418
Sommario	420
Concetti chiave	420
Esercizi	420
<b>Capitolo 17 Liste</b>	423
Introduzione	423
17.1 Le liste	423
17.2 Operazioni con le liste semplici	426
17.3 Lista doppiamente concatenata	432
17.4 Liste circolari	435
Sommario	435
Concetti chiave	436
Esercizi	436

<b>Capitolo 18 Pile e code</b>	439
<b>Introduzione</b>	439
<b>18.1</b> Concetto e gestione di una pila	439
<b>18.2</b> Concetto e gestione di una coda	448
<b>Sommario</b>	454
<b>Concetti chiave</b>	454
<b>Esercizi</b>	454
<b>Capitolo 19 Ricorsione</b>	457
<b>Introduzione</b>	457
<b>19.1</b> Funzioni ricorsive	457
<b>19.2</b> Confronto fra ricorsione e iterazione	460
<b>19.3</b> Soluzione di problemi attraverso la ricorsione	463
<b>19.4</b> <i>QuickSort</i>	468
<b>Sommario</b>	473
<b>Concetti chiave</b>	473
<b>Esercizi</b>	473
<b>Capitolo 20 Alberi</b>	475
<b>Introduzione</b>	475
<b>20.1</b> Gli alberi	475
<b>20.2</b> Alberi binari	479
<b>20.3</b> Struttura di un albero binario	480
<b>20.4</b> Visita di un albero	483
<b>20.5</b> Albero binario di ricerca	487
<b>Sommario</b>	492
<b>Concetti chiave</b>	492
<b>Esercizi</b>	493
<b>Indice analitico</b>	495

# Prefazione alla seconda edizione italiana



Il linguaggio di programmazione C++ è normato da un gruppo di lavoro ISO che finora ha pubblicato sei revisioni dello standard e sta attualmente lavorando alla prossima revisione, la C++23.

La prima standardizzazione risale al 1998, versione informalmente conosciuta come C++98. Nel 2003, il gruppo pubblicò una nuova versione (ISO/IEC 14882: 2003) che risolse i problemi allora identificati nel C++98.

Il testo originario *Programacion en C++, Algoritmos, estructuras de datos y objetos*, di Luis Joyanes Aguilar, fu pubblicato nel 2006 e conteneva insieme esempi e concetti relativi a entrambe le standardizzazioni degli anni precedenti, la C++98 e la C++2003, presentando spesso incoerenze che generavano confusione, problemi questi che l'edizione italiana, curata nel 2008, risolse solo in parte.

Nel 2011 fu rilasciata la successiva revisione del C++ (ISO/IEC 14882: 2011) la quale includeva molte aggiunte, sia al core del linguaggio sia alla libreria standard. Nel 2014 venne rilasciato il C++14 con piccoli miglioramenti rispetto al C++11, mentre una profonda revisione è stata recentemente fatta per il C++17, approvata e pubblicata nel dicembre 2017. Lo scorso dicembre 2020 è stata poi pubblicata l'ultima revisione dello standard, la C++20.

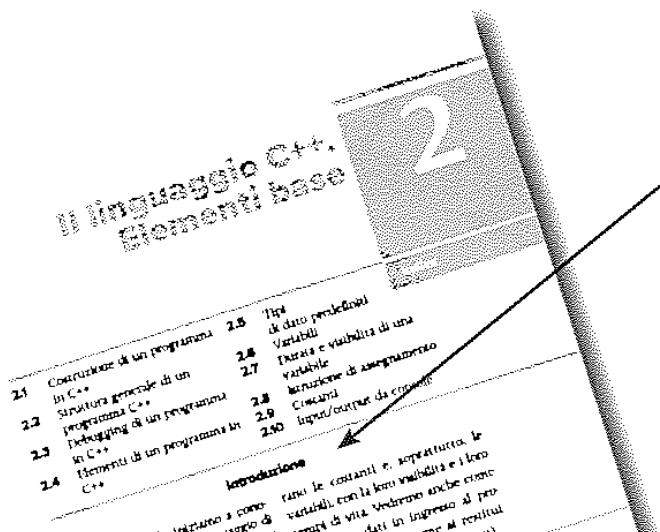
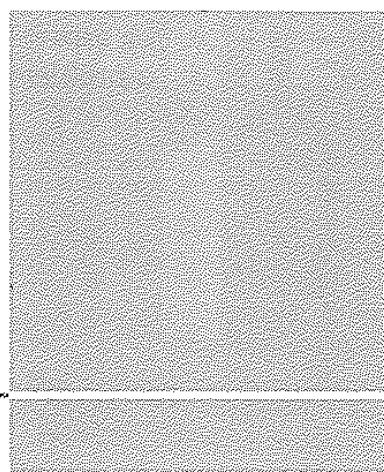
McGraw Hill ha quindi ritenuto opportuno realizzare una nuova edizione italiana del libro che tenesse conto dei nuovi standard e correggesse definitivamente gli errori presenti nella stesura originale.

Si è fatto un profondo lavoro di revisione strutturale degli esempi di codice in maniera da adeguarli alle ultime versioni del C++. Al momento nel libro non c'è codice che non sia stato testato sull'ultima versione del compilatore "GCC 10.2" e, per superare questo controllo sistematico, si è dovuto riscrivere la maggior parte dei programmi. Si è cercato di conservare però lo spirito principale dell'edizione originale, sia nella linea didattica sia nell'orientamento decisamente più pragmatico che formale. Sono state eliminate tutte le ripetizioni, le ridondanze concettuali e le nomenclature non più attuali (soprattutto relativamente alla libreria standard del C++). Sono stati totalmente riscritti alcuni capitoli nonché i codici in esso presentati.

Il risultato è un testo più asciutto ed essenziale, che dovrebbe ridurre i tempi di apprendimento dei fondamenti della programmazione in C/C++, linguaggi che l'indice TIOBE 2021 colloca rispettivamente al primo (il C) e al quarto posto (il C++) per diffusione al mondo.

Aldo Franco Dragoni  
Università Politecnica delle Marche

# Guida alla lettura



L'**Introduzione** presenta i punti chiave di ciascun capitolo e gli obiettivi di apprendimento che lo studente dovrebbe raggiungere, permettendogli di inquadrare l'argomento che si prepara a studiare.

In questo capitolo tratteremo i fondamenti del linguaggio di programmazione C++. Vedremo come creare un programma, quali sono i tipi di dato predefiniti del C++ e come si dichiarano. Vedremo cosa sono e come si dichiarano le costanti e, soprattutto, le variabili, con la loro validità e i loro tempi di vita. Vedremo anche come si leggono dati in ingresso al programma (input) e come si restituiscono i risultati in uscita (output).

2.1 Costruzione di un programma in C++  
C++ è un linguaggio di programmazione di alto livello e i suoi programmi sono scritti in ASCII, mentre i computer, come il lettore già sa, eseguono solo programmi in linguaggio macchina (sequenze d'instruzioni espresse in codice binario). Pertanto è necessario tradurre il codice sorgente in codice macchina (verrà in codice binario). Questo processo di traduzione è detto compilazione.

I programmi nascono come idee nella mente del programmatore, che poi le sviluppa fino ad abbucarne un algoritmo espresso in linguaggio naturale, in uno pseudocodice, o direttamente in un linguaggio di programmazione, come il C++.

Una volta che è stato scritto il codice del programma si deve procedere alla sua esecuzione. Le tappe da seguire dipendono dal compilatore e dall'ambiente di sviluppo utilizzati. Comunque, saranno simili alle seguenti:

7.1 Parte I Fondamenti di programmazione  
7.1.1 Esempio 3 pg  
L'espressione è sicuramente falsa e quindi il primo operando di > è tra i due casi falso falso (falsa) e quando il primo operando di > è tra i due casi vero falso (vera) e quindi il primo operando di > è tra i due casi vero vero (vera). Questa proprietà si dice valutazione in cortocircuito.

Esempio 3 pg  
Supponiamo che si debba valutare l'espressione  
 $(x = 0 \wedge 1) \wedge (y = 2)$   
se l'operando  $x = 0$  è falso (è a negativo) tutta l'espressione è sicura-mente falso, pertanto non è necessario valutare il secondo operando, la quale era città di calcolare la radice quadrata di numeri (i) definiti.

La valutazione in cortocircuito ha due benefici importanti:  
1. L'espressione AND/OR al primo membro può agire come semistella per evitare di compiere un'operazione potenzialmente costosa al secondo membro.  
2. L'espressione AND/OR al primo membro può far risparmiare una considerabile quantità di tempo nella valutazione di condizioni complesse al secondo membro.

Esempio 2 pg  
I lavori di cui si apprezzano nell'espressione bastano  
 $x = 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge 6$   
che cessa di calcolare per zero il secondo membro dato che se  $x = 0$  allora  
 $x = 0$   
e tale è la seconda espressione  
 $x = 0 \wedge 3$   
non viene valutata  
Lo stesso vantaggio si ha valutando l'espressione basata sulla della precedente  
 $x = 0 \wedge 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5$   
dato che se  $x = 0$  la prima espressione  
 $x = 0$   
e non ci bisogna quindi di valutare la seconda espressione  $\sim 1$ .  
Si può scrivere direttamente un valore di tipo bool a una variabile di tipo bool.

I numerosi **Esempi** presentano allo studente la realizzazione pratica di quanto appena esposto.

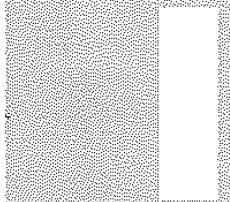
I numerosi **box** mettono in evidenza gli aspetti più importanti o problematici dell'argomento studiato.

**Il Sommario** di fine capitolo propone una sintesi degli argomenti trattati.  
**I Concetti chiave** aiutano lo studente a fissare i punti più importanti del capitolo.

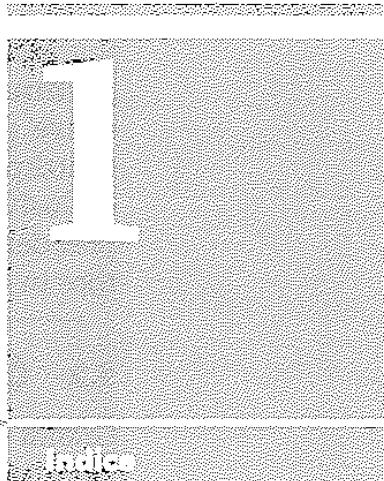
Gli **Esercizi** di fine capitolo sono un potente strumento di autoverifica per lo studente: le soluzioni sono disponibili sulla pagina web dedicata al libro, consultabile sul sito [www.mheducation.it](http://www.mheducation.it).

# **Fondamenti di programmazione**

**PARTIE**



# Introduzione all'informatica e alla programmazione



<b>1.1</b>	Cos'è un computer?	<b>1.6</b>	Programmazione orientata agli oggetti
<b>1.2</b>	Organizzazione fisica di un computer ( <b>hardware</b> )	<b>1.7</b>	Il sistema operativo
<b>1.3</b>	Rappresentazione dell'informazione nei computer	<b>1.8</b>	Linguaggi di programmazione
<b>1.4</b>	Concetto di algoritmo	<b>1.9</b>	Il linguaggio C
<b>1.5</b>	Programmazione strutturata	<b>1.10</b>	Il linguaggio C++
		<b>1.11</b>	C versus C++
		<b>1.12</b>	Il linguaggio di modellazione unificato (UML 2.0)

## Introduzione

Il libro introduce il lettore all'arte della programmazione attraverso uno dei linguaggi più versatili e diffusi: il C++. Questo capitolo inizia descrivendo l'organizzazione fisica (*hardware*) e logica (*software applicativo e sistema operativo*) del computer. S'introducono poi i concetti di "digitalizzazione dell'informazione" e

quello di "algoritmo". Si passa poi ai due paradigmi di programmazione caratteristici del C++: la *programmazione strutturata* e la *programmazione orientata agli oggetti*, per concludere introducendo l'*Unified Modelling Language* (UML) come strumento per la descrizione e risoluzione dei problemi.

### 1.1 Cos'è un computer?

Un **computer** è un dispositivo elettronico utilizzato per processare informazione digitale in **ingresso** (*input*) e ottenere risultati digitali in **uscita** (*output*), come si osserva nella Figura 1.1. Sia i dati in ingresso sia quelli in uscita possono essere numeri, testi, immagini, suoni o filmati. Il modo più semplice per interagire con il computer è utilizzare *mouse*, tastiera e schermo (monitor), ma oggi esistono altri dispositivi molto popolari come scanner, microfoni, altoparlanti, videocamere, fotocamere digitali ecc.; tramite *modem* e schede di rete è poi possibile collegare il computer ad altri attraverso reti, la più importante delle quali è **Internet**.

Con il termine **hardware** si denoma l'insieme dei componenti fisici che costituiscono il computer e i dispositivi che realizzano i compiti di

ingresso e uscita dei dati. Nella memoria del computer risiede il **programma**, cioè la sequenza delle istruzioni che esso esegue; la persona che scrive programmi si chiama **programmatore** e l'insieme dei programmi scritti per un computer si denomina **software**. Questo libro si occupa quasi esclusivamente di software, ma fa un breve cenno anche all'hardware, da prendere come ripasso o introduzione, a secondo delle conoscenze del lettore in questa materia.

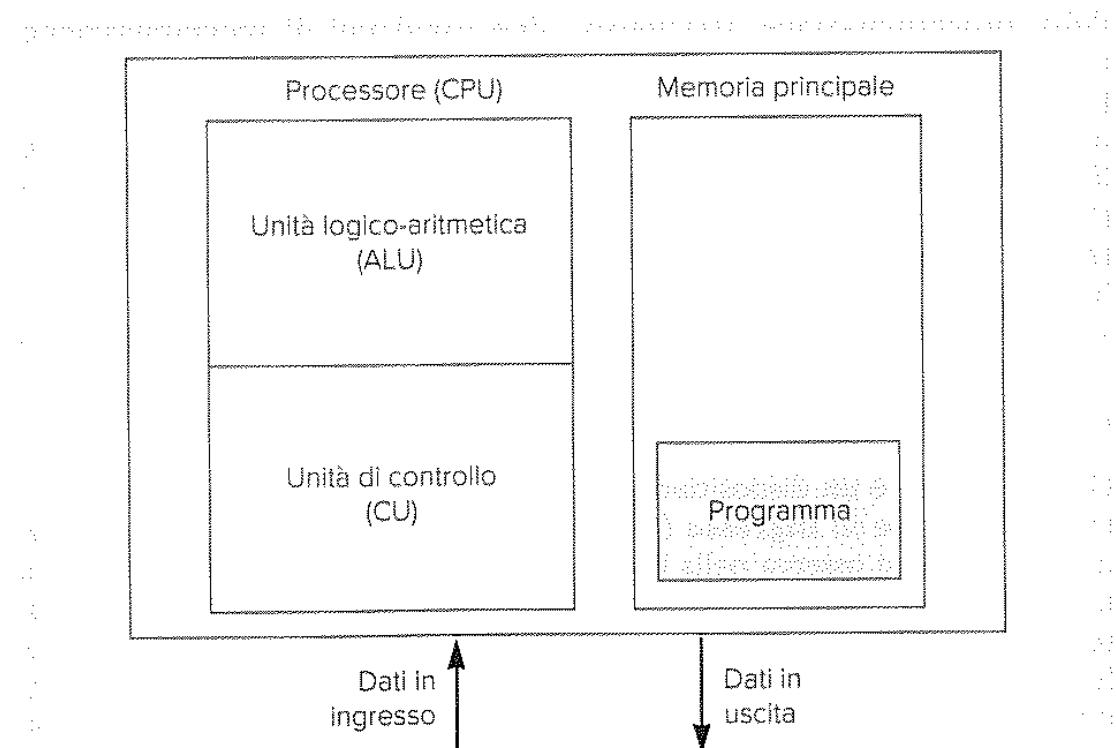
## 1.2 Organizzazione fisica di un computer (hardware)

Grandi o piccoli che siano, i sistemi per l'elaborazione digitale delle informazioni sono organizzati come illustrato in Figura 1.1. Possiedono tre componenti principali: il *processore* o *Central Processing Unit (CPU)*, costituito dall'*Arithmetic Logic Unit (ALU)* e dalla *Control Unit (CU)*, la *memoria principale* o *centrale* e il *programma*.

Se a questa struttura fondamentale si aggiungono i *dispositivi di input/output* e la *memoria secondaria* si ottiene lo schema fondamentale di un elaboratore digitale, denominato "architettura di Von Neumann" (Figura 1.2).

### 1.2.1 Dispositivi di input/output

I dispositivi di *ingresso/uscita* (input/output - I/O) sono le componenti *periferiche* del sistema per la comunicazione fra il computer e l'utente.



**Figura 1.1**  
Organizzazione fisica di un computer.

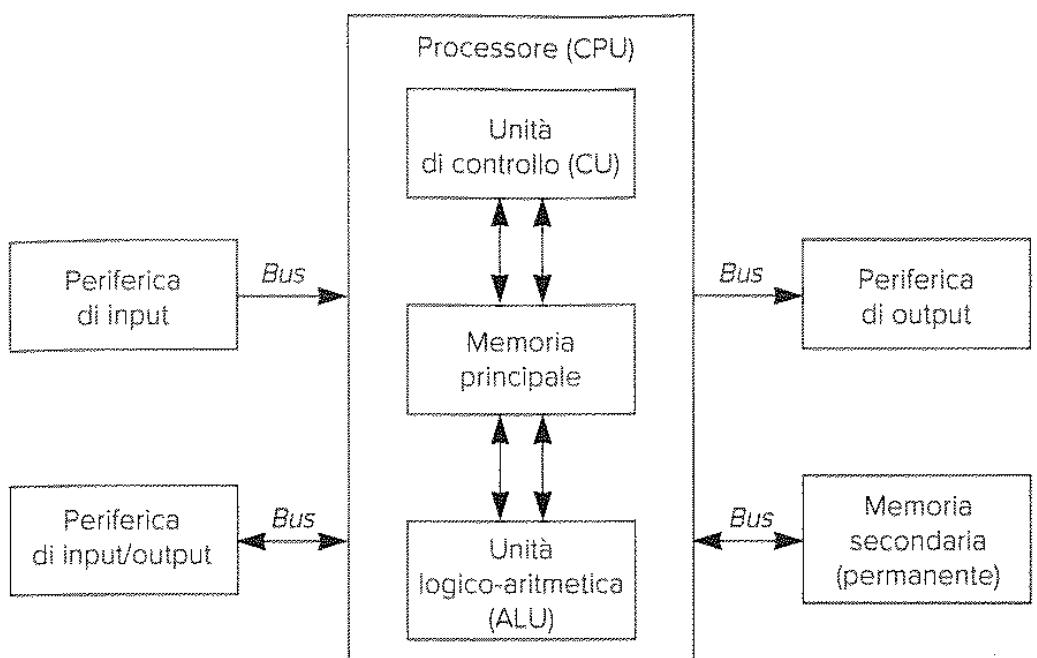


Figura 1,2

Architettura di Von Neumann.

I dispositivi di ingresso servono a introdurre dati (informazione digitale) e/o programmi nella memoria principale del computer. Le principali periferiche di input sono **tastiere**, **penne ottiche**, **joystick**, **lettori di codici a barre**, **scanner**, **microfoni**, **videocamere**, **lettori di tessere digitali** e **lettori RFID** (sistemi per l'identificazione tramite radio frequenza). Ma il dispositivo di input forse più popolare è il **mouse**, che muove un puntatore su uno schermo grafico facilitando l'interazione uomo-macchina.

I dispositivi di uscita servono per presentare i risultati dell'elaborazione dei dati. La principale periferica di output è lo **schermo o monitor**. Altri dispositivi di uscita sono le **stampanti**, i **plotters**, gli **altoparlanti** ecc.

I dispositivi di memoria secondaria come **dischi magnetici**, **CD-ROM**, **DVD**, **memorie flash**, **USB** ecc., sono anch'essi annoverati fra le periferiche di I/O.

## 1.2.2 La memoria centrale (principale)

La **memoria centrale**, o **principale**, può contenere informazioni di due tipi: **istruzioni** (di un programma) e **dati** (su cui operano le istruzioni). Perché un programma possa essere **eseguito**, deve essere prima caricato in memoria (probabilmente dalla memoria secondaria) con un'operazione denominata **loading** (caricamento). Poi, in corso di **esecuzione**, esso dovrà leggere anche dalla memoria i dati di cui ha bisogno, e metterà sempre nella memoria i

risultati di elaborazioni parziali, come anche informazioni funzionali al corretto espletamento del suo lavoro.

Per aumentare la velocità di accesso ai dati, i processori attuali utilizzano una piccola memoria intermedia, particolarmente veloce, denominata *cache*, in cui vengono collocati dati e istruzioni più frequentemente utilizzati dal programma in esecuzione. I microprocessori più veloci addirittura incorporano una propria memoria *cache*.

#### Organizzazione della memoria

Per **bit** (*binary digit*) s'intende un contenitore astratto d'informazione binaria, cioè 0 o 1. Una sequenza di 8 bit è detta **byte**. La memoria è costituita da **celle** tutte uguali e capaci di rappresentare più bit. Esse sono i mattoni fondamentali della memoria e sono scritte o lette per intero in un'unica operazione. La capacità della memoria si misura quindi in byte.

Se si associa a ogni carattere dell'alfabeto una particolare sequenza di 8 bit, allora la frase:

Ciao Mortimer va tutto bene.

sarà lunga 28 byte: le 23 lettere, 4 spazi (uno spazio è pure esso un carattere rappresentato da un particolare byte) e 1 punto, mentre il numero del passaporto

P5-748-7891

occuperà 11 byte. Questo tipo di dato, ovvero il dato "testuale", è detto *alfanumerico* perché costituito da lettere dell'alfabeto, simboli di punteggiatura, cifre numeriche e caratteri speciali (simboli: \$, #, \* ecc.).

I numeri veri e propri, cioè quelli che sono oggetto di operazioni matematiche, vengono invece rappresentati in modo del tutto diverso e possono occupare 2, 4 o 8 byte consecutivi.

Alla cella di memoria sono associati due concetti fondamentali: il suo *indirizzo* e il suo *contenuto*. L'*indirizzo* ne indica univocamente la posizione e viene utilizzato dalle istruzioni per scrivere o leggere il byte di quella cella. Questo byte ne costituisce il suo *contenuto*. La Figura 1.3 mostra una memoria di 1.000 parole con indirizzi da 0 a 999.

Quando si scrive un byte in una cella di memoria quello che vi era scritto prima viene cancellato senza possibilità di recupero.

Per definire la capacità di una memoria si utilizzano multipli di potenze di 2, come descritto nella Tabella 1.1.

Oggi i Personal Computer (PC) hanno comunemente una memoria che va da 512 MB a 32 GB.

Nella memoria principale si trovano:

- i dati inviati dai dispositivi di ingresso per essere elaborati;
- i programmi che elaborano le informazioni digitali;
- i risultati ottenuti dalle elaborazioni pronti per essere inviati a un dispositivo di uscita.

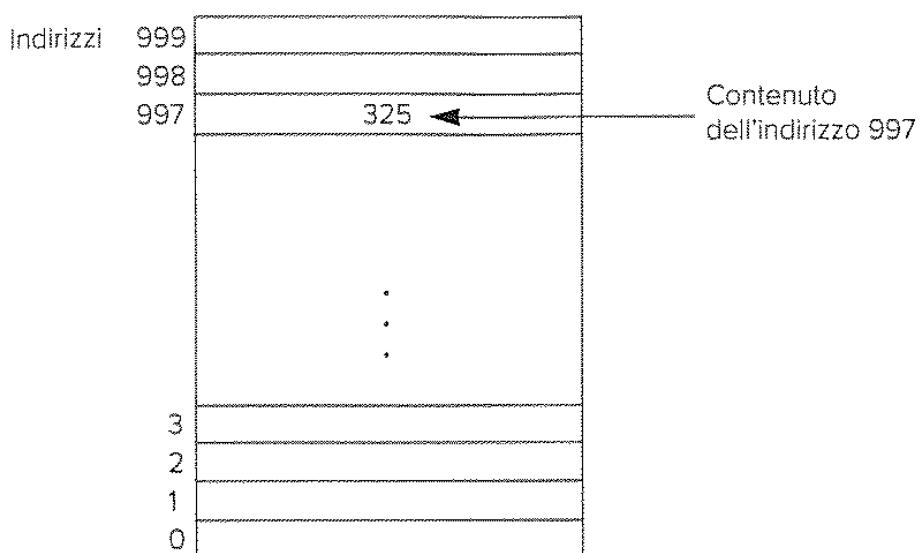


Figura 1.3

Organizzazione sequenziale della memoria centrale

Tabella 1.1 Unità di misura della memoria

Byte	Byte (B)	8 bit			
Kilobyte	Kbyte (KB)	$2^{10}$ byte	cioè 1.024 byte	in decimale	1.024 byte
Megabyte	Mbyte (MB)	$2^{20}$ byte	cioè 1.024 Kbyte	in decimale	1.048.576 byte
Gigabyte	Gbyte (GB)	$2^{30}$ byte	cioè 1.024 Mbyte	in decimale	1.073.741.824 byte
Terabyte	Tbyte (TB)	$2^{40}$ byte	cioè 1.024 Gbyte	in decimale	1.099.511.627.776 byte

### Tipi di memoria principale

Vi sono due tipi di memoria principale: la RAM e la ROM. La **RAM** (*Random Access Memory*, memoria ad accesso aleatorio) è volatile (cioè perde il suo contenuto quando si spegne il computer) e può essere sia scritta che letta accedendo direttamente a ogni sua cella; essa di fatto coincide con ciò che comunemente s'intende per "memoria principale". La **ROM** (*Read Only Memory*, memoria di sola lettura) è una memoria elettronica permanente (le informazioni non si perdono quando si spegne il computer) in cui non si può scrivere (viene prescritta dal produttore), per questo la ROM contiene normalmente i programmi che servono ad avviare il computer (*Basic Input Output System*, BIOS).

### 1.2.3 L'unità centrale di processo

L'**unità centrale di processo** (*Central Processing Unit*, CPU) è il cuore del sistema di elaborazione. Essa gestisce l'informazione (programmi e dati) leggendoli dalla memoria, eseguendo le istruzioni e scrivendo in memoria i risultati.

Essa contiene una *unità di controllo* (*Control Unit, CU*) e una *unità aritmetico-logica* (*Arithmetic Logic Unit, ALU*) (Figura 1.1). L'**unità di controllo** coordina le attività del computer, determinando le operazioni che si debbono realizzare e in che ordine. L'**unità aritmetico-logica** compie le operazioni aritmetiche e logiche, come somma, sottrazione, moltiplicazione, divisione e operazioni di confronto.

#### 1.2.4 Il microprocessore

Il **microprocessore** è il *chip* (*circuito integrato*) che realizza le funzioni della CPU.

Il primo microprocessore commerciale, l'Intel 4004, fu presentato il 15 novembre 1971. Esistono vari produttori di microprocessori, come Zilog, Motorola, Intel, AMD, STMicroelectronics ecc. Storici microprocessori a 8 bit furono l'Intel 8080, lo Zilog Z80 e il Motorola 6800. Il primo PC IBM nacque con gli 8086 e 8088 di Intel, mentre la Apple si affidò al Motorola MC68000 per il suo Macintosh. Negli anni ottanta è stata la volta dell'architettura a 32 bit dell'Intel con 80386 e 80486. Poi sono venuti i Motorola MC 68020, MC68400 e gli AMD 80386, 80486.

Nel 1993 apparve l'Intel Pentium (80586) e poi Intel Pentium Pro, Intel Pentium II/III e AMD K6. Nel 2000 Intel e AMD controllavano il mercato con Pentium IV, Titanium, Pentium D e AMD Athlon XP, AMD Duxor. Negli anni 2005 e 2006 apparvero le nuove tecnologie **Intel Core Duo, AMD Athlon 64** ecc.

#### 1.2.5 Memoria secondaria

Per poter essere eseguito, un programma deve risiedere in memoria, ma questa ha capacità limitata e si cancella quando si *spegne* il computer. Per questo servono **dispositivi di memoria secondaria**, o "di massa" (*mass storage* o *secondary storage*).

I più comuni sono *dischi magnetici, CD-ROM e DVD*.

Il *Compact Disk (CD)* è un supporto ottico sviluppato nel 1980 e commercializzato a partire dal 1982. Ne esistono tre tipi: il **CD-ROM** (di sola lettura), il **CD-R** (registrabile) e il **CD-RW** (riscrivibile). La loro capacità di immagazzinamento va da 650 MB a 875 MB. I **DVD** (*Digital Versatile Disk*) hanno una tecnologia analoga ma una capacità maggiore, sufficiente a contenere film digitali. I formati più popolari sono DVD-ROM, DVD±R, DVD±RW, DVD±RAM, e le loro capacità variano da 4,7 GB, 8,5 GB fino a 17,1 GB, a seconda se sono a uno, due strati o doppia faccia. Nel 2006 sono apparsi **Blu-ray** e **HD DVD**. La loro capacità va da 15 GB sino a 200 GB.

L'informazione nella memoria di massa si organizza in unità indipendenti chiamati **file** (archivi) costituiti da semplici sequenze di byte. I file possono contenere *dati o programmi* e devono essere trasferiti nella memoria principale per essere *elaborati o eseguiti*.

## Confronto tra memoria principale e secondaria

La memoria principale è molto più veloce e cara della memoria secondaria. I dati nella memoria principale sono *volatili* e spariscono quando si *spegne* il computer mentre quelli in memoria secondaria sono *permanenti*. Per poter essere processati i file dei programmi e dei dati debbono essere prima caricati nella memoria principale.

La **scheda di rete** consente di collegare un computer a una **rete locale** per accedere alle risorse di altri computer. Il **modem** mette in comunicazione due computer attraverso la rete telefonica analogica. Grazie a queste capacità di connessione il computer può anche collegarsi a una rete dati privata Intranet/Extranet o con la grande e pubblica Internet. Le reti wireless permettono collegamenti senza fili su brevi distanze (IEEE 802.11x, BlueTooth, ZigBee) o grandi (Wi-Max).

## 1.3 Rappresentazione dell'informazione nei computer

I computer elaborano informazione in formato digitale, per cui è necessario introdurre un po' l'argomento della "digitalizzazione" e capire come testi, immagini e suoni possano essere tradotti in sequenze di bit.

Altri temi importanti sono le tecniche per la compressione dei file per ridurre lo spazio da loro occupato e la durata della loro trasmissione, nonché i metodi per riconoscere gli errori nella trasmissione dei file.

### 1.3.1 Rappresentazione di testi

I testi si rappresentano tramite un codice in cui a ciascuno dei diversi caratteri (lettere dell'alfabeto, cifre e segni di interpunkzione) si assegna una particolare sequenza di bit. I caratteri sono di cinque tipi:

1. **alfabetici** (lettere maiuscole e minuscole);

A, B, C, D, E, ... X, Y, Z, a, b, c, ... , x, y, z

2. **numerici** (cifre del sistema decimale);

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

3. **speciali** (simboli ortografici e matematici);

( ) Ñ ñ ! ? & > # ¢...

4. **geometrici e grafici** (simboli per rappresentare figure geometriche);

| —| —|| ♠ , ...

5. **di controllo** (rappresentano comandi, come il carattere per passare alla linea successiva [NL], per tornare all'inizio della linea [RC] - *Carriage Return*, CR -, emettere un "bip" [BEL] ecc.).

Le codifiche più utilizzate per rappresentarli sono **ASCII** e **Unicode**.

- **Codice ASCII** (*American Standard Code for Information Interchange*).  
Il codice ASCII base utilizza 7 bit e permette quindi di rappresentare  $2^7 = 128$  caratteri, ma le lettere maiuscole e minuscole sono limitate a quelle dell'alfabeto inglese. In realtà però le celle di memoria sono di un byte ciascuna, per cui si aggiunge un bit settato a zero all'inizio della sequenza per assegnare un carattere a ogni cella.
- **Codice Unicode** ([www.unicode.org](http://www.unicode.org)).  
Questo codice utilizza due byte per rappresentare ogni simbolo, quindi permette di rappresentare  $2^{16} = 65.536$  simboli differenti. I nuovi posti sono utilizzati per superare il limite dell'ASCII e contemplare anche i caratteri alfabetici di tutte le lingue parlate.

Un testo diventa quindi una sequenza di byte, cioè un **file di testo**, che potrà essere in formato ASCII oppure Unicode. Questi file sono normalmente costruiti e modificati da programmi che si chiamano **editori di testo** e vanno distinti da quelli più elaborati prodotti da *word processors* tipo Microsoft Word perché questi ultimi contengono sequenze di byte che rappresentano cambi di formato, di tipi di fonti ecc. e possono utilizzare codici proprietari diversi da ASCII o Unicode.

In C++ per rappresentare i *caratteri* si usa il tipo predefinito `char`, che associa ogni carattere a un codice ASCII esteso a 8 bit (256 caratteri diversi).

### 1.3.2 Rappresentazione di valori numerici

Per rappresentare valori numerici non conviene utilizzare la codifica testuale. Se, per esempio, rappresentiamo il numero 65 come sequenza di caratteri ASCII, utilizzando un byte per simbolo occupiamo 16 bit. Con due byte il maggior numero intero rappresentabile sarebbe 99. Ma se invece di utilizzare la codifica testuale per rappresentare il numero utilizziamo direttamente la *notazione binaria* (sistema posizionale come il decimale, ma con base 2 invece che 10), il range di numeri interi rappresentabili con 16 bit andrà da 0 a 65.535 ( $2^{16} - 1$ ). Pertanto, la notazione binaria (o varianti di essa) viene preferita per la codifica dei numeri. Il numero viene originalmente immesso come testo (sequenza di cifre ASCII), ma poi l'applicazione che userà quel valore numerico provvederà a convertirlo nella notazione binaria specifica per quel tipo di numero. Esistono infatti due tipi di dati numerici: gli interi e i numeri reali.

#### Rappresentazione dei numeri interi

I dati di tipo intero (`int`, in C++) si rappresentano normalmente su quattro byte.

Gli interi si possono rappresentare con segno (`signed`, in C++) o senza segno (`unsigned`, in C++). Poiché un bit serve per specificare il segno, gli interi senza segno possono contenere valori positivi il doppio più grandi. Normalmente, se non viene specificato diversamente l'intero è inteso essere considerato con il segno. Il C++ consente poi variazioni sul tipo intero.

### Rappresentazione dei numeri reali

I numeri reali sono quelli che contengono una parte decimale, come 2,6 (due virgola sei) o 3,14152. Essi si rappresentano in *notazione scientifica*, anche detta "in *virgola mobile*". Nel sistema anglosassone la virgola decimale è rappresentata dal punto, per cui questi numeri vengono detti di tipo *floating point*.

Questi numeri vengono rappresentati secondo lo standard **IEEE 754** per il calcolo in virgola mobile.

### 1.3.3 Rappresentazione delle immagini

Esistono due modi per rappresentare immagini: la rappresentazione *raster* e quella *vettoriale*.

Nel primo caso l'immagine viene rappresentata come matrice di piccoli quadratini, chiamati *pixel* (*picture element*), ognuno dei quali possiede un suo colore uniforme. Associando a ogni colore un codice binario, l'immagine viene rappresentata come una sequenza di bit, denominata *bitmap*. Per esempio, nel caso di un'immagine in bianco e nero, possiamo associare al bianco l'"1" e al nero lo "0", così che, una volta scomposta in pixel, l'immagine viene codificata in una sequenza di bit. Per immagini con toni di grigio si associa a ogni tono un byte, dal nero puro che ha lo "0" al bianco puro che è associato al "256". Così l'immagine diventa una sequenza di byte. I colori si scompongono invece nei tre componenti principali, Rosso, Giallo e Blu (rappresentazione *RGB*), ognuna delle quali è associata a un byte che ne rappresenta l'intensità. Un'immagine di  $n$  pixel diventa quindi un file di  $3n$  byte, dove ogni 3 byte consecutivi rappresentano una particolare combinazione di Rosso, Giallo e Blu associata a un certo pixel. Questi byte possono poi essere collocati sul file con schemi diversi. A causa della notevole dimensione di questi file sono stati studiati vari modi per comprimerli senza compromettere significativamente la qualità delle immagini raster. Ne risulta una varietà di formati, i più utilizzati dei quali sono riportati nella Tabella 1.2.

La **rappresentazione vettoriale** (Tabella 1.3) si usa per immagini artificiali (disegni tecnici, caratteri tipografici ecc.) e consiste nello scomporre l'immagine in una collezione di oggetti come linee, poligoni e testi con i loro rispettivi attributi e dettagli (spessore, colore ecc.).

### 1.3.4 Rappresentazione dei suoni

La digitalizzazione dei suoni avviene in tre fasi. Prima si campiona l'ampiezza dell'onda acustica (catturata da un microfono e convertita in onda elettrica)

**Tabella 1.2** I più diffusi formati di immagini raster

Formato	Origine e descrizione
<b>BMP</b>	<b>Microsoft.</b> Formato semplice con immagini di grande qualità ma con l'inconveniente di occupare molto spazio (non utile per il web)
<b>JPEG</b>	Gruppo <b>JPEG</b> . Qualità accettabile per immagini naturali. Include compressione con perdita d'informazione. Si utilizza nel web
<b>GIF</b>	<b>CompuServe.</b> Molto adeguato per immagini non naturali (logotipi, bandiere, disegni animati...). Molto usato nel web

Tabella 1.3 I più diffusi formati di immagini vettoriali

Formato	Descrizione
<b>IGES</b>	ASME/ANSI. Standard per scambio di dati e modelli di (AutoCAD...)
<b>Pict</b>	Apple Computer. Immagini vettoriali
<b>EPS</b>	Adobe Computer
<b>TrueType</b>	Apple e Microsoft per EPS

a un certo istante di tempo, poi si "quantizza" il campione estratto per poterlo rappresentare su uno o due byte, quindi si codifica in binario il valore ottenuto e lo si registra in memoria. Tutto il procedimento viene ripetuto a intervalli regolari detti intervalli di campionamento.

Per esempio, per la registrazione stereo ad alta fedeltà si utilizza una frequenza di campionamento di 44.000 Hz e una quantizzazione su 2 byte per canale (4 quindi in tutto). Un'ora di musica stereo occupa quindi  $44.000 \times 3.600 \times 4 = 633.600.000$  byte, cioè circa la capacità dei primi CD audio.

Un sistema totalmente diverso per rappresentare la musica è il formato MIDI (*Musical Instruments Digital Interface*); questi file non sono però registrazioni digitali ma spartiti musicali opportunamente codificati e riprodotti mediante sintetizzatori denominati *sequencer*.

## 1.4 Concetto di algoritmo

L'obiettivo di questo testo è quello di insegnare a programmare. Il programmatore è anzitutto una persona che risolve problemi, perciò per riuscire a essere un buon programmatore bisogna imparare a risolvere problemi in modo rigoroso e sistematico. Con l'espressione **metodologia della programmazione** intendiamo la *metodologia necessaria per concepire algoritmi risolutivi di problemi e implementarli mediante opportuni linguaggi di programmazione*.

Un algoritmo è un metodo per risolvere un problema. Il termine **algoritmo** deriva da *Mohammed al-Khowārizmi*, matematico persiano che visse nel IX secolo e assunse a grande fama grazie all'enunciazione di regole "passo passo" per sommare, sottrarre, moltiplicare e dividere numeri decimali. Dalla latinizzazione del suo cognome derivò poi il termine "algoritmo". Euclide, il grande matematico greco che inventò un metodo per trovare il massimo comune divisore di due numeri è considerato l'altro padre dell'algoritmica.

Il professore Niklaus Wirth – inventore di Pascal, Modula-2 e Oberon – titolò uno dei suoi più famosi libri, *Algoritmi + Strutture di dati = Programmi*, volendo dire che si può realizzare un buon programma soltanto con il progetto di un algoritmo e una corretta struttura di dati. Questa equazione sarà una delle ipotesi fondamentali di questo libro.

La soluzione di un problema richiede l'elaborazione di un algoritmo per risolverlo. I passi sono elencati di seguito.

1. *Analisi del problema e sviluppo dell'algoritmo*, che descrive la sequenza ordinata di passi – senza ambiguità – che conducono alla soluzione di un certo problema.

2. *Codifica dell'algoritmo* in un programma scritto mediante un opportuno linguaggio di programmazione.
3. *Esecuzione e validazione* del programma sul computer.

Senza algoritmo non può esistere un programma, ma gli algoritmi sono indipendenti, sia dal linguaggio di programmazione in cui si esprimono sia dal computer che li eseguirà. Un algoritmo può essere scritto in linguaggi diversi ed essere eseguito su computer diversi, ma sarà sempre lo stesso algoritmo. Gli algoritmi sono quindi più importanti dei linguaggi di programmazione e dei computer. Un linguaggio di programmazione è soltanto un mezzo per esprimere un algoritmo e un computer è soltanto un processore per eseguirlo. Tanto il linguaggio di programmazione quanto il computer sono i mezzi per ottenere un fine: riuscire a eseguire l'algoritmo ed effettuare il processo corrispondente.

Quindi è molto importante imparare a progettare algoritmi. All'insegnamento e alla pratica della progettazione di algoritmi è dedicata gran parte di questo libro. Si tratta di un'attività che richiede creatività e conoscenze profonde delle tecniche di programmazione.

#### 1.4.1 Caratteristiche degli algoritmi

Un buon algoritmo deve essere:

- *preciso*: deve indicare cioè la sequenza corretta di ogni passo;
- *deterministico*: ogni volta che si applica deve produrre lo stesso risultato;
- *finito*: deve comportare un numero finito di passi.

#### Esempio 1.1

Un cliente emette un ordine a un fornitore. Il fornitore esamina nel suo database la scheda del cliente; se il cliente è solvente allora l'impresa accetta l'ordinazione, altrimenti la rifiuterà. Scrivere l'algoritmo corrispondente. I passi dell'algoritmo sono:

1. Inizio.
2. Leggere l'ordine.
3. Esaminare la scheda del cliente.
4. Se il cliente è solvente, accettare ordine; in caso contrario, rifiutare ordine.
5. Fine.

#### Esempio 1.2

Concepire e sintetizzare un algoritmo per capire se un numero è primo oppure no. Un numero è primo se può essere diviso solo per sé stesso, oltre che per il numero 1. L'algoritmo di risoluzione del problema passa per il dividere successivamente il numero per 2, 3, 4 e così via.

1. Inizio.
2. Porre X uguale a 2 ( $x = 2$ , x variabile che rappresenta i divisori del numero esaminato N).

3. Dividere N per X ( $N/X$ ).
4. Se il risultato di  $N/X$  è intero, allora N non è un numero primo e andare al punto 7; in caso contrario, continuare il processo.
5. Sommare 1 a X ( $X \leftarrow X + 1$ ).
6. Se X è uguale a N, allora N è un numero primo; in caso contrario, andare al punto 3.
7. Fine.

Per esempio, se N è 131, i passi sarebbero:

1. Inizio.
2.  $X = 2$ .
- 3 e 4.  $131/X$ . Siccome il risultato non è intero, si continua il processo.
5.  $X \leftarrow 2 + 1$ , dunque  $X = 3$ .
6. Siccome X non è 131, si va al punto 3.
- 3 e 4.  $131/X$  risultato non è intero.
5.  $X \leftarrow 3 + 1$ ,  $X = 4$ .
6. Siccome X non è 131 si va al punto 3.
- 3 e 4.  $131/X$ ..., ecc.
7. Fine.

#### ESEMPIO 1.2

Concepire un algoritmo per calcolare la somma dei numeri pari compresi fra 2 e 1.000. Il problema consiste nel sommare  $2 + 4 + 6 + 8 \dots + 1.000$ . Utilizzeremo le parole SOMMA e NUMERO (variabili, saranno introdotte più tardi) per rappresentare le somme successive  $(2+4)$ ,  $(2+4+6)$ ,  $(2+4+6+8)$  ecc. La soluzione del problema si può trovare con il seguente algoritmo:

1. Inizio.
2. Porre SOMMA a 0.
3. Porre NUMERO a 2.
4. Sommare NUMERO a SOMMA. Il risultato sarà il nuovo valore della somma (SOMMA).
5. Incrementare NUMERO di 2 unità.
6. Se NUMERO  $\leq 1.000$  andare al passo 4;
7. in caso contrario, scrivere l'ultimo valore di SOMMA e terminare il processo.
8. Fine.

## 1.5 Programmazione strutturata

C, Pascal e FORTRAN sono esempi di *linguaggi procedurali*. Ogni istruzione indica al compilatore di realizzare un compito: caricare un dato dall'input, produrre un output, sommare tre numeri, dividere per cinque ecc. In un linguaggio procedurale, un programma è una sequenza di istruzioni o sentenze. Nel caso di piccoli programmi, questo paradigma computazionale è efficiente; il programmatore deve solo scrivere questa lista di istruzioni in un linguaggio di programmazione, compilarla nel computer ed esso le eseguirà.

Ma quando i programmi si fanno più grandi perché aumenta la complessità del problema da risolvere, la lista di istruzioni cresce al punto che il programmatore ha parecchie difficoltà a mantenere il controllo di tutto il codice (generalmente possono arrivare a comprendere qualche centinaia di linee di codice). Per risolvere questo problema i programmi si compongono di unità più piccole, le *funzioni* (*sottoprogrammi*, *subroutines* in altri linguaggi di programmazione). Ogni funzione risolve un compito definito e possiede un'interfaccia chiaramente definita (il *prototipo* o *intestazione* della funzione) con la quale comunicare con altre funzioni.

Con il tempo, l'idea di scomporre un programma in funzioni portò al raggruppamento delle funzioni in unità più grandi chiamate *moduli*, ma il principio continuava a essere lo stesso. Man mano che i programmi si fanno più grandi e complessi, il paradigma della programmazione strutturata comincia a dare segni di debolezza, essendo molto difficile gestire grandi programmi in maniera efficiente.

Un primo motivo di debolezza nella programmazione strutturata sta nel fatto che le funzioni hanno accesso anche ai dati globali, e quindi potrebbero modificare valori che poi vanno ad alterare il funzionamento di altre funzioni, rendendo quindi imperfetto il concetto di programmazione per moduli fra loro indipendenti. Infatti, in un programma procedurale esistono due tipi di dati:

- *dati locali*, sono definiti dentro una funzione (ovvero sono "locali" a quella funzione) e sono implicitamente protetti da modifiche da parte di altre funzioni;
- *dati globali*, sono definiti al di fuori di qualunque funzione e sono quindi accessibili da qualunque funzione del programma (Figura 1.4).

Un programma è normalmente costituito da molte funzioni e dati globali, e ciò comporta una moltitudine di collegamenti tra funzioni e dati con conse-

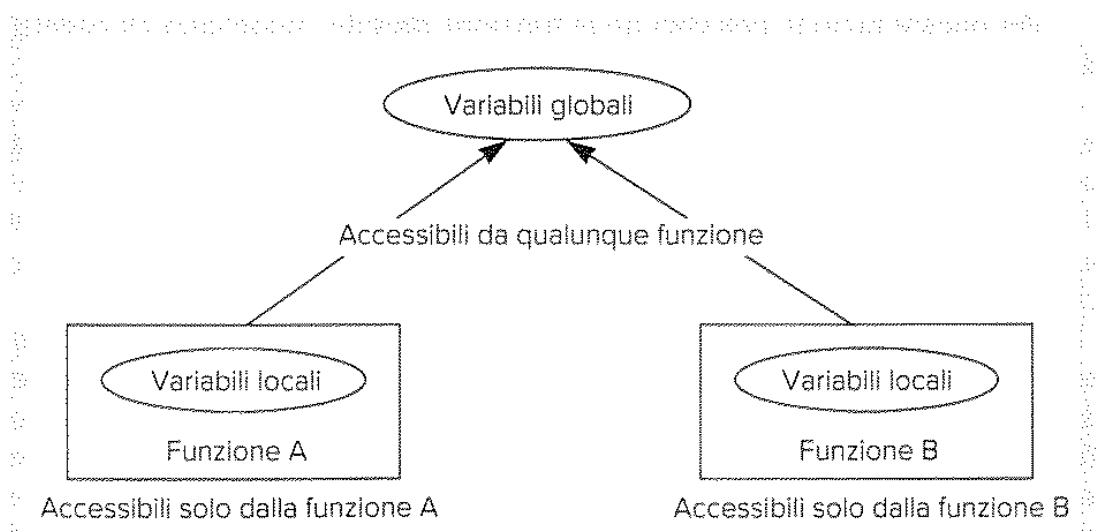


Figura 1.4

Dati locali e globali.

guenti difficoltà di comprensione e lettura; ma la cosa peggiore è che il programma risulta difficile da modificare perché i cambi nella struttura dei dati globali richiedono la riscrittura di tutte le funzioni che vi accedono, e non è detto che queste siano sempre possibili da operare.

La programmazione strutturata migliora la chiarezza, aumenta l'affidabilità e facilita l'aggiornamento dei programmi; tuttavia, per programmi grandi o su vasta scala, presenta sfide di difficile soluzione.

## 1.6 Programmazione orientata agli oggetti

Un secondo problema della programmazione strutturata sta nel fatto che la separazione fra dati e funzioni non corrisponde ai modelli delle cose del mondo reale così come siamo portati a concepirli noi esseri umani. Nel mondo fisico si ha a che fare con oggetti come persone, macchine o aerei. Questi oggetti non sono come i dati né come le funzioni. Complessi che siano o no, essi hanno *attributi* e *comportamenti*. Per esempio, gli **attributi** nelle persone sono la loro età, la loro professione, il loro domicilio ecc.; in una macchina, la potenza, il numero di matricola, il prezzo, il numero di porte ecc; in una casa, la superficie, il prezzo, l'anno di costruzione, l'indirizzo ecc. Gli attributi del mondo reale corrispondono ai dati di un programma; infatti essi hanno un valore specifico, come 200 metri quadrati, 20.000 dollari, cinque porte ecc. Il **comportamento** è invece l'azione che gli oggetti del mondo reale possono compiere in risposta a un determinato stimolo. Premendo il freno l'auto si ferma; premendo invece l'acceleratore essa aumenta la sua velocità ecc. Il comportamento corrisponde a una funzione: si manda in esecuzione una funzione (i programmati dicono "si chiama" una funzione) per fare qualcosa (mostrare a video l'elenco degli impiegati di un'impresa ecc.).

Per queste ragioni, né i dati né le funzioni, da sole, modellano gli oggetti del mondo reale in un modo efficiente.

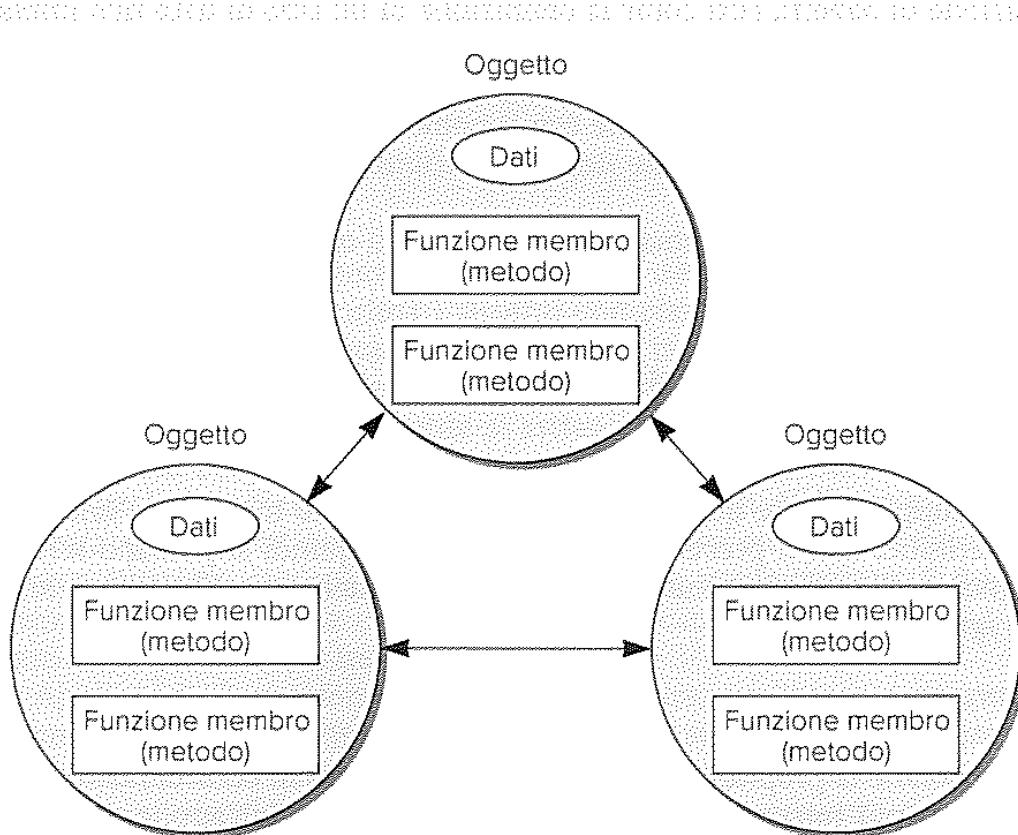
La **programmazione orientata agli oggetti** (*Object Oriented Programming*, OOP) è forse la metodologia di programmazione oggi più diffusa e ha portato alla definizione di specifici linguaggi di programmazione. L'idea fondamentale dei linguaggi orientati agli oggetti è di combinare in un unico modulo sia i dati sia le funzioni che operano su di essi. Tale modulo viene chiamato **oggetto**.

Il C++ nasce come evoluzione del C verso l'OOP. In esso le funzioni di un oggetto (i suoi "comportamenti") si chiamano *funzioni membro*, o *metodi* (nome derivato dallo Smalltalk, uno dei primi linguaggi orientati agli oggetti), e sono l'unico mezzo per accedere ai suoi dati. I dati di un oggetto si dicono invece *attributi*. Se si desidera leggere i dati di un oggetto, si chiama una funzione membro di quell'oggetto; questo metodo può anche cambiare il valore dei dati. Normalmente non si può accedere ai dati direttamente perché sono nascosti per essere protetti da alterazioni accidentali. Dati e funzioni sono cioè *incapsulati in un'unica entità*. L'*incapsulamento* e l'*occultamento* dei dati sono ter-

mini chiave nella trattazione dei linguaggi orientati agli oggetti. Per modificare i dati di un oggetto si usano le uniche funzioni predisposte per farlo. Questo semplifica la scrittura, la correzione (*debugging*) e l'aggiornamento dei programmi. Un programma C++ è costituito normalmente da un insieme di oggetti che comunicano tra loro tramite chiamata ad altre funzioni membro (Figura 1.5), cioè *invia*ndo un messaggio ad altri oggetti.

Nel paradigma orientato agli oggetti il programma è un insieme finito di oggetti che contengono dati e operazioni ( dette "funzioni membro" o "metodi" in C++) che possono lavorare quei dati. Gli oggetti comunicano tra di loro tramite messaggi.

L'oggetto è quindi il cuore dell'OOP: un'entità che gioca un ruolo definito nella risoluzione del problema, per cui bisogna individuare bene quali essi debbano essere. La loro struttura interna e il loro comportamento non hanno inizialmente molta importanza. Più importante è il ruolo che questi giocano nel sistema. Gli aspetti rilevanti di un oggetto dipendono dalla specifica applicazione in cui esso opererà. Un oggetto non deve necessariamente rappresentare qualcosa di concreto o tangibile. Per esempio, può descrivere un'entità astratta come un processo.



**Figura 1.5**

Organizzazione tipica di un programma orientato agli oggetti.

Nell'OOP un problema non si scomponete in funzioni, come nella tradizionale programmazione strutturata del linguaggio C, ma in oggetti. Pensare in termini di oggetti ha un grande vantaggio: si associano gli oggetti del problema alle entità del mondo reale. Non c'è limite oggettivo alla fantasia di cosa possa essere rappresentato da un oggetto. Categorie tipiche sono:

- risorse umane: impiegati, studenti, clienti, venditori, soci;
- strutture dati: vettori, liste, pile, alberi, grafi;
- tipi di dato definiti dagli utenti: ora, numeri complessi, punti dello spazio, figure geometriche;
- elementi di un computer: menù, finestre, mouse, tastiera, stampante, USB;
- oggetti fisici: auto, aerei, treni, navi, motociclette, case;
- componenti di videogiochi: console, comandi, volanti, connettori, memoria, accesso a Internet.

Lo stato di un oggetto viene determinato dai valori che assumono i suoi attributi.

Una **classe** è una descrizione di oggetti simili, che appartengono cioè alla medesima categoria di oggetti. Gli oggetti sono *istanze* di **classi**. Così come le *variabili* sono di un certo *tipo*, così gli *oggetti* sono di una certa *classe*, quindi definire una classe è come definire un tipo di dato, solo che essa contiene sia dati sia funzioni (metodi). La sua definizione non implica la creazione di oggetti, così come la definizione di un tipo di dato non implica la definizione in memoria di una variabile di quel tipo.

Una classe si rappresenta in **UML** (*Unified Modeling Language*) come una scatola che contiene una sezione con il nome della classe e, optionalmente, altre due sezioni con il nome dei suoi attributi e dei suoi metodi (Figura 1.6).

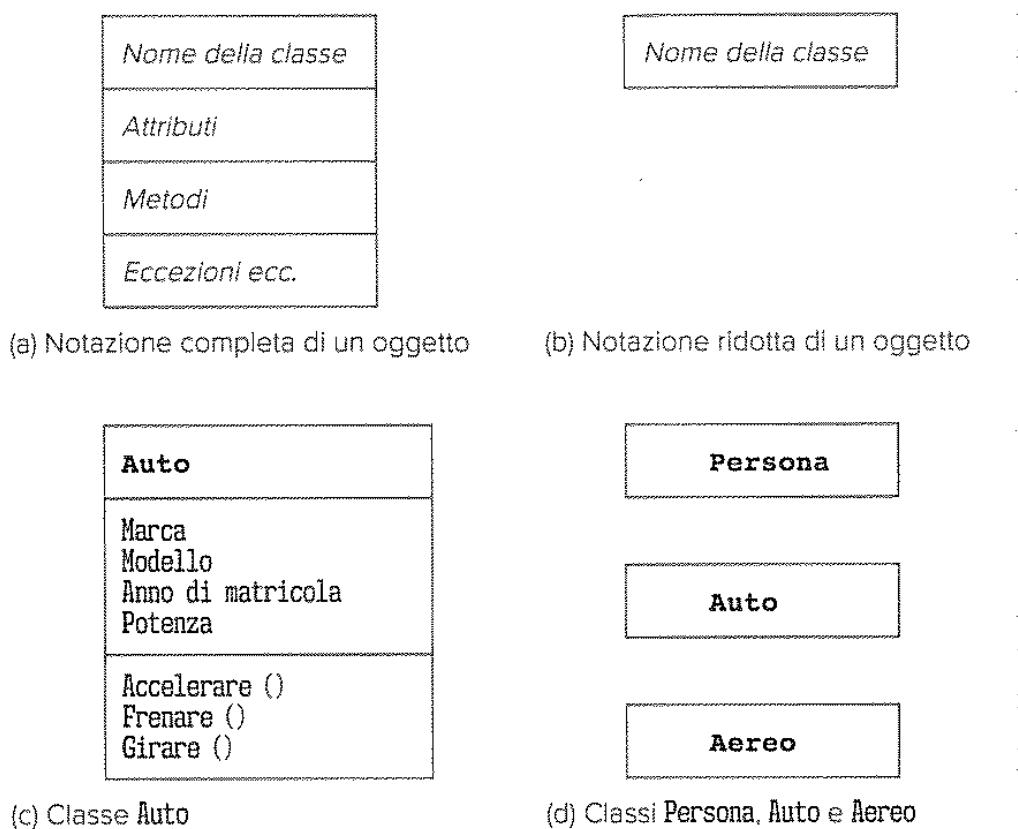
I linguaggi di programmazione orientati agli oggetti supportano queste quattro proprietà:

- **astrazione;**
- **incapsulamento e occultamento;**
- **ereditarietà;**
- **polimorfismo.**

Il C++ supporta tutte queste proprietà, ma ha voluto mantenere una retrocompatibilità con il precedente linguaggio C, e quindi si possono scrivere programmi C++ che non utilizzano le caratteristiche specificamente orientate agli oggetti.

#### 1.6.1 Astrazione

L'astrazione è la proprietà che consiste nel prendere in considerazione solo gli aspetti più importanti degli oggetti in funzione dell'applicazione che s'intende realizzare. Il termine si riferisce al fatto che si determinano le proprietà esterne di un'entità senza specificare i dettagli della sua composizione interna. È l'astrazione che permette di utilizzare dispositivi complessi, come un computer, un'automobile, una lavatrice o un forno a



**Figura 1.6**  
Rappresentazione di classi in UML.

microonde, senza conoscerne la struttura e i dettagli interni. Tramite l'*astrazione* si progettano prima i sistemi complessi, poi i componenti più piccoli di cui essi sono costituiti e così via, in maniera ricorsiva, fino ai componenti elementari. Ogni componente rappresenta un *livello di astrazione* in cui l'uso del componente si astrae dai dettagli della sua composizione interna. Pertanto, l'*astrazione* possiede diversi gradi che aiutano a strutturare la complessità intrinseca dei sistemi reali. Nella progettazione orientata agli oggetti questo significa concentrarsi su *cosa è* e *cosa fa* un oggetto piuttosto che sul *come* esso deve essere implementato. È durante il processo di astrazione che si decide quali caratteristiche e comportamenti deve possedere il modello.

Applicando l'*astrazione* si possono costruire, analizzare e progettare sistemi di computer complessi e grandi che non si potrebbero realizzare se si dovesse modellare direttamente al livello dettagliato. In ogni livello di astrazione si visualizza il sistema in termini di componenti, denominati **astratti**, la cui composizione interna si ignora.

Per gli oggetti è interessante non solo sapere *come* sono strutturati i loro dati, ma anche *cosa* si può fare con essi, cioè le operazioni che ne costituiscono l'anima. Il primo concetto dell'orientazione agli oggetti nac-

que con i *tipi di dato astratti* (*Abstract Data Type*, ADT). Un tipo di dato astratto descrive appunto non soltanto gli attributi di un oggetto, ma anche il suo comportamento (le operazioni) e corrisponde quindi al concetto di *classe*.

### 1.6.2 Incapsulamento e occultamento

L'*incapsulamento* consiste nell'aggregare dati e operazioni a essi connesse in *classi* di oggetti caratterizzati proprio da quei tipi di dato e quei comportamenti. L'*occultamento* che ne risulta consiste nel separare l'aspetto esterno di una classe di oggetti (detto *interfaccia* verso il mondo esterno) dai suoi dettagli implementativi interni, ovvero alcuni metodi saranno privati a quella classe, mentre altri saranno pubblici e ne costituiranno l'interfaccia visibile e utilizzabile dall'esterno della classe.

Lo sviluppo di un programma orientato agli oggetti consiste almeno nei seguenti passi.

1. Identificare gli *oggetti* del sistema.
2. Raggruppare in *classi* gli oggetti che hanno caratteristiche e comportamento comuni.
3. Identificare *dati* e *operazioni* di ognuna delle classi.
4. Identificare le *relazioni* che possono esistere tra le classi, ovvero in che modo una classe può interfacciarsi con le altre o con il programma principale.

Nell'OOP si parte cioè col progettare le classi che rappresentano le cose di cui tratta il programma. Per esempio, un programma per la grafica può definire classi che rappresentano rettangoli, linee, pennelli e colori, corredate con le relative operazioni, come spostamento di un cerchio, rotazione di una linea ecc. La progettazione di classi coerenti e utili può essere un compito difficile, ma i linguaggi OOP facilitano il compito al programmatore perché incorporano classi generiche predefinite. I compilatori di linguaggi OOP forniscono infatti numerose librerie di classi, incluse quelle per utilizzare i sistemi grafici e le finestre. Uno dei punti di forza del C++ è proprio la grande quantità di classi ben scritte ed efficienti dei suoi compilatori.

### 1.6.3 Ereditarietà

Le classi possono essere ordinate in una gerarchia tale per cui quelle inferiori possono condividere attributi e comportamenti di quelle superiori. Per esempio, lavatrici, frigoriferi, forni a microonde, tostapani, lavastoviglie ecc., sono tutti elettrodomestici, quindi nell'OOP ognuno di questi dispositivi può essere modellato come una **sottoclasse** della classe elettrodomestico e, a sua volta, elettrodomestico è una **superclasse** delle classi lavatrice, frigorifero, forno\_a\_microonde, tostapane, lavastoviglie ecc. La classe animale ha come sottoclassi anfibi, mammiferi, insetti, uccelli ecc. (Figura 1.7), e la classe veicolo ha auto, moto, camion, autobus ecc.

Il processo che muove dalle classi basse verso quelle alte si chiama *generalizzazione*, quello che muove dalle classi alte verso quelle basse si chiama *specializzazione*, e a questa transizione si associa il meccanismo esplicito della

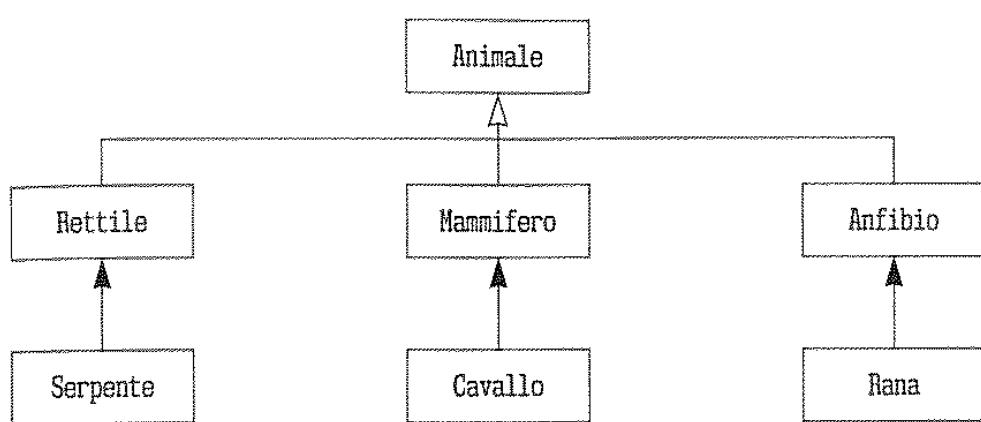


Figura 1.7

Ereditarietà di classi in UML. Il diagramma mostra la gerarchia di classi con le loro sottoclassi e i relativi esemplificatori.

**ereditarietà** che permette di definire nuove classi a partire da altre più generiche aggiungendo caratteristiche specifiche. Il principio è che ogni sottoclasse condivide caratteristiche comuni con la superclasse da cui deriva, ma si distingue dalle altre sottoclassi della medesima superclasse per via di proprie caratteristiche specifiche. In C++ la superclasse si denomina *classe base* e le altre *classi derivate*.

#### 1.6.4 Riusabilità

Una volta che una classe è stata scritta e validata si può distribuire perché altri programmatore la possano utilizzare nei propri programmi; questa proprietà si chiama *riusabilità*. Il concetto è simile a quello delle funzioni di libreria di un linguaggio procedurale come il C, che possono essere utilizzate in altri programmi, ma in C++ l'ereditarietà estende il concetto di *riusabilità* poiché il programmatore può riutilizzare una classe esistente derivandone una nuova mediante l'aggiunta di attributi e metodi.

L'aumentata possibilità di riutilizzare software esistenti è un altro beneficio dell'OOP in quanto, grazie alla riusabilità di classi in nuovi progetti, le imprese riescono a ridurre i costi di gestione e sviluppo dei programmi.

In pratica, la riusabilità ha due vantaggi: la consistente riduzione del codice (le proprietà comuni di diverse classi devono essere implementate una sola volta) e la facilità di aggiornamento (le proprietà comuni devono essere modificate una volta sola, se necessario).

#### 1.6.5 Polimorfismo

L'ereditarietà comporta anche un altro vantaggio: facilita il **polimorfismo**, ovvero la proprietà di una funzione di comportarsi in modo differente in funzione dell'oggetto su cui si applica. In pratica il polimorfismo sta nella capacità di un metodo di essere interpretato correttamente soltanto dall'oggetto che l'invoca. Con il polimorfismo una funzione può avere lo stesso nome in diffe-

renti classi, ma eseguire differenti operazioni. Così, per esempio, la funzione di "aprire" si può definire in differenti classi: aprire una porta, aprire una finestra, aprire un giornale, aprire un archivio, aprire un conto in banca, aprire un libro ecc., ma consiste sempre in un'operazione diversa. Un altro esempio significativo è quello degli operatori "+" e "\*" applicati a numeri interi e a numeri complessi; le due funzioni matematiche possono conservare il nome, ma nel primo caso la somma e la moltiplicazione sono operazioni semplici, mentre nel caso dei numeri complessi (caratterizzati da parte reale e parte immaginaria) sarà necessario seguire un altro metodo specifico e ottenere un risultato che sarà pure un numero complesso. Quando si permette a un operatore di applicarsi a differenti tipi di dato con funzionalità diverse, si dice che l'operatore è *sovraffatto* (*overloaded*).

## 1.7 Il sistema operativo

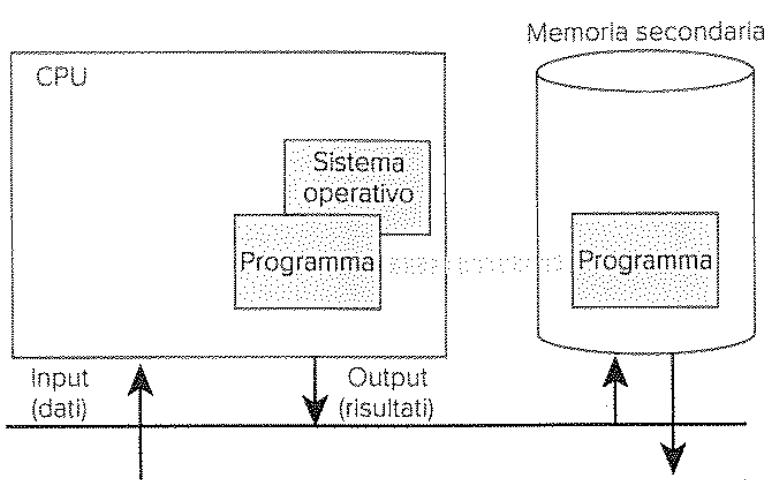
I programmi installati su un computer ne costituiscono il *software*. In base alla destinazione d'uso esistono due tipi di software: quello "di sistema" e quello "applicativo". Il **software di sistema** è l'insieme di programmi che gestisce le risorse stesse del computer, come il processore, la memoria e le periferiche, mentre il **software applicativo** è l'insieme di programmi scritti per eseguire compiti specifici esterni al sistema di elaborazione stesso. I due tipi di software sono ovviamente correlati, perché il software applicativo esterno utilizza l'hardware per mezzo dello strato interno costituito dal software di sistema. Quindi, in generale, sostituendo il software di sistema, quello applicativo non gira più. Il software di sistema, più comunemente noto come **sistema operativo**, coordina quindi le differenti funzioni del sistema di elaborazione e gestisce l'hardware del computer per conto del software applicativo. Il sistema operativo viene spesso fornito con un corredo di programmi di utilità (*utilities*) i più importanti dei quali sono i *compilatori*, che traducono i programmi scritti in linguaggio ad alto livello (come il C++) in linguaggio macchina (quello binario eseguibile dalla CPU), e gli *editori* di file di testo.

L'utente del sistema comunica con il sistema operativo attraverso l'interfaccia utente. I sistemi operativi moderni utilizzano interfacce utenti grafiche (*Graphical User Interface, GUI*) che fanno uso di finestre, menù, icone e bottoni che si controllano dalla tastiera, dal mouse o altri dispositivi come i *joystick*.

Normalmente, il sistema operativo è installato sul disco fisso e viene caricato nella RAM da parte di un piccolo programma, il già menzionato BIOS, residente nella ROM e attivato all'accensione del computer. Questo processo viene detto *bootstrap*.

La Figura 1.8 mostra il processo generale di esecuzione di un programma, dal suo caricamento nella CPU da parte del sistema operativo, all'ottenimento dei dati di input, dalla corrispondente periferica fino all'ottenimento dei risultati in uscita sulla periferica di output. Sia l'input sia l'output possono essere testi, numeri, immagini, suoni, filmati o segnali per controllare apparecchiature digitali. Esistono compilatori C++ per tutti i sistemi operativi.

I sistemi operativi possono possedere le seguenti importanti caratteristiche.



**Figura 1.8** Esecuzione di un programma. Perché il programma non ha bisogno della memoria principale?

### Multitasking

Il multitasking permette a più programmi di condividere le risorse di un computer, in particolare della sua CPU. Mentre un programma utilizza la CPU, gli altri possono attendere che si compiano le loro operazioni di I/O. Dopo un certo quanto di tempo (*time sharing*) la CPU viene consegnata a un altro programma e così via, ciclicamente per tutti.

### Multiutenza

Un sistema operativo multiutente può permettere a parecchi utenti di condividere contemporaneamente le risorse del computer. Centinaia o migliaia di utenti si possono collegare al computer che assegna un ambiente di lavoro e un certo quanto di tempo ai processi di ciascuno di loro, in maniera che, data l'alta velocità della CPU, la sensazione di ognuno è quella di avere un computer dedicato.

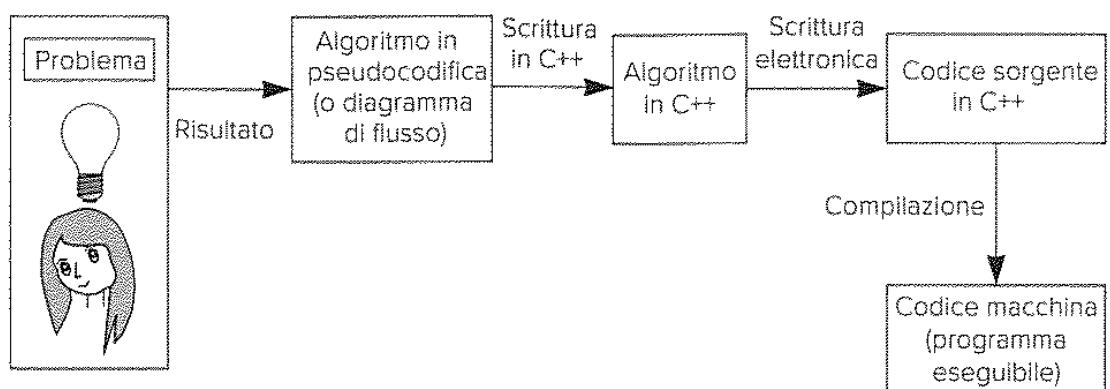
### Multiprocessore

Un sistema operativo è multiprocessore quando può utilizzare più CPU contemporaneamente per farle lavorare in parallelo. Il sistema operativo può assegnare a ogni CPU differenti istruzioni dello stesso programma o programmi differenti simultaneamente.

## 1.8 Linguaggi di programmazione

Affinché la CPU esegua un programma essa deve essere capace di *interpretarlo*, il che significa:

- capire le istruzioni a ogni passo;
- realizzare le operazioni corrispondenti.

**Figura 1.9**

Processo di trasformazione di un algoritmo dallo pseudocodice al programma eseguibile.

Oggi i programmi si scrivono in linguaggi di programmazione **ad alto livello** (come il C++), e i relativi programmi si dice siano in **codice sorgente**. Il codice sorgente è comprensibile ai programmatore ma non alla CPU che è capace solo di eseguire istruzioni scritte nel suo **linguaggio macchina**. Quest'ultimo non è invece facilmente comprensibile al programmatore. Per questo sono necessari programmi **traduttori** che traducano il codice sorgente in codice macchina. Tutto il processo di conversione di un algoritmo scritto in pseudocodice fino al programma eseguibile è mostrato nella Figura 1.9.

I più diffusi linguaggi di alto livello sono il C/C++, C#, Java, Visual Basic, XML, HTML, Perl, PHP, JavaScript..., anche se si utilizzano ancora i classici COBOL, FORTRAN, Pascal e BASIC.

I programmi traduttori si dividono in **compilatori** e **interpreti**.

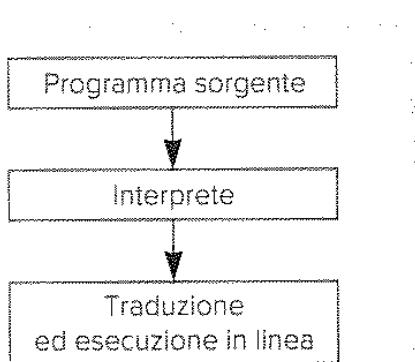
### Interpreti

Un *interprete* (Figura 1.10) traduce ed esegue un'istruzione sorgente alla volta dalla prima all'ultima nel programma. BASIC e Smalltalk erano esempi di linguaggi interpretati. Anche Java può essere quasi considerato un linguaggio interpretato.

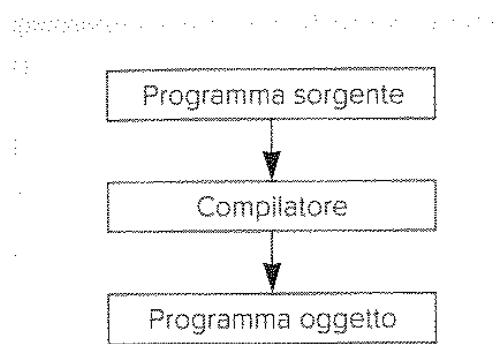
### Compilatori

Un *compilatore* traduce *tutto* il programma sorgente in un programma eseguibile. L'operazione si chiama **compilazione** del programma (Figura 1.11). Il programma compilato e corretto (dopo vari cicli di compilazione, testing, correzione del sorgente e ricompilazione) si chiama *programma eseguibile*. Ogni linguaggio ad alto livello deve avere il suo compilatore, anzi, deve avere un compilatore per ogni CPU e ogni sistema operativo sui quali voglia essere utilizzato.

Il processo che porta dal sorgente all'eseguibile integra normalmente anche una fase chiamata *di linking* (collegamento). Questo perché un pro-



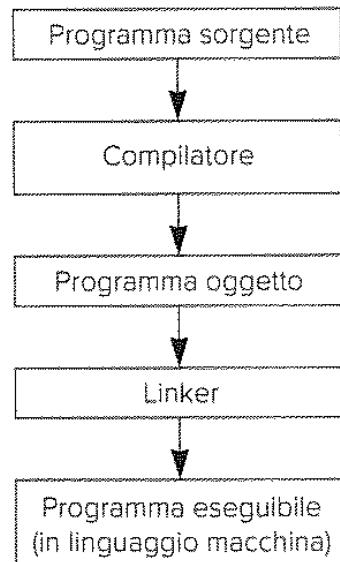
**Figura 1.10** La traduzione di un programma sorgente da parte di un interprete.



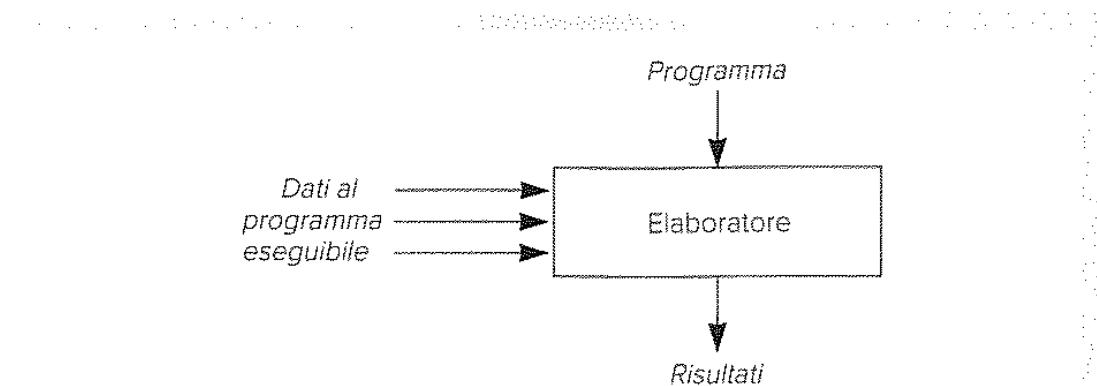
**Figura 1.11** La compilazione di programmi.

gramma fa sempre uso di librerie (almeno per gestire l'I/O) che devono essere collegate al programma. Tutto il processo è schematizzato nella Figura 1.12. Esso si compone di solito dei seguenti passi.

1. Scrivere il *programma sorgente* con un *text editor* e salvarlo sulla memoria di massa.
2. Caricare il sorgente in memoria.
3. *Compilare* il programma sorgente.
4. *Verificare e correggere gli errori di compilazione*.
5. Ottenere il *programma oggetto*.
6. *Linkare il programma oggetto alle librerie per ottenere il programma eseguibile*.
7. Eseguire il programma eseguibile.



**Figura 1.12** Fasi della compilazione.



**Figura 1.13**  
Esecuzione di un programma.

Il processo è mostrato nella Figura 1.13. Nel Capitolo 2 si descriverà in dettaglio il processo per il linguaggio C.

## 1.9 Il linguaggio C

Benché il linguaggio C nasca per e con il sistema operativo UNIX, oggi esso non è praticamente associato ad alcun sistema operativo, né ad alcuna macchina in particolare. Anzi, quasi tutti i sistemi operativi sono scritti in C.

C è un'evoluzione dei linguaggi BCPL – sviluppato da Martin Richards – e B – sviluppato da Ken Thompson nel 1970 – per il primitivo UNIX del computer DEC PDP-7.

C nacque nel 1978 con la pubblicazione di *The C Programming Language*, di Brian Kernighan e Dennis Ritchie (Prentice Hall, 1978). Da allora ebbe una crescita tumultuosa e apparvero subito varie personalizzazioni e compilatori sviluppati da terze parti non coinvolte nel progetto originale. Per questo si rese presto necessaria la sua standardizzazione.

Nel 1983, l'*American National Standard Institute* (ANSI) creò un comitato (denominato X3J11) per dare “*una definizione del linguaggio C non ambigua e indipendente dalla macchina*”. Nacque così lo standard ANSI del linguaggio C. Esso assicura che qualunque compilatore ANSI C incorpori tutte le caratteristiche del linguaggio specificate dallo standard. Questo significa che i programmatore che scriveranno secondo lo standard C avranno la sicurezza che i programmi gireranno su qualunque sistema che abbia un compilatore ANSI C.

C è un linguaggio *general purpose* per la programmazione strutturata, ma, essendo nato per scrivere un sistema operativo, fu pure immediatamente caratterizzato da una assoluta potenza (capacità di interazione con l’hardware) e flessibilità.

L’ANSI C introduceva elementi e funzionalità non presenti nella prima versione di C, come l’assegnamento di strutture ed enumerazioni. Inoltre si definì una nuova forma di dichiarazione di funzioni (i prototipi). Ma la standardizzazione più importante fu forse quella della grande libreria di funzioni.

Oggi C è ancora uno dei linguaggi di programmazione più utilizzati nell'industria del software e nelle università. Praticamente tutti i sistemi operativi, Windows, UNIX, Linux, MacOS, Solaris..., supportano diversi tipi di compilatori C, spesso gratuiti. Tutti i compilatori C++ possono eseguire programmi scritti in ANSI C.

## 1.10 Il linguaggio C++

C++, Java e C# derivano dal linguaggio C estendendolo con caratteristiche orientate agli oggetti e a Internet. Con il C essi formano attualmente il *poker* di linguaggi più impiegati al mondo.

Benché C sia un linguaggio molto potente, ha due caratteristiche che lo rendono inappropriato come introduzione alla moderna programmazione. Innanzitutto C è ritenuto di difficile comprensione dai programmatore principianti. Inoltre C fu pensato agli inizi degli anni settanta, ma la programmazione è cambiata molto negli anni ottanta e novanta.

Bjarne Stroustrup, dell'AT&T Bell Laboratories, sviluppò C++ agli inizi degli anni ottanta. Ispirandosi anche al Simula 67 [Dahl, 1972] (lo segnala lui stesso [Stroustrup, 1997]), Stroustrup disegnò C++ come un C con caratteristiche di OOP. In generale, C standard è un sottoinsieme di C++ e la maggioranza dei programmi C sono anche programmi C++, ma non il contrario.

### **IMPORTANTE: Sito ufficiale di Bjarne Stroustrup**

Bjarne Stroustrup, progettista e implementatore del C++, è il riferimento principale per lo studente e il programmatore di C++. Le sue opere *The C++ Programming Language*, *The Design and Evolution of C++* y *C++ Reference Manual* sono letture obbligate.

Il suo personale sito web su AT&T Labs Researchs deve essere fra i preferiti del lettore.

<https://www.stroustrup.com/>

Il sito è aggiornato frequentemente, contiene una grande quantità di informazioni e una eccellente sezione di FAQ (*Frequently Asked Questions*).

Sono state presentate diverse versioni di C++ e la sua evoluzione è stata studiata in [Stroustrup 94]. Le caratteristiche più importanti incorporate nel C++ sono: classi, ereditarietà multipla, genericità, funzioni virtuali, *template* ed eccezioni. Come il suo autore ha spiegato, C++ si evolve di anno in anno “sempre per risolvere problemi trovati dagli utenti e come conseguenza di conversazioni tra l'autore, i suoi amici e i suoi colleghi”.

Nel dicembre del 1989, su iniziativa di Hewlett Packard, si riunì il comitato X3J16 dell'ANSI e cominciò anche per C++ il processo di standardizzazione (*The Annotated C++ Reference Manual* [Ellis 89]). Nel giugno del 1991 all'ANSI si unì ISO (International Organization for Standardization) con il suo

proprio comitato (ISO-WG-21), creando una task force comune ANSI/ISO per sviluppare uno standard per C++. Questi comitati si riunirono tre volte l'anno per arrivare a uno standard che facesse di C++ un linguaggio importante e di ampia diffusione.

Nel 1995 il comitato pubblicò diversi *working paper* e nell'aprile di quell'anno fu pubblicato un libro bianco sullo standard. Nel dicembre del 1996 se ne rilasciò una seconda versione (CD2). Questi documenti non solo affinarono la descrizione delle caratteristiche di C++, ma ampliarono pure il linguaggio con eccezioni, identificazione a *run-time* (*Run-Time Type Identification*, RTTI), template e la STL (*Standard Template Library*). Stroustrup pubblicò nel 1997 la terza edizione del suo libro *The C++ Programming Language*. Nel 1998 il comitato produsse lo standard ISO/IEC 14882: 1998 [Standard98]. Nel 2003 vide la luce una seconda versione dello Standard [Standard03] disponibile su internet come file PDF su <http://www.ansi.org>. La nuova edizione è una revisione tecnica, cioè riordina l'edizione precedente – fissazione dei tipi, riduzione delle ambiguità e simil –, ma non cambia le caratteristiche del linguaggio.

Poiché ci vuole molto tempo perché i compilatori recepiscono le ultime specifiche di un linguaggio, esistono compilatori che ancora non sono conformi allo standard. Per cercare di evitare problemi, tutti i programmi di questo libro sono stati compilati con le ultime versioni di diversi compilatori.

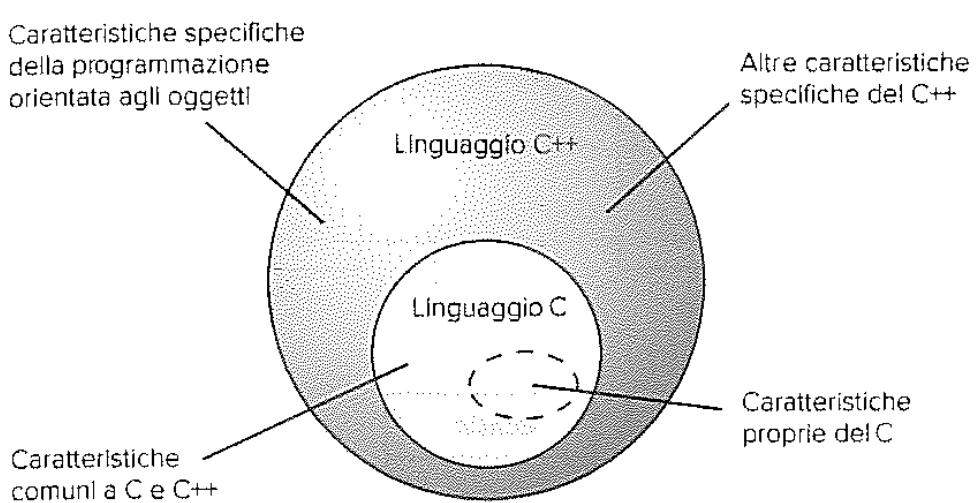
Benché C++ sia un linguaggio orientato agli oggetti, esso può essere utilizzato come linguaggio procedurale stile C se si vuole lavorare secondo la classica programmazione strutturata, concentrandosi su algoritmi e strutture di dati.

### 1.11 C versus C++

C++ è un superinsieme di C. Ha ovviamente una sua libreria di classi, ma utilizza anche la libreria standard ANSI C. Perciò quasi tutte le istruzioni C sono accettate in C++ ma non il contrario. Gli elementi più importanti aggiunti sono legati al concetto di classe e alla OOP (C++ fu chiamato all'inizio "C con classi"). Tuttavia, sono state aggiunte anche altre caratteristiche come una migliore impostazione dell'I/O. La Figura 1.14 mostra le relazioni tra C e C++.

Anche se si può scrivere un programma in C++ simile a un programma in C, raramente si fa. C++ fornisce ai programmati nuove potenzialità e spinge a utilizzarle, ma recepisce in parte anche caratteristiche specifiche e potenti del C. Chi conosce già C troverà familiari molte proprietà che imparerà presto e bene, ma troverà pure parecchie proprietà nuove e molto potenti che gli faranno apprezzare questo nuovo linguaggio approfittando della potenza dell'OOP.

Il nostro obiettivo è di aiutare a scrivere programmi OOP prima che sia possibile. Tuttavia, poiché il C++ deriva dal C, bisogna possedere una buona familiarità con il "vecchio stile *procedurale*", perciò i primi capitoli del libro vertono su di essa.

**Figura 1.14**

Caratteristiche di C e C++.

Questo libro descrive lo standard ANSI/ISO C++ seconda edizione, (ISO/IEC 14882: 2003) e gli esempi funzioneranno con qualunque compilatore C++ compatibile con esso. Tuttavia, malgrado gli anni trascorsi, C++ Standard si considera ancora nuovo e vi possono essere discrepanze con vecchi compilatori ancora in uso.

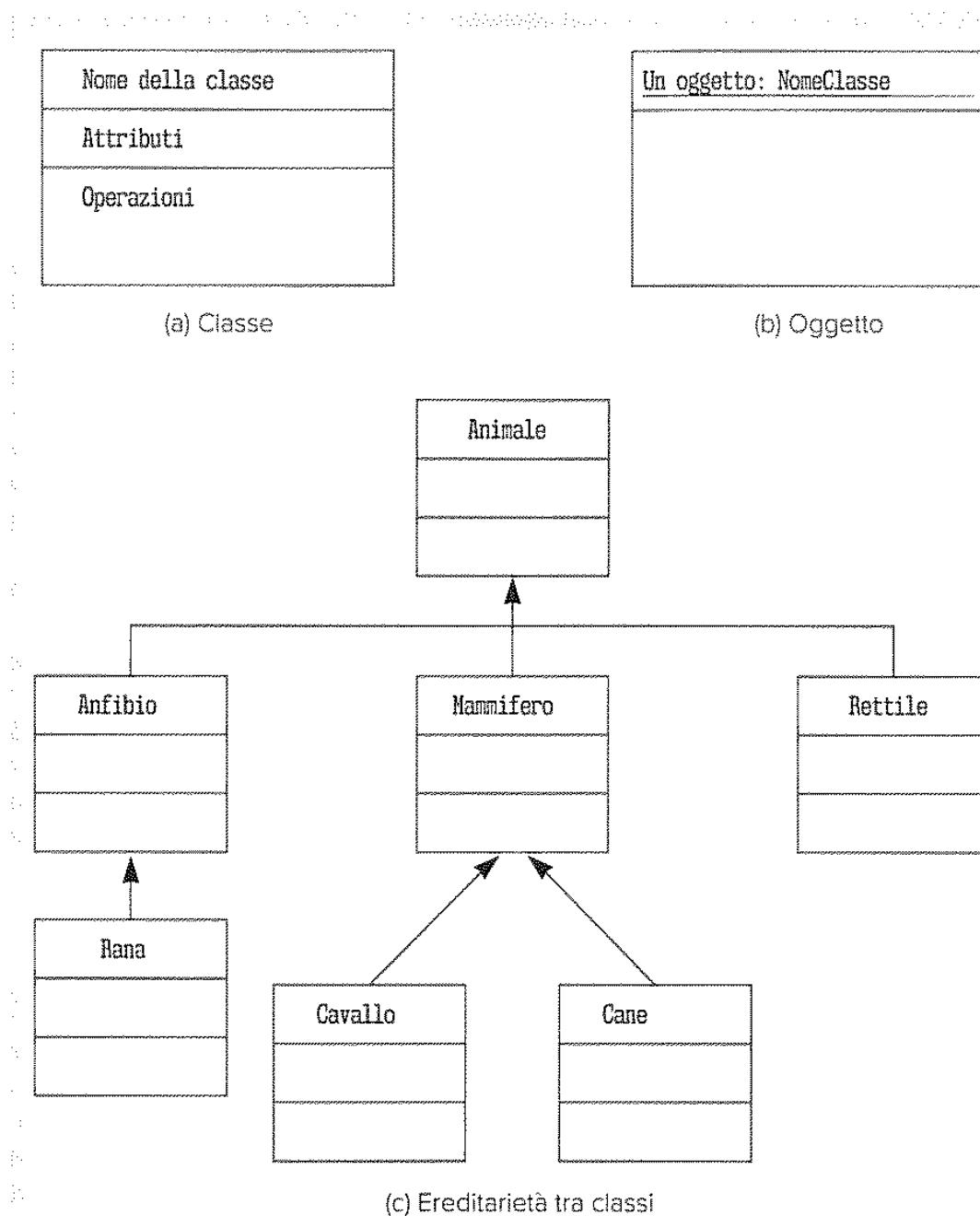
## 1.12 Il linguaggio di modellazione unificato (UML 2.0)

UML è un linguaggio grafico per la modellazione di programmi. Esso fornisce un mezzo per visualizzare graficamente l'organizzazione astratta dei programmi senza scendere nei dettagli del codice reale.

UML nasce come strumento grafico e metodologico per l'analisi di problemi e la sintesi di programmi OOP. In programmi di grandi dimensioni è infatti difficile capire il funzionamento esaminando solo il codice; strumenti come UML sono praticamente indispensabili per vedere come si relazionano le parti, l'una con le altre.

UML possiede un insieme completo di diagrammi grafici: diagrammi di classe che mostrano le relazioni tra classi, diagrammi di oggetti che mostrano come si relazionano oggetti specifici tra di loro, diagrammi di casi d'uso che mostrano come gli utenti di un programma interagiscono con il programma, ecc. UML è difatti lo standard per le rappresentazioni grafiche in OOP. Non è una metodologia di sviluppo del software ma, semplicemente, un mezzo per esaminare il software che si sta sviluppando. Nella Figura 1.15 si mostrano rappresentazioni grafiche di classi, oggetti e relazioni di generalizzazione e specializzazione (ereditarietà) tra classi.

Il linguaggio di modellazione si dice "unificato" perché è basato su diversi modelli precedenti. Nel 1994, James Rumbaugh e Grady Booch decidono



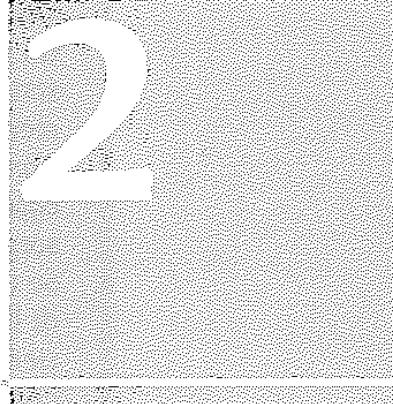
**Figura 1.15**  
Rappresentazioni grafiche di oggetti, classi ed ereditarietà in UML 2.0.

infatti di unificare le loro notazioni OMT e Booch. Nel 1995, Juan Jacobson si uni alla squadra dei «tre amici» per fondare la Rational Software. Nel 1997 viene pubblicato UML 1.0 e in quello stesso anno OMG (Object Management Group - un consorzio di più di 1.000 società e università attive nel campo delle tecnologie orientate agli oggetti) adotta la notazione e crea una commissione incaricata di studiare l'evoluzione di UML. Nel 2003 viene presentato UML 1.5, e nel 2005 UML 2.0. Nel 2006 Rumbaugh, Booch e Jacobson pubblicano le nuove edizioni della Guida Utente e del Manuale di Riferimento.

### Definizioni

- Hardware
- Unità centrale di processo
- Microprocessore
- Memoria ausiliare
- Memoria centrale
- Software
- Algoritmo
- Diagramma di flusso
- Linguaggio di programmazione
- Linguaggio macchina
- Compilatore
- Interprete

# Il linguaggio C++. Elementi base



<b>2.1</b>	Costruzione di un programma in C++	<b>2.5</b>	Tipi di dato predefiniti
<b>2.2</b>	Struttura generale di un programma C++	<b>2.6</b>	Variabili
<b>2.3</b>	Debugging di un programma in C++	<b>2.7</b>	Durata e visibilità di una variabile
<b>2.4</b>	Elementi di un programma in C++	<b>2.8</b>	Istruzione di assegnamento
		<b>2.9</b>	Costanti
		<b>2.10</b>	Input/output da console

## Introduzione

In questo capitolo iniziamo a conoscere i fondamenti del linguaggio di programmazione C++. Vedremo come creare un programma, quali sono i tipi di dato predefiniti del C++ e come si dichiarano. Poi vedremo cosa sono e come si dichia-

rano le costanti e, soprattutto, le variabili, con la loro visibilità e i loro tempi di vita. Vedremo anche come si leggono dati in ingresso al programma (input) e come si restituiscono i risultati in uscita (output).

### 2.1 Costruzione di un programma in C++

C++ è un linguaggio di programmazione di alto livello e i suoi programmi sono scritti in ASCII, mentre i computer, come il lettore già sa, eseguono solo programmi in linguaggio *macchina* (sequenze di istruzioni espresse in codice binario). Pertanto è necessario tradurre il codice sorgente in codice macchina (ovvero in codice binario). Questo processo di traduzione è detto *compilazione*.

I programmi nascono come idee nella mente del programmatore, che poi le sviluppa fino ad abbozzare un *algoritmo* espresso in linguaggio naturale, in uno *pseudocodice*, o direttamente in un linguaggio di programmazione, come il C++.

Una volta che è stato scritto il codice del programma si deve procedere alla sua esecuzione. Le tappe da seguire dipendono dal compilatore e dall'ambiente di sviluppo utilizzati; comunque, saranno simili alle seguenti.

1. Utilizzare un *editor di testo* per scrivere il programma sorgente e salvarlo in un file chiamato *file sorgente* o *codice sorgente*. Per convenzione, i programmi scritti in C++ hanno l'estensione .cpp, mentre quelli scritti in C hanno l'estensione .c.
2. *Compilare il codice sorgente*. Il compilatore traduce il codice sorgente in *codice oggetto*. In C++, il file prodotto in questo passo ha lo stesso nome di quello sorgente ma estensione .obj oppure .o.
3. Se il passo 2 ha avuto successo, il *linker* collega al codice oggetto le funzioni di libreria C++ di cui il programma fa uso. La libreria standard del C++ contiene il codice oggetto di svariate funzioni che realizzano compiti comuni, come la gestione dell'Input/Ouput (I/O) o calcoli matematici. Il collegamento produce una versione eseguibile del programma che viene posta in un file, il quale ha generalmente il nome del sorgente ed estensione .exe sotto Windows e nulla sotto Linux.

La Figura 2.1 riassume queste fasi. Normalmente il compilatore compie sia la compilazione vera e propria sia il collegamento. Per effettuare questi passaggi risultano molto comodi gli **ambienti di sviluppo integrati (IDE - Integrated Development Environment)** che incorporano un editore di testi, un compilatore, un linker, un gestore di progetti, un "debugger" e altri programmi di utilità.

Esistono vari compilatori C++ per ogni sistema operativo, così come esistono molti IDE e non avrebbe senso riportare una graduatoria di compilatori e IDE anche perché essi variano nel tempo e la classifica non risulterebbe certamente aggiornata al momento in cui viene letto questo libro. Quello che importa è che il lettore trovi presto quelli che fanno al caso suo e inizi subito a sperimentarli con i programmi che seguono.

#### 2.1.1 Esempio pratico di compilazione

1. Scrivere il seguente programma sorgente come file di testo (sequenza di caratteri ASCII) utilizzando un text editor, e dargli il nome ciao.cpp.

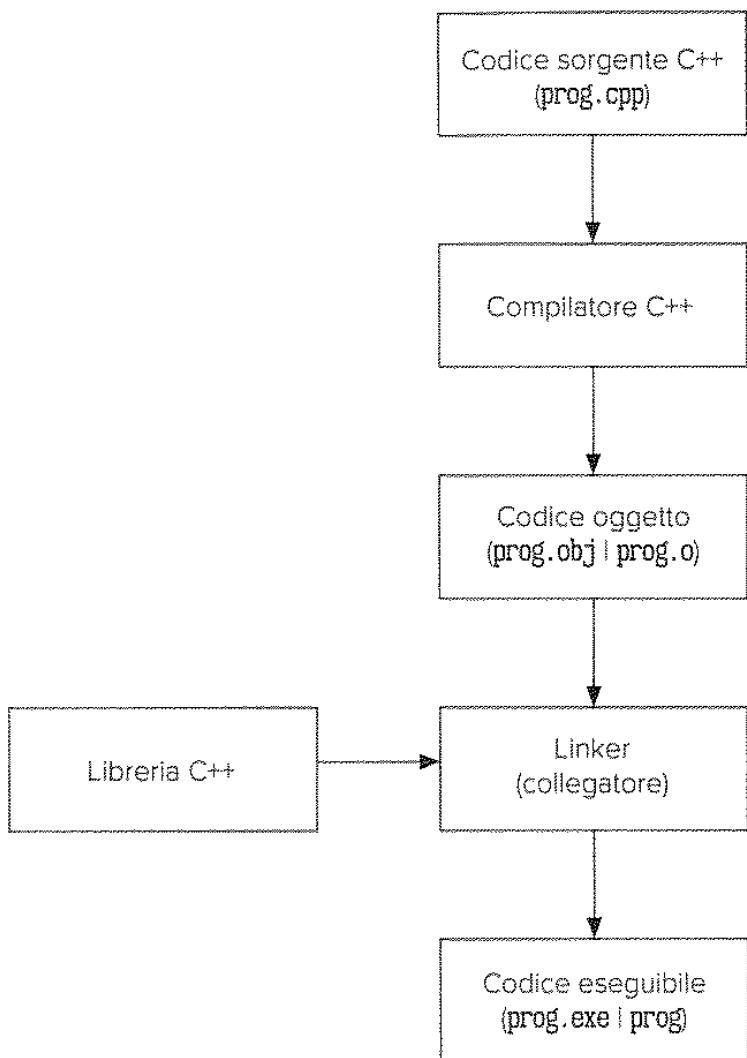
```
#include <iostream>

int main()
{
    std::cout << "Ciao mondo!\n";
    return 0;
}
```

2. Compilare il programma con un compilatore C++.



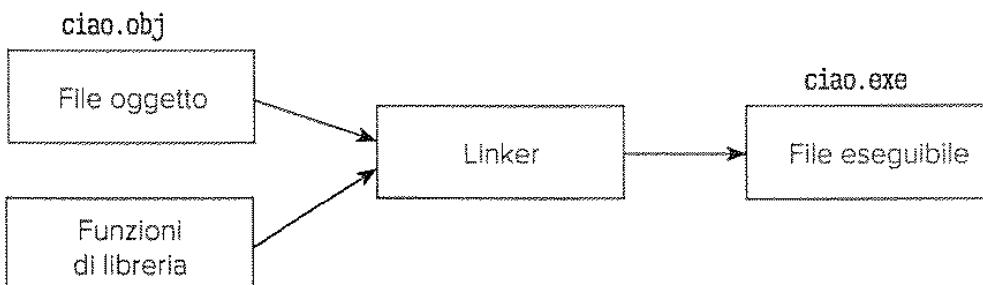
Se per esempio il nostro sistema operativo è Linux e si usa il compilatore "open source" GNU C++ della Free Software Foundation, aperto un terminale nella stessa cartella dove si trova il file ciao.cpp possiamo digitare:

**Figura 2.1**

Tappe nell'esecuzione di un programma.

```
% g++ -o ciao ciao.cpp
```

Il compilatore g++ creerà un file eseguibile denominato ciao (in Linux di solito non si aggiunge l'estensione .exe).



3. Eseguire il programma.

Se si esegue il programma dalla riga di comando della console di Linux digitando il comando

ciao

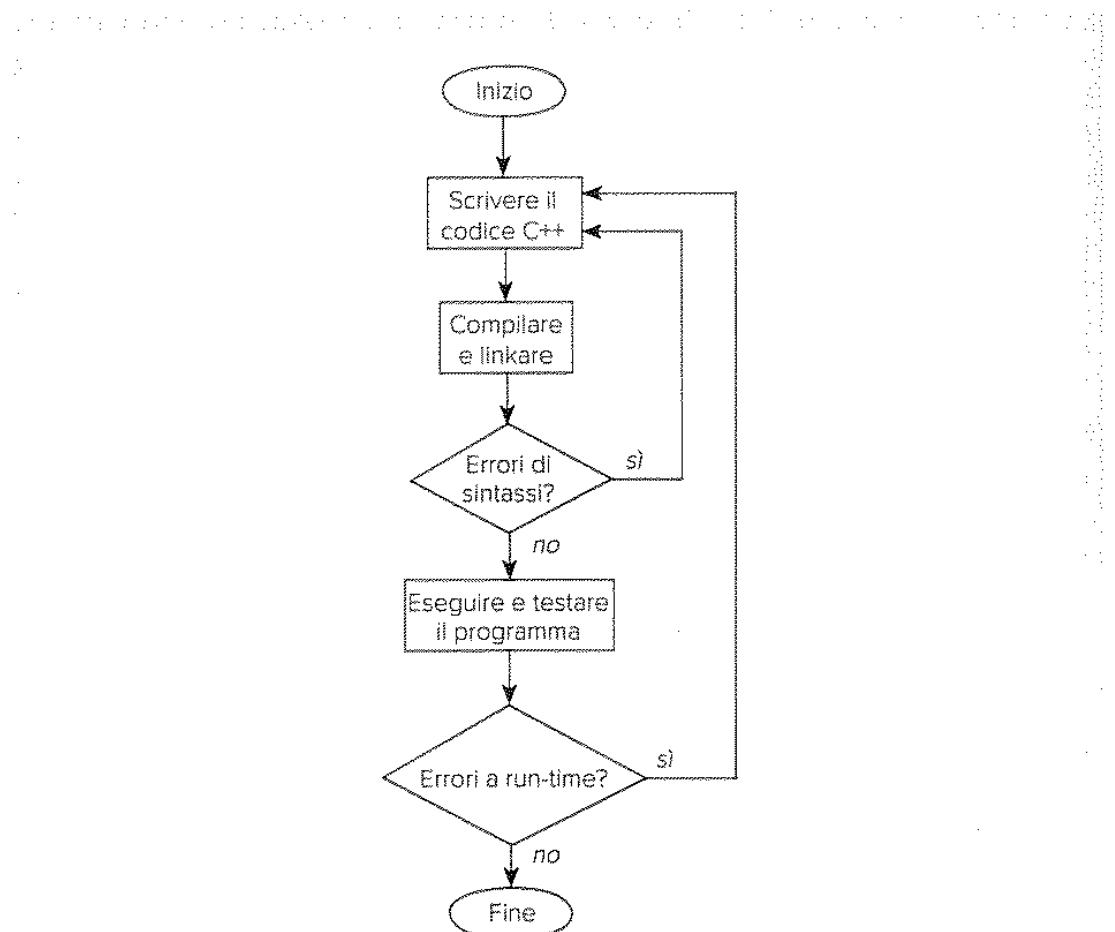
sullo schermo apparirà il messaggio:

Ciao mondo

### 2.6.2 Sviluppo di un programma in C++

Il processo completo di creazione di un programma eseguibile (Figura 2.2) implica le seguenti tappe.

1. Scrittura del codice sorgente sul computer (salvare con un nome; per esempio, demo.cpp).
2. Compilazione (*compiler*) del codice sorgente (demo.obj).



**Figura 2.2**  
Messa a punto di un programma C++.

3. Collegare (*linker*) il codice oggetto prodotto con quello delle librerie dichiarate nel programma sorgente (normalmente questo passo è svolto insieme a quello precedente).
4. Il compilatore/collegatore individua eventuali errori di sintassi che devono essere corretti, quindi si torna al punto 1.
5. Collaudo del programma. Se il programma non fa quello che il programmatore si aspettava si torna al punto 1.

Al punto 4, se il codice sorgente non contiene errori la compilazione termina con successo, altrimenti il compilatore darà informazioni circa le righe esatte che contengono gli errori.

Tuttavia, anche se la fase di compilazione non rileva errori, può darsi (anzi, è probabile) che l'eseguibile non si comporti correttamente come desiderato. Dopo che un programma è stato compilato con successo, bisogna quindi eseguire il programma dando input diversi per verificare che faccia esattamente quello che è stato progettato per fare. Nel caso di programmi commerciali o professionali si richiede maggiore qualità e si deve provare il programma parecchie volte per avere la *quasi* sicurezza che funzioni correttamente. Gli errori riscontrati in fase di collaudo (punto 5) si dicono *errori di esecuzione* o *logici*. In questi casi il programma è scritto sintatticamente bene ma non fa quello che dovrebbe fare a causa di errori nella logica dei passi di esecuzione. Il processo di collaudo (*testing*) di un programma con l'individuazione e la riparazione della causa del problema si chiama *debugging*.

Se esistono errori logici nel programma, bisogna determinarne la natura, andare indietro, revisionare il codice sorgente e in molti casi ricostruire il programma. Se i programmi sono semplici, magari con una quantità moderata di collaudi e riparazioni di errori, si termina presto. Ma se i programmi sono complessi, può essere necessario molto tempo per analisi, esecuzioni e collaudi. In questi casi si dovrà avere pazienza e seguire un metodo sistematico per la messa a punto del programma.

## 2.2 Struttura generale di un programma C++

Un programma in C++ è costituito da una o più funzioni, una delle quali deve chiamarsi obbligatoriamente `main`. Una funzione è una sequenza di istruzioni (cioè un sottoprogramma) che compie una o più azioni. Un programma C++ contiene anche una serie di direttive `#include` che consentono di includere *header file* (ovvero *file di intestazioni*) con funzioni e dati predefiniti. Un programma C++ può quindi contenere:

- obbligatoriamente una funzione `main()`;
- direttive di preprocessore come `#include`, `using...`;
- dichiarazioni globali;
- funzioni definite dall'utente;
- commenti al programma;
- istruzioni.

La struttura tipica di un programma C++ è illustrata nella Figura 2.3. Un semplice esempio di programma in C++ è il seguente:

```

1: #include <iostream>
2: using namespace std;
3: // Il programma stampa "Benvenuto alla programmazione in C++"
4: int main()
5: {
6:     cout << "Benvenuto alla programmazione in C++\n";

```

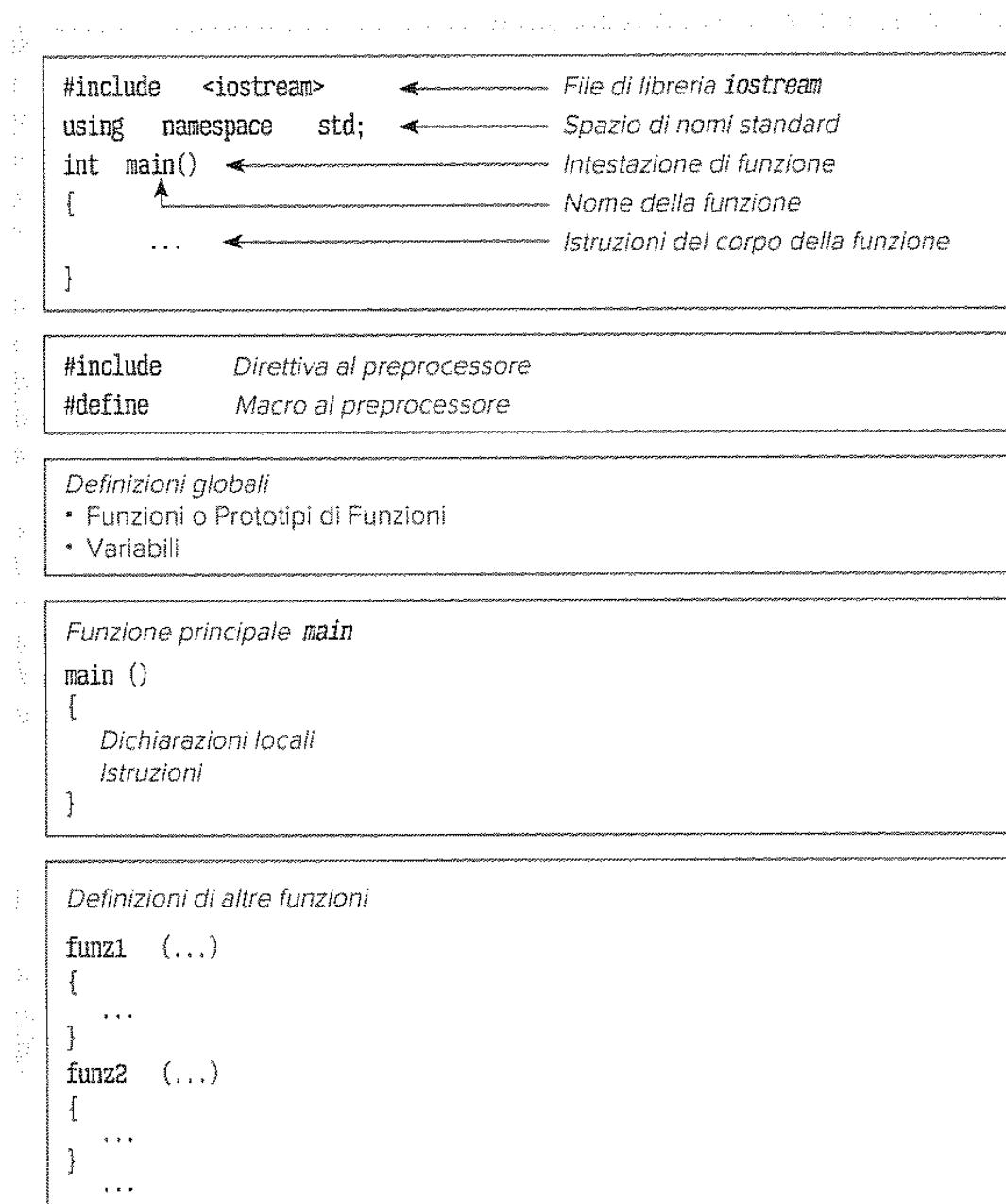


Figura 2.3

Struttura tipica di un programma C++.

```
7:     return 0;
8: }
```

La direttiva della riga 1 include la libreria di funzioni `iostream` per gestire l'I/O e serve per eseguire l'istruzione `cout` di cui alla riga 6. Le parentesi angolari `<` e `>` si utilizzano per indicare che il file è parte della libreria standard del C++, quindi il compilatore "sa" dove tale libreria si trovi all'interno del file system.

La riga 2 è la direttiva `using` che permette di utilizzare il namespace (spazio dei nomi) standard `std` e di riconoscere così il nome standard `cout` del flusso d'output evitando così di riproporre il prefisso `std::` dell'esempio precedente.

La riga 3 è un *commento*. Iniziati dalle doppie sbarre inclinate `(//)`, i commenti sono ignorati dal compilatore ma sono molto importanti perché servono al lettore per capire cosa fa il codice. Il testo ignorato dal compilatore va dalle doppie sbarre inclinate fino al termine della riga.

La riga 4 è l'intestazione della funzione `main()`, obbligatoria in ogni programma C. Essa indica l'inizio del programma e, come qualunque funzione, richiede le parentesi tonde `()` ed eventualmente degli argomenti. La quinta e l'ottava riga contengono solo le parentesi graffe `{` e `}` che racchiudono il corpo della funzione `main()` e sono necessarie in tutti i programmi C.

La riga 6 contiene l'istruzione

```
cout << "Benvenuto alla programmazione in C++\n";
```

che comanda al sistema di inviare il messaggio "Benvenuto alla programmazione in C++\n" al "canale di output standard" `cout`, ovvero il *flusso standard di output* che normalmente invia i byte allo schermo del computer. L'output su schermo sarà

Benvenuto alla programmazione in C++

Il simbolo `\n` non viene scritto perché indica "*nuova riga*" (*newline*). Mettendolo alla fine della sequenza di caratteri tra virgolette si chiede al sistema che cominci una nuova riga dopo aver stampato i caratteri precedenti, terminando pertanto la riga attuale.

La riga 7 contiene l'istruzione `return 0`. Questa termina l'esecuzione del programma e restituisce il controllo al sistema operativo del computer. Il numero 0 si utilizza convenzionalmente per segnalare che il programma ha terminato correttamente.

Il simbolo `<<` si dice *operatore di output* oppure *operatore di inserimento*, perché "inserisce" il messaggio nel "flusso" di output.

Il punto e virgola `(;)` alla fine delle sesta e settima riga serve a specificare che l'istruzione è finita, quindi dopo di esso ne inizia un'altra. Non è necessario che stia alla fine di una riga perché questa può contenere diverse istruzioni e si può estendere un'istruzione su parecchie righe.

### 2.2.1 Il preprocessore e le direttive al compilatore

Come il C, anche il C++ utilizza automaticamente un **preprocessore** all'inizio della compilazione. Il **preprocessore** è un programma che prepara il

codice sorgente affinché il compilatore possa lavorare correttamente. Permette di includere nel codice altri file (ovvero gli header file), definire macroistruzioni, eliminare i commenti ecc.

I programmi C/C++ cominciano di solito con direttive. Le direttive più usuali sono `#include`, `#define` e `using`.

#### Direttiva `#include`

La direttiva `#include nome_file` indica al processore di inserire il contenuto del file `nome_file` nel sorgente. Il preprocessore rimpiazza tutta la direttiva `#include` con il contenuto dell'header file `nome_file`. Per esempio:

```
#include <iostream> // inserisce il contenuto del file iostream
```

L'header file `iostream` è fondamentale nei programmi C++ perché contiene le funzioni di libreria per comunicare con il mondo esterno (io sta per "input/output" e stream per "flusso"). Esso include, fra tante altre funzioni, le dichiarazioni per poter utilizzare gli identificatori `cout` (*channel of output*) e `cin` (*channel of input*). Quindi i programmi che utilizzano `cin` e `cout` debbono includere il file di libreria `iostream`.

#### Header file

I compilatori C/C++ incorporano molte funzioni di libreria raccolte in variati header file che le raggruppano per funzionalità simili. Gli header file del C (utilizzabili anche con i compilatori C++) si caratterizzano per l'estensione `.h`. Per esempio, la libreria `math.h` contiene parecchie funzioni matematiche, ma dallo "Standard ANSI/ISO C++" del 2011 tale libreria si chiama `cmath`.

#### Direttiva `using`

Se si utilizza (come è bene fare) il file `iostream` dello standard ANSI/ISO C++ (invece dell'`iostream.h` delle antiche versioni di C++) bisognerà utilizzare la direttiva `using namespace std` per permettere che le definizioni di `iostream` siano accessibili nel programma senza premettere ogni volta il prefisso `std::`. Questa necessità viene dal fatto che su un programma C++ possono valere differenti spazi di nomi (*namespaces*). Uno **spazio di nomi**, come si vedrà, è la parte del programma su cui certi identificatori sono riconosciuti. Per esempio, `using namespace std` indica che gli identificatori di variabili e istruzioni che seguono possono essere cercati dentro lo spazio di nomi `std`.

In conclusione, se si vuole che il programma utilizzi gli identificatori `cin` e `cout` per indicare rispettivamente i canali di input e di output bisogna scrivere queste due righe all'inizio del programma:

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

ma poiché queste due righe sono praticamente scontate, negli esempi che seguiranno saranno omesse.

### 2.2.2 Funzione main

Ogni programma C++ contiene una funzione `main()` che è la funzione principale, quella che parte quando si lancia il programma. Tutte le altre funzioni sono chiamate direttamente o indirettamente dalla `main`.

La funzione `main()` ha la struttura rappresentata nella Figura 2.4. Il corpo della funzione `main()` è la sequenza di istruzioni che costituiscono il programma.

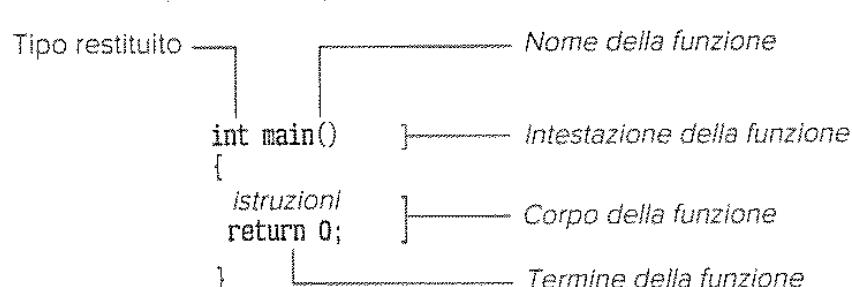
Ogni istruzione deve terminare con il punto e virgola, la sua omissione è l'errore più ricorrente in C/C++.

In generale, una funzione C++ viene *chiamata* (cioè *attivata*) da un'altra funzione e l'intestazione costituisce l'interfaccia tra le due. La parte che precede il nome della funzione definisce il tipo del valore che essa restituirà alla funzione che l'ha invocata (*valore di ritorno*, cioè restituito in *output*). Le parentesi dopo il nome contengono invece la lista degli *argomenti*; anche detti *parametri formali*, essi descrivono il tipo e la quantità dei valori che la funzione dovrà ricevere dalla funzione chiamante.

La `main()` è una funzione speciale perché essa è invocata non da un'altra funzione ma dal sistema operativo. Essa restituisce un valore di tipo intero (indicato con la parola riservata `int`). Il programma termina quando si esegue l'istruzione:

```
return 0;
```

che restituisce al sistema operativo il valore 0 (che è di tipo `int`). Anche se lo standard ANSI/ISO C++ non esige questa istruzione, vecchi compilatori richiedono il suo uso e quindi consigliamo di inserirla. L'istruzione può anche essere scritta `return(0)` e indica al sistema operativo che il programma ha terminato la sua esecuzione con successo. Uno stato diverso da zero indica un errore; normalmente 1 si utilizza per gli errori più semplici, tali come sintassi non corretta o file non trovato.



*Nota:* le istruzioni terminano con il punto e virgola  
`return 0;` termina la funzione `main()`

Figura 2.4  
Funzione `main`.

### 2.2.3 Dichiarazioni globali

Le *dichiarazioni globali* sono quelle scritte prima della funzione `main()`. La posizione in cui vengono scritte indica al compilatore che quegli elementi sono comuni a tutte le funzioni del programma. Se, per esempio, si dichiara una variabile globale di tipo intero per indicare il numero di abitanti di una città:

```
int abitanti;
```

qualunque funzione del programma (incluso ovviamente la `main()`) può accedervi.

La zona delle dichiarazioni globali può includere dichiarazioni di funzione, dette *prototipi di funzione*; per esempio:

```
int media(int a, int b);
```

indica che in tutto il programma si potrà "invocare" una funzione che si chiama `media` e che accetta in input due valori di tipo intero, anche se non sappiamo cosa essa ci farà con quei due valori.

### 2.2.4 Funzioni definite dall'utente

Un programma C++ è un insieme di funzioni. Tutti i programmi si costruiscono a partire da una o più funzioni che si integrano a formare un'applicazione. Tutte le funzioni contengono una o più istruzioni e servono generalmente per realizzare un compito ben definito.

Tutte le funzioni hanno un *nome*. Si può assegnare qualunque nome a una funzione, ma normalmente si cerca di darle uno facile da ricordare e associabile al compito svolto dalla funzione.

Le funzioni si mandano in esecuzione proprio invocando il loro nome e passandogli i valori che eventualmente richiedono nell'intestazione. Questa operazione si dice **chiamata** della funzione.

Per mandare in esecuzione una funzione bisogna che questa sia stata prima dichiarata o definita, altrimenti il compilatore segnala un errore. Una **dichiarazione** di funzione, o **prototipo** di funzione, contiene il nome della funzione, il tipo del valore che essa restituisce e la lista dei *tipi* degli argomenti che essa prende in input. Per esempio, se il programmatore usa una funzione che restituisce la potenza ennesima (con esponente intero passato al primo argomento) un intero (passato come secondo argomento), potrebbe dichiararla così:

```
int eleva_a_ennesima_potenza(int, int);
```

La **definizione** di una funzione specifica invece anche le *istruzioni* che la funzione dovrà eseguire e il nome dei suoi argomenti formali, seguendo questo schema:

<i>tipo_restituito nome_funzione(lista_di_parametri)</i>	<i>intestazione della funzione</i>
[	
<i>istruzioni</i>	<i>corpo della funzione</i>

```

    return; //opzionale      ritorno della funzione
}                                fine della funzione

```

Il tipo restituito dalla funzione può essere void, e in questo caso la funzione si definisce spesso **procedura**. fornisce anche moltissime funzioni predefinite, cioè le *funzioni di libreria*. Per essere utilizzate esse richiedono l'inclusione dell'header file che le contiene, come iostream, math ecc. Vediamo un esempio

```

void saluto(); // prototipo della procedura
int main()
{
    saluto(); // chiamata della procedura
    return 0;
}
void saluto() //definizione della procedura
{
    cout << "Ciao a tutti!\n";
}

```

## 2.2.5 Commenti

I *commenti* sono informazione che si aggiunge al sorgente per spiegare al lettore cosa faccia il codice. Il compilatore ovviamente li ignora. È buona pratica di programmazione commentare il più possibile, sia per sé stessi (quando si torna a leggere il sorgente dopo molto tempo) sia per gli altri. All'inizio si includono di solito il nome del file, il nome del programmatore, una breve descrizione di cosa faccia il programma, la data in cui è stata creata la versione e il numero di versione.

In C++ i commenti di un programma si possono introdurre in due forme:

- *stile C standard*;
- *stile C++*.

### Stile C standard

I commenti in C standard cominciano con la sequenza /\* e terminano con la sequenza \*/. Tutto il testo situato tra le due sequenze è un commento ignorato dal compilatore.

```
/* PROVA1.CPP  Primo programma C++ */
```

oppure:

```
/*
Programma:      PROVA1.CPP
Programmatore:  Pepe Mortimer
Descrizione:   Primo programma C++
Data creazione: 17 giugno 1994
Revisione:     Nessuna
*/
```

o anche:

```
cout << "Ciao a tutti!\n";      /* istruzione di output */
```

#### Stile C++

In stile C++ i commenti iniziano con la doppia sbarra inclinata // e finiscono alla fine della riga.

```
// PROVA1.CPP      Primo programma C++
```

oppure:

```
//Programma:      PROVA1.CPP
//Programmatore:  Pepe Mortimer
//Descrizione:    Primo programma C++
//Data creazione: 17 giugno 1994
//Revisione:     Nessuna
```

o anche:

```
cout << "Ciao a tutti!\n";      // istruzione di output
```

Non si possono annidare commenti, cioè scrivere un commento dentro un altro.

### 2.2.6 Funzioni di libreria

Le *funzioni di libreria* eseguono compiti d'interesse generale come accedere a file, fare calcoli matematici e convertire tipi di dato. Per utilizzarle bisogna includere nel programma la libreria che le contiene. Anche se non sono del tutto comprensibili anticipiamo qui due esempi affinché il lettore si familiarizzi con il concetto di funzione di libreria.

#### Esempio 2.1

*Il seguente programma copia un messaggio in un vettore di caratteri e lo stampa sullo schermo utilizzando la funzione:*

```
char * strcpy (char *, const char *);
```

*(in cui il primo argomento rappresenta la destinazione e il secondo la provenienza) definita nella libreria <string>.*

```
#include <cstring>
int main()
{
    char messaggio[20]; // qui si definisce un vettore di caratteri
    strcpy (messaggio, "ciao mondo\n");
    cout << messaggio;
    return 0;
}
```

In quest'esempio vediamo l'uso della funzione `sqrt` inclusa nella libreria standard `cmath`.

### Esempio 2.2

*Il seguente programma utilizza la funzione di libreria:*

```
double sqrt (double);
per calcolare la radice quadrata di un numero.

#include <cmath>

int main()
{
    double num, sol; // definizione delle variabili num e sol
    cout << "Per favore, introduca un numero: ";
    cin >> num;      //il numero viene letto e scritto in num
    sol = sqrt(num); //scrive in sol la radice di num
    cout << "La radice quadrata è " << sol << endl;
    return 0;
}
```

## 2.3 Debugging di un programma in C++

Un programma non gira quasi mai la prima volta che si tenta di compilarlo. In generale gli errori possono essere di tre tipi:

1. errori *sintattici* nell'uso delle regole grammaticali del linguaggio sorgente;
2. errori *logici* nel progetto dell'algoritmo;
3. errori *logici* nella traduzione dell'algoritmo (corretto) in un programma sorgente (scorretto).

Come abbiamo detto, gli errori del primo tipo li individua direttamente il compilatore (*compile time errors*). Il processo per scoprire e correggere gli altri due tipi di errore (*run-time errors*) si dice *debugging* del programma. Correggere uno di questi errori è più facile che scovarlo. Purtroppo nella storia del software alcuni sono stati scoperti solo quando già avevano prodotto perdite di vite umane e di beni, e chissà quanti errori logici sono ancora da scoprire fra i miliardi di righe di codice che stanno girando in questo momento sui processori di tutto il mondo.

### 2.3.1 Errori di sintassi

Errori sintattici tipici sono l'errata scrittura di parole riservate o di identificatori definiti dal programmatore stesso, e l'omissione di segni di interpunkzione o di operatori (virgolette, punto e virgola ecc.). Gli errori di sintassi li scopre di solito il compilatore segnalandoli mediante *messaggi di errore* (o *diagnosi*) che dovrebbero spiegare il problema (apparente). Questi messaggi

sono a volte difficili da interpretare e purtroppo si può arrivare a conclusioni erronee. Inoltre essi cambiano da un compilatore all'altro. Alcuni esempi:

- punto e virgola dopo l'intestazione `main();`
- omissione della parentesi graffa d'apertura (`()`) o chiusura (`()`) del corpo di una funzione;
- omissione del punto e virgola alla fine di un'istruzione;
- omissione della doppia sbarra inclinata prima di un commento;
- omissione delle doppie virgolette quando si chiude una stringa;
- ...

I messaggi d'errore variano a seconda dei compilatori, ma si distinguono fra errori veri e propri, che impediscono la compilazione, e semplici "avvertenze" (*warnings*) che indicano istruzioni sospette ma grammaticalmente legittime. Per esempio, in questo programma:

```
int main()
{
    float x, y, z; // Dati locali alla funzione main
    y = 10.0;
    z = x + y;
    cout << "Il valore di z è = " << z << endl;
}
```

il compilatore dovrebbe avvisare il programmatore che la variabile `x` non inizializzata viene utilizzata in un'espressione aritmetica e quindi si potrebbe verificare un errore logico nella valutazione di `z` dando troppo per scontato che il valore della variabile `x` sia 0.

### 3.2.2 Errori logici

Ci sono due tipi di *run-time errors*: quelli che non impediscono l'esecuzione del programma (ma producono risultati scorretti) e quelli "fatali" che invece provocano la terminazione del programma (esempi tipici sono la divisione di un numero per zero e il tentativo di leggere una zona di memoria non consentita). Ovviamente gli errori logici del secondo tipo sono più gravi e difficili da scovare.

Per esempio, l'istruzione:

```
double area_cerchio = diametro * PIGRECO + PIGRECO;
```

è sbagliata perché anche il secondo operatore dovrebbe essere di moltiplicazione (`*`) ma il programmatore ha scritto invece l'operatore somma aritmetica (`+`). Il compilatore non produce alcun messaggio di errore perché non è stata violata alcuna regola sintattica, quindi il programma sarà compilato ed eseguito ma non produrrà risultati corretti perché la formula utilizzata per calcolare l'area del cerchio contiene un errore logico. Si possono scoprire questi errori verificando minuziosamente tutto il programma confrontandone l'output con i risultati attesi, ma l'esperienza, la cono-

scenza del C++ e delle tecniche algoritmiche potranno solo ridurre la possibilità di compiere errori logici, non annullarla del tutto. Posto che si riesca a capire che il programma non restituisce risultati corretti, trovare l'errore è uno dei compiti più difficili della programmazione. Per esempio, errori silenti potrebbero capitare quando si legge un dato dall'input e lo si mette in un contenitore (cioè una variabile) definito di un tipo non congruo con il valore letto (per esempio si legge un carattere e lo si mette in una variabile numerica), oppure (gravissimo) quando si raccolgono più dati di quanti il contenitore preposto dal programmatore per accoglierli è in grado di contenere: questo gravissimo errore si chiama *buffer overflow* ed è alla base di molti attacchi e violazioni alla sicurezza dei sistemi informatici.

A volte la correzione di un errore logico può incidentalmente comportare l'introduzione di un altro. In questo caso si parla di **errori di regressione**.

### 2.3.3 Collaudi

Per determinare se un programma contiene un errore logico silente, lo si lancia più volte utilizzando vari dati di collaudo e verificandone ogni volta la correttezza del comportamento. Questo **collaudo** (*testing*) si deve fare parecchie volte, utilizzando differenti dati di input idealmente preparati da persone diverse dal programmatore, che possano indicare casi d'uso a lui non evidenti.

Stabilito che un programma contiene errori logici, la loro localizzazione è una delle parti più difficili della programmazione. Il **debugger** è un programma che serve proprio per aiutare a scoprire, verificare e correggere errori logici eseguendo le istruzioni del programma un passo alla volta (si dice che bisogna seguire la *traccia*, o fare il *tracing* del programma) fino a trovare il punto in cui il programma si comporta in maniera sbagliata. Per semplificare il tracing, i compilatori C++ forniscono un debugger integrato nell'IDE. Il debugger permette appunto di eseguire il programma riga dopo riga, osservando ogni volta gli effetti dell'istruzione sui valori degli oggetti del programma.

Non è quasi mai possibile verificare un programma per tutti i possibili dati di collaudo. Esistono casi di programmi utilizzati professionalmente per anni senza problemi prima che l'immissione di una specifica combinazione di valori in ingresso producesse finalmente un comportamento anomalo mettendo in evidenza la presenza di un errore logico.

Man mano che i programmi crescono in dimensione e complessità, il problema dei collaudi si fa sempre più serio. Non importa quanti collaudi si facciano: «i collaudi non terminano mai, solo si sospendono, e non esistono garanzie che siano stati trovati e corretti tutti gli errori di un programma». Come osservò Dijkstra, già agli inizi degli anni settanta: «*I collaudi mostrano la presenza di errori, non la loro assenza. Non si può provare che un programma è corretto, ma solo che è scorretto*».

## 2.4 Elementi di un programma in C++

Un programma C++ consiste in uno o più file. Un file è compilato in differenti fasi. La prima fase è il preprocessamento che produce una sequenza di *tokens* (elementi lessicali).

Esistono cinque classi di *token*: identificatori, parole riservate, letterali, operatori e separatori.

### 2.4.1 Identificatori

Un *identificatore* è una sequenza di caratteri che possono essere solo *lettere*, *cifre* e il carattere *underscore* (\_) che serve a indicare un dato o una funzione. In C++ il primo carattere di un identificatore non può essere una cifra. Il C++ è *case sensitive*, distingue cioè fra lettere maiuscole e minuscole. Sono esempi di identificatori:

```
nome_classe
alfa
Giorno_Mese_Anno
Alfa
i
Stanza120
```

Gli identificatori possono essere di qualunque lunghezza, ma farli corti è utile per non sbagliare. Consigli di stile:

1. identificatori di variabili con tutte lettere minuscole;
2. identificatori di costanti con tutte lettere maiuscole;
3. identificatori di funzioni con le minuscole ma iniziando con la maiuscola.

### 2.4.2 Parole riservate, segni d'interpunzione e separatori

Il C++ riserva a sé stesso i seguenti identificatori (*keywords o reserved words*) perché li usa con significati e compiti specifici predefiniti.

asm	double	mutable	struct
auto	else	namespace	switch
bool	enum	new	template
break	explicit	operator	this
case	extern	private	throw
catch	float	protected	try
char	for	public	typedef
class	friend	register	union
const	goto	return	unsigned
continue	if	short	virtual
default	inline	signed	void
delete	int	sizeof	volatile
do	long	static	wchar_t
			while

Tutte le istruzioni devono terminare con un punto e virgola (;).

Altri segni d'interpunzione sono:

{ % ^ & \* ( ) - + = [ ] ~ [ ] \ ^ : < > ? , . / "

I separatori sono gli spazi bianchi (' '), le tabulazioni (\t), il carattere di ritorno di carrello (*carriage return* \r) e quello di nuova linea (*newline* \n).

## 2.5 Tipi di dato predefiniti

C/C++ non supporta molti tipi di dato predefiniti (Tabella 2.1), ma consente al programmatore di crearne di propri. Quelli predefiniti sono relativi a queste tipologie:

- numeri **interi**: tipo **int** e varianti, ovvero tipi **short**, **long** e **unsigned**;
- numeri **reali**: (in virgola mobile) tipi **float**, **double** o **long double**;
- **caratteri**: tipo **char** (ASCII), **wchar\_t** (non ASCII);
- enumerativi: **bool** (tipo *falso-vero*), **enum** (tipo enumerativo);
- vuoto: **void**.

I caratteri si rappresentano quindi su 8 bit con il tipo **char** (gli ASCII) e su 16 bit per il tipo **wchar\_t** (gli UNICODE).

**Tabella 2.1** Tipi di dato predefiniti in C/C++

Tipo	Dimensione in byte	Minimo ... Massimo
char	1	-127 ÷ 127 oppure 0 ÷ 255
unsigned char	1	0 ÷ 255
signed char	1	-127 ÷ 127
int	4	-2147483648 ÷ 2147483647
unsigned int	4	0 ÷ 4294967295
signed int	4	-2147483648 ÷ 2147483647
short int	2	-32768 ÷ 32767
unsigned short int	2	0 ÷ 65,535
signed short int	2	-32768 ÷ 32767
long int	8	-2,147,483,648 ÷ 2,147,483,647
signed long int	8	-2,147,483,648 ÷ 2,147,483,647
unsigned long int	8	0 ÷ 4,294,967,295
long long int	8	-(2^63) ÷ (2^63)-1
unsigned long long int	8	0 ÷ 18,446,744,073,709,551,615
float	4	
double	8	
long double	12	
wchar_t	2 o 4	1 carattere UNICODE

Inserendo caratteri in vettori (ne parleremo in seguito) si creano le *stringhe*, ovvero sequenze di caratteri terminate con il carattere ASCII NULL (quello costituito da 8 bit tutti a 0).

Esistono caratteri ASCII che hanno compiti speciali in C/C++, quindi non possono essere scritti normalmente; si fornisce per essi una **sequenza di escape**. Per esempio, il carattere apostrofo si può scrivere come

\'

e il carattere nuova riga

\n

La Tabella 2.2 elenca le diverse sequenze di escape del C++.

Il tipo **void** indica il non-valore. Questo tipo si utilizza per:

1. dichiarare una funzione che non restituisce un valore (cioè una procedura);
2. definire puntatori generici (vedremo più avanti).

Il tipo enumerativo **bool** può assumere solo due valori, true e false. La maggior parte delle espressioni booleane appaiono in strutture di controllo che servono per indirizzare il flusso delle istruzioni C++. Raramente si ha la necessità di leggere valori **bool** come dato di input o di visualizzare valori **bool** come risultati di programma, ma se necessario, si può visualizzare il valore della variabile di tipo **bool** utilizzando l'operatore di output <<. Così, se bandiera è false, l'istruzione

```
cout << "Il valore di bandiera è " << bandiera;
```

**Tabella 2.2 Sequenze di escape**

Sequenza di Escape	Significato	Dec	Hex
\n	Nuova riga	10	0A
\r	Ritorno di carrello	13	0D
\t	Tabulazione orizzontale	9	09
\v	Tabulazione verticale	11	0B
\a	Bip sonoro	7	07
\b	Retrocessione di uno spazio	8	08
\f	Avanzamento di una pagina	12	0C
\0	Zero, nullo	0	0
\	Sbarra inclinata inversa	92	5C
\`	Apice	39	27
\^	Virgolette	34	22
\?	Punto interrogativo	34	22
\ooo	Numero ottale	Tutti	Tutti
\xhh	Numero esadecimale	Tutti	Tutti

visualizzerà

Il valore di bandiera è 0

Se invece bisogna leggere in input un dato boolean, si può scrivere sull'input l'intero 0 per inserire il valore false e l'intero 1 per il valore true.

Rimandiamo alla definizione del concetto di "costante" la definizione del tipo enumerativo enum.

## 2.6 Variabili

Una *variabile* è una sequenza di una o più celle di memoria caratterizzata da un *indirizzo* (quello della prima cella) e da un *tipo*. L'indirizzo è associato dal compilatore al *nome* della variabile (un identificatore), mentre la variabile stessa viene utilizzata per ospitare valori di quel tipo che possono poi essere modificati. Le variabili possono ospitare ogni tipo di dato: stringhe, numeri e strutture. Una *costante*, invece, è una variabile il cui valore non può essere modificato.

### 2.6.1 Definizione di una variabile

Per poter essere utilizzata in un programma ogni variabile deve essere prima definita, così si riserva spazio in memoria per ospitare un dato di quel tipo. Per definire una variabile bisogna quindi scrivere il tipo di dato e l'identificatore scelto come nome della variabile. Il formato della definizione è:

**<tipo di dato> <nome variabile>**

Si possono anche definire più variabili dello stesso tipo nella stessa riga:

**<tipo\_di\_dato> <nom\_var1>, <nom\_var2> ... <nom\_varn>**

Per esempio:

```
int valore;
int valore1, valore2;
char risposta;
float f;
double h;
bool trovato;
```

Il nome della variabile deve essere un identificatore valido. È frequente utilizzare l'*underscore* ( \_ ) per ottenere maggiore leggibilità e un'intuitiva corrispondenza con l'elemento del mondo reale rappresentato.

### 2.6.2 Inizializzazione di variabili

Quando si definisce una variabile, questa non ha alcun valore predefinito associato (benché il più delle volte il valore di default sia 0). Ciò significa che

se si utilizza quella variabile il programma potrebbe funzionare in maniera imprevedibile, è pertanto bene inizializzare le variabili.

Il formato generale di una definizione con inizializzazione è:

**<tipo di dato> <nome variabile> = <espressione>**  
dove <espressione> è qualunque espressione valida il cui valore sia dello stesso tipo che *tipo*.

Per esempio:

```
int valore = 99;
int num_parte = 1141, num_item = 45;
bool trovato = true;
char risposta = 'S';
int contatore = 1;
char var2 = '\t';
float peso = 156.45;
```

Si può definire una variabile in due punti dentro un programma:

- all'inizio di un file o blocco di codice;
- nel punto di utilizzo.

#### All'inizio di un file o blocco di codice

Questo è il metodo tradizionale C per definire una variabile: all'inizio del file o all'inizio della funzione che la utilizza.

```
int MioNumero; //variabile all'inizio del file

int main()
{
    cout << "Quale è il suo numero preferito?";
    cin >> MioNumero;
    return 0;
}
```

o anche:

```
int main()
{
    int i; //variabile all'inizio di funzione
    int j;
    ...
}
```

#### Nel punto di utilizzo

C++ fornisce una maggiore flessibilità per definire variabili, è infatti possibile definirle nel punto preciso dove saranno utilizzate. Questa proprietà si uti-

lizza molto nei cicli (Capitolo 4). In C il sistema per dichiarare una variabile per controllare il ciclo for è:

```
int j;
for(j = 0; j < 10; j++)
{
    // ...
}
```

ma in C++ è possibile dichiarare la variabile nel momento del suo utilizzo dentro l'istruzione for:

```
for(int j = 0; j < 10; j++)
{
    // ...
}
```

Come vedremo, in C++ le definizioni di variabili si possono trovare fra le istruzioni eseguibili, ma il loro ambito di visibilità rimane confinato al blocco d'istruzioni in cui sono definite.

### Esempio 2.3

*Il seguente programma mostra come una variabile possa essere definita in qualsiasi parte di un programma C++.*

```
int main()
{
    int luna_miglia = 238857;      //distanza terra-luna in miglia
    cout << "Distanza alla Luna " << luna_miglia << " miglia\n";
    int luna_chilo;
    luna_chilo = luna_miglia * 1.609; //miglio = 1.609 chilometri
    cout << "In chilometri è " << luna_chilo << " km.\n";
}
```

## 2.7 Durata e visibilità di una variabile

La zona di un programma in cui una variabile è attiva si dice *visibilità (scope)* della variabile. Le variabili in C++ possono essere:

- *locali;*
- *globali.*

### 2.7.1 Variabili locali

Un blocco di istruzioni è una sequenza di istruzioni racchiusa da una coppia di parentesi graffe. Le variabili locali sono quelle definite all'interno di un blocco di istruzioni, tipicamente quello di una funzione, e sono visibili dal punto in cui sono definite fino alla fine di quel blocco.

Le regole che le riguardano sono:

1. una variabile locale non può essere modificata da istruzioni esterne al blocco in cui è definita;
2. funzioni/blocchi diversi possono utilizzare nomi identici per le proprie (distinte) variabili locali;
3. le variabili locali non esistono in memoria fino a che non si esegue la funzione/blocco in cui sono definite.

L'esempio più importante riguarda le variabili definite nel blocco del main:

```
int main()
{
    int a, b, somma; //variabili locali al programma principale
    cout << "Immetti due numeri /n";
    cin >> a >> b;
    somma = a + b;
    cout << "La loro somma è " << somma;
}
```

### 2.7.2 Variabili globali

Le *variabili globali* sono quelle che si dichiarano al di fuori di qualunque blocco, quindi anche al di fuori dalla `main()`, e sono visibili dappertutto (quindi da qualunque funzione), meno che nei blocchi (ovvero nelle funzioni) in cui esistono variabili locali con lo stesso nome.

```
int a, b, c; //definizione di variabili globali
int main()
{
    int valore; // definizione di variabile locale
    //...
}
```

La memoria assegnata a una variabile globale rimane occupata fino al termine del programma. Quando è possibile è meglio utilizzare variabili locali, non solo per risparmiare spazio di memoria, ma anche perché le funzioni possono modificare le variabili globali e si possono fare in questo modo errori difficilmente individuabili.

### 2.7.3 Spazio di nomi

Lo **spazio di nomi** (*namespace*) è una caratteristica del C++ introdotta per semplificare la scrittura dei programmi. Si tratta di un blocco, definito e nominato dal programmatore, che contiene definizioni di funzioni e variabili, in maniera tale che si possa accedere a questi elementi dal di fuori del blocco semplicemente anteponendo al loro nome quello dello spazio stesso.

**Esempio 2.4**

```
namespace geo
{
    double PIGRECO = 3.141592;
    double lungcircon (double raggio)
        { return 2*PIGRECO*raggio; }
} //fine spazio di nome geo
```

in questo blocco vengono definite una variabile e una funzione che non potranno essere utilizzate al di fuori del blocco, cosicché la seguente chiamata di funzione è errata:

```
double c = lungcircon(16); //errore, non funziona perché la funzione
                           // lungcircon non è visibile da fuori il blocco
```

ma poiché il blocco è nominato "geo", si può accedere ai suoi elementi dal di fuori del blocco semplicemente anteponendo il nome del blocco al nome degli elementi richiesti ponendo fra i due l'*operatore di risoluzione dei nomi* (::)

```
double c = geo::lungcircon(16); //corretto
```

Per evitare di scrivere troppe volte questi identificatori composti si può utilizzare la direttiva `using` la quale dichiara che da essa in poi si sottointenderà l'anteposizione del nome dello spazio di nomi a quelli dei suoi elementi. Per esempio:

```
using namespace geo;
double c = lungcircon(16); //corretto
```

La validità della direttiva `using` arriva fino alla fine del blocco in cui è inserita.

```
void calcoloSup()
{
    using namespace geo;

    double c = lungcircon(r); //corretto
}
double c = lungcircon(r); //scorretto
```

la funzione `lungcircon` è accessibile soltanto dentro il corpo della funzione `calcoloSup` perché in essa vale la direttiva `using namespace geo`.

**Esempio 2.5**

```
int test;           //test è una variabile globale
namespace demo
{
    int test;       //inizializzazione di test nel namespace demo
    int test;       //variabile locale con stesso nome variabile globale
}                   //fine namespace demo
```

```

int main () [
    ::test = 0;      // assegna il valore 0 alla variabile globale
    cout << test;   // stampa 0
    demo::test = 10; // assegna il valore 10 alla variabile locale
    cout << demo::test;// stampa 10
]

```

### Spazi di nomi negli header file

I namespace si utilizzano spesso negli header file. Ogni libreria ha il suo proprio namespace. Per esempio, la libreria standard `<iostream>` contiene lo spazio di nomi `std`; dopo averla inclusa sarebbe necessario invocare il flusso della standard output `cout` come `std::cout`. Per evitare ciò si utilizza la direttiva `using`

```
using namespace std;
```

Questa direttiva, inserita all'inizio del sorgente, fa sì che in tutto il programma siano accessibili gli elementi definiti nella libreria `<iostream>`. La direttiva non era necessaria nei vecchi programmi C++ perché il vecchio standard per le librerie `<iostream.h>` non utilizzava i namespace.

I namespace permettono di mantenere separate le librerie del C++ per non farle interferire le une con le altre. I nomi possono riferirsi a variabili, funzioni, strutture, enumerazioni, classi e membri di strutture e classi. Al crescere della dimensione dei programmi cresce pure il rischio di conflitti fra i nomi, specialmente se si utilizzano librerie di classi definite indipendentemente in diversi programmi. Per esempio, due librerie possono avere definito classi diverse con gli stessi nomi: `Lista`, `Albero` e `Nodo`. Ora nulla vieta che si possa desiderare di utilizzare la classe `Lista` di una libreria e la classe `Albero` dell'altra, di ognuna utilizzando poi la propria versione di `Nodo`. Tali conflitti si risolvono proprio qualificando gli spazi di nomi.

## 2.8 Istruzione di assegnamento

Alle variabili definite e visibili si può dare un valore tramite l'*istruzione di assegnamento*, ovvero l'operatore `=`.

**variabile = espressione;**

### Esempio 2.6

```

temperatura = 75.45;
conto = conto + 5;
spazio = velocita * tempo;

int risposta; //risultato di un'operazione
risposta = (1+4)*5;

```

Alla variabile a sinistra dell'operatore è assegnato il valore dell'espressione a destra; il punto e virgola termina l'istruzione.

**Esempio 2.7**

```
int main()
{
    aux = 4*8;
    cout << "Due per " << aux << " fa: " << 2*aux << "\n";
    cout << "Tre per " << aux << " fa: " << 3*aux << "\n";
    return (0);
}
```

**2.9 Costanti**

Alcuni dati rimangono inalterati per tutto il programma. In C++ esistono quattro tipi di costanti:

- costanti letterali;
- costanti enumerative;
- costanti definite;
- costanti dichiarate.

Le costanti si possono definire con la parola riservata `const` assegnandole contemporaneamente un valore; questo valore non si può modificare durante il programma e qualunque intento di alterarlo produrrà un messaggio di errore nella fase di compilazione.

**2.9.1 Costanti letterali**

Le costanti letterali sono di quattro tipi:

- costanti intere;
- costanti carattere;
- costanti in virgola mobile;
- costanti stringa.

**Costanti intere**

Seguono determinate regole:

- non utilizzare virgolette o punti per le migliaia  
123456    *invece di*    123.456
- per forzare un valore al tipo `long` terminare con la L maiuscola  
1024    *è un tipo int*  
1024L    *è un tipo long*
- per forzare un valore al tipo `unsigned`, terminarlo con la U maiuscola  
4352U    *è un tipo unsigned*
- si possono utilizzare formati non decimali  
0777 *Formato ottale*                         (vengono precedute dalla cifra 0)  
0xFF3A *Formato esadecimale*                 (vengono precedute da "0x" oppure, "0X")

- si possono combinare i suffissi

`3466UL`

### Costanti reali

Una costante in virgola mobile rappresenta un numero reale; hanno sempre il segno e rappresentano approssimazioni di valori esatti.

`82.0 347 .63 -83 47e-4 1.25E7 61.e+4`

Per scrivere numeri in notazione esponenziale, si deve far seguire la parte decimale del numero dalla lettera E (oppure e), poi dall'esponente. Per esempio,

`4.5E+5 -3.2E-6 7.12E6`

La notazione scientifica viene rappresentata con un esponente positivo o negativo.

`2.5E4` *equivale a* 25000  
`5.435E-3` *equivale a* 0.005435

### Costanti carattere

Una costante char è un carattere ASCII racchiuso tra apici.

`'A' 'b' 'c'`

Una costante char supporta anche caratteri speciali che non si possono rappresentare utilizzando la tastiera, come, per esempio, i codici ASCII alti e le sequenze di escape.

#### Esempio 3.8

```
int main()
{
    char bip = '\a';           //emette un bip sonoro
    char backspace = '\b';    //sposta il cursore indietro di uno spazio
    cout << bip;
    cout << backspace;
    return 0;
}
```

### Costanti stringa

Una *costante stringa* (o semplicemente *stringa*) è una sequenza di caratteri racchiusa tra virgolette:

`"123"`  
`"12 ottobre 1492"`  
`"questa è una stringa"`

Si può scrivere una stringa su più righe, terminando ogni riga con "\n".

`"questa è una stringa\nche ha due righe"`

Si può concatenare stringhe, scrivendo

"ABC" "DEF" "GHI" "JKL"

che equivale a

"ABCDEFGHIJKL"

In memoria, le stringhe si rappresentano con una sequenza di ASCII terminata dal carattere 0, quello cioè costituito da 8 bit tutti a 0. Il carattere nullo marca la fine della stringa ed è inserito automaticamente dal compilatore C++ alla fine delle costanti stringa. Per rappresentare il byte 0 il C++ definisce la costante `NULL`.

Si faccia bene attenzione al fatto che una costante carattere si racchiude tra due apici, mentre le costanti stringa si racchiudono fra virgolette. Per esempio,

'A'	è una costante carattere memorizzata come 01100001
"A"	è una costante stringa memorizzata come 01100001 00000000

### 2.9.2 Costanti enumerative

Il C++ introduce la parola riservata `enum` per definire nuovi tipi di dato semplicemente elencando in sequenza i valori costanti ammessi per quei tipi. Per esempio, un nuovo tipo `Colori` può essere dichiarato dal programmatore così:

```
enum Colori {Rosso, Arancione, Giallo, Verde, Blu, Viola};
```

ovvero elencando dentro le parentesi graffe i valori costanti enumerativi che una variabile dichiarata di quel tipo può assumere; per esempio, la variabile `Colorefavorito`:

```
Colori Colorefavorito;
```

può assumere solo uno dei predetti valori. A livello macchina, il compilatore *enumera* gli identificatori, cioè assegna un valore a ogni costante cominciando dallo 0; così, ROSSO equivale a 0, ARANCIONE a 1 ecc. Per esempio, i seguenti assegnamenti sono equivalenti:

```
Colorefavorito = Verde;
Colorefavorito = 3;
```

È possibile alterare l'enumerazione fatta dal compilatore assegnando alle costanti enumerative valori diversi da quelli corrispondenti alla sequenza naturale

```
enum Semaforo {Verde, Giallo = 10, Rosso};
```

### 2.9.3 Costanti definite

Ai valori costanti possono essere assegnati nomi simbolici tramite la direttiva `#define`. Per esempio, con queste direttive al compilatore:

```
#define NUOVARIGA '\n'
#define PIGRECO 3.141592
#define VALORE 64
```

il preprocessore del compilatore C++ compie preliminarmente un'operazione di taglia-e-cuci sostituendo in tutto il codice sorgente le costanti letterali '\n', (carattere) 3.141592 (reale) e 64 (intera) alle sequenze di caratteri (costanti simboliche) NUOVARIGA, PI e VALORE ovunque esse appaiano nel testo.

```
cout << "Il valore è " << VALORE << NUOVARIGA;
```

Le direttive non sono istruzioni, perciò non terminano con il punto e virgola.

#### 2.3.4 Costanti dichiarate

A volte è bene che una variabile, una volta acquisito il suo valore (per esempio, all'atto della sua definizione) non modifichino mai il loro valore nel corso dell'esecuzione del programma, ovvero non siano mai soggette a operazioni di assegnamento. Il qualificatore const, premesso al nome della variabile, permette di ottenere questo risultato. Il formato generale per creare una "costante dichiarata" (utilizzando una "definizione" di variabile, la qual cosa genera una confusione di termini) è:

```
È possibile utilizzare questo tipo di costante anche con i tipi di dati complessi come strutture, classi, ecc. In questo caso si deve specificare il tipo di dato prima del nome della costante.
```

```
const int mesi=12;
const float pigreco = 3.141592;
const char carattere = '@';
const int ottale = 0233;
const char stringa[] = "Corso di C++";
```

Al pari di una normale definizione di variabile, questo tipo di costante occupa spazio nella memoria, mentre i valori degli altri tipi di costante no.

#### 2.40 Input/output da console

La libreria iostream fornisce un ampio insieme di istruzioni e operatori per l'I/O; in particolare essa definisce quattro oggetti, cin, cout, cerr e clog, per consentire l'I/O di dati dalla console del sistema operativo.

Come sappiamo, per utilizzare questa libreria bisogna inserire nel sorgente le direttive:

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

iostream definisce due tipi denominati istream e ostream, che rappresentano flussi di input e di output, rispettivamente. Un **flusso** (*stream*) è una sequenza di caratteri preparati per leggere o scrivere da/su un qualunque

dispositivo di I/O. Il termine «flusso» suggerisce che i caratteri si generano o consumano serialmente nel tempo.

### Oggetti standard di input/output

Per maneggiare l'input, si utilizza un oggetto del tipo `istream` chiamato `cin`; questo oggetto è noto come *input standard*, ed è normalmente collegato alla tastiera. Per l'output, si utilizza un oggetto del tipo `ostream` chiamato `cout`, noto anche come *output standard*, che si collega normalmente allo schermo.

La libreria definisce anche altri due oggetti `ostream` denominati `cerr` e `clog`. L'oggetto `cerr` è noto come *standard error* e si utilizza normalmente per generare messaggi di errore e precauzione (*warning*) agli utenti dei programmi. L'oggetto `clog` si utilizza per informazioni generali sull'esecuzione del programma.



In generale, il sistema associa ognuno di questi oggetti con la finestra nella quale si esegue il programma. Così, quando si leggono dati da `cin`, i dati si leggono dalla finestra in cui si sta eseguendo il programma e quando scriviamo con `cout`, `cerr` o `clog`, l'output si scrive nella stessa finestra.

La maggior parte dei sistemi operativi fornisce un modo di ridirigere i flussi di I/O quando si esegue un programma. Utilizzando la ridirezione l'utente del programma (non il programmatore) può associare questi flussi con file a sua scelta.

#### 2.10.1 Canale di input (`cin`)

La libreria `iostream` definisce l'oggetto `cin` per rappresentare il flusso di input (Figura 2.5), e utilizza l'operatore *di estrazione* `>>` per estrarre da esso caratteri e metterli in una qualche variabile. `>>` è un operatore intelligente; trasforma la serie di caratteri ricevuti dalla tastiera, rendendoli in un formato accettabile per la variabile dove si colloca l'informazione. Se non si ridirige esplicitamente `cin`, l'input viene letto dalla tastiera.

L'istruzione

```
cin >> numero;
```

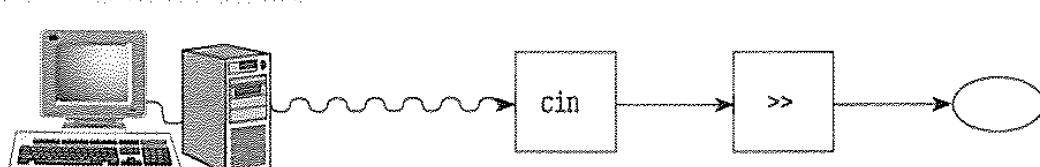


Figura 2.5

Input di una variabile con `cin`.

aspetta che l'utente digitи sulla tastiera il valore di numero e prema "invio".

```
int n;      double x;
cin >> n;   cin >> x;
```

L'estrattore `>>` puо operare in cascata.

```
cin >> variabile1 >> variabile2 >> ...;
```

### 2.10.2 Canale di output (`cout`)

La libreria `iostream` definisce l'oggetto `cout` per rappresentare il flusso di output, e utilizza l'operatore *di inserzione* `<<` che invia i dati presenti alla sua destra nell'oggetto alla sua sinistra (normalmente il `cout`) (Figura 2.6). Si possono cosi visualizzare in output valori di variabili e stringhe di testo. Per esempio:

```
cout << "Questa є una stringa";
```

visualizza in output:

Questa є una stringa.

L'inseritore `<<` puо operare in cascata:

```
cout << "Ciao mondo C++, " << "sono affascinato!";
```

Per andare a capo si puо usare la sequenza di escape '`\n`':

```
cout << '\n'; // cosi si salta una riga
```

Per esempio, l'istruzione:

```
cout << "Benvenuto\nal\nC++!";
```

visualizzerà a schermo:

```
Benvenuto
al
C++
```

ma in C++ un altro metodo per andare a capo є quello di utilizzare l'identificatore riservato `endl`.

```
cout << endl; // cosi si salta una riga
```

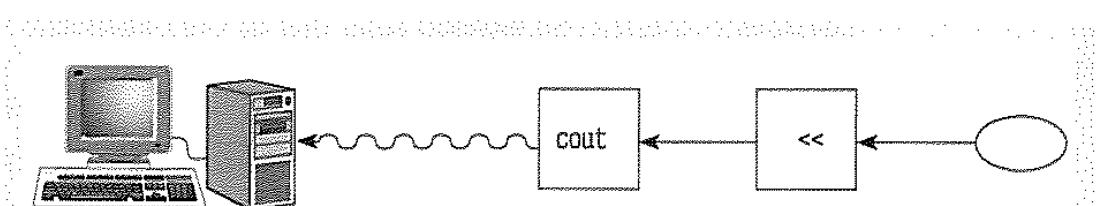


Figura 2.6

Output di una stringa variabile con `cout`.

**PROBLEMA**

L'esecuzione di << non muove automaticamente il cursore alla riga successiva su cout, lo si deve esplicitamente indicare concludendo la stringa con '\n' oppure concudendo la cascata con << endl;

**Esempio 2.9**

```
cout << "Il buono, il" << "brutto,";
cout << "ed il cattivo" << endl;
```

**CALCOLO**

In questo capitolo abbiamo iniziato a programmare in C++. Abbiamo introdotto la struttura generale di un programma e la funzione `main()`. Abbiamo introdotto il concetto di *tipo di dato* e visto come si definiscono le *variabili* di quel tipo. Abbiamo visto l'importanza della direttiva `#include` che consente d'accedere alle funzioni definite nei file di libreria. Il flusso dello standard input `cin` è l'operatore d'estrazione `>>` per ottenere input dalla

tastiera, così come il flusso dello standard output `cout` è l'operatore d'inserzione `<<` per mandare l'output sullo schermo.

Nei successivi capitoli si analizzeranno in profondità ognuno dei componenti qui illustrati e s'introdurranno le specifiche caratteristiche del C++ che permettono di scrivere programmi orientati agli oggetti.

Soluzioni degli esercizi sul sito web [www.mheducation.it](http://www.mheducation.it)

**RISOLUZIONE**

- Codice eseguibile
- Codice sorgente
- Codice oggetto
- Commenti
- Direttiva `#include`
- Header file
- Funzione `main()`
- Tipi di dato predefiniti: `char`, `int`, `float`/`double`
- Variabili
- Costanti
- Flussi di I/O, `cin >>`, `cout <<`

**Esercizi**

Cosa c'è di scorretto nel seguente programma?

```
#include <iostream>
// Questo programma stampa "Ciao mondo!"
main()
{
    cout << "Ciao mondo!\n"
```

```
    return 0;
}
```

"Debuggare" il seguente programma

```
// un programma C++ senza errori
#include <iostream>
```

```
int main()
{
    cout << "Il linguaggio di programmazione C++"
    << endl; // { main()
}
```

**Esercizio 1** Scrivere ed eseguire un programma che stampi la data del giorno.

**Esercizio 2** Scrivere ed eseguire un programma in C++ che scriva in due righe diverse le frasi: *"Benvenuto al C++"* e *"Cominceremo presto a programmare in C"*.

**Esercizio 3** Qual è l'output del seguente programma, se si introducono da tastiera 'A' e 'F'?

```
int main()
{
    char primo, ultimo;
    cout << "Introduca le sue iniziali:\n";
    cout << "\t Primo cognome:";
    cin << primo;
    cout << "\t Secondo cognome:";
    cin << ultimo;
    cout << "Ciao, " << primo << ". " << ultimo <<
    ".!\n";
    return 0;
}
```

**Esercizio 4** Anche se non si è ancora in grado di rispondere con certezza, provare a

immaginare quale sia l'output del seguente programma e testarlo con il proprio compilatore.

```
#include <iostream>
using namespace std;
#define collaudo "Prova ad immaginare"
int main()
{
    char frase[21] = " cosa viene scritto a
    schermo.";
    cout << collaudo << endl;
    cout << " se sei in grado, \n";
    cout << frase << endl;
}
```

**Esercizio 5** Scrivere un programma che stampi a schermo la lettera B composta da asterischi.

```
*****
*   *
*   *
*****
```

**Esercizio 6** Scrivere un programma che legga una variabile intera e due reali e le visualizzi a schermo.

# Operatori ed espressioni

3

<b>3.1</b>	Operatori ed espressioni	<b>3.7</b>	Operatori di manipolazione dei bit
<b>3.2</b>	Operatore di assegnamento	<b>3.8</b>	Operatore condizionale
<b>3.3</b>	Operatori aritmetici	<b>3.9</b>	Operatore virgola
<b>3.4</b>	Operatori di incremento e decremento	<b>3.10</b>	Operatore sizeof
<b>3.5</b>	Operatori relazionali	<b>3.11</b>	Conversioni di tipo
<b>3.6</b>	Operatori logici		

## Introduzione

In questo capitolo introduciamo la sintassi C++ per le espressioni matematiche e logiche. Vedremo gli operatori aritmetici, logici e relazionali, gli operatori di manipolazione di bit,

gli operatori condizionali e quelli speciali. S'introdurranno i concetti di *precedenza* e *associatività*, e si analizzeranno le regole per la conversione dei tipi di dato.

### 3.1 Operatori ed espressioni

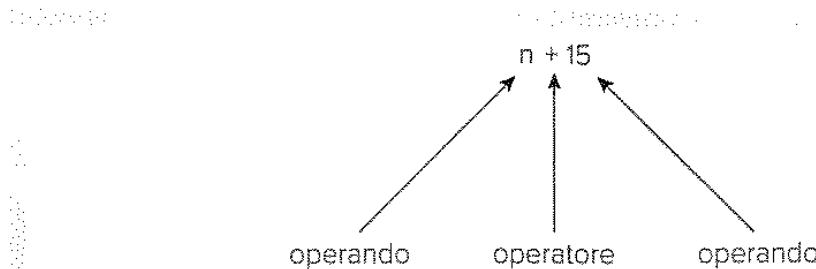
Un'espressione è costituita da uno o più operandi che si combinano tra loro tramite operatori per impostare un'operazione e restituire un risultato. Una costante o una variabile costituiscono già un'elementare forma d'espressione che non ha operatore ma restituisce comunque un valore, per esempio:

```
4.5          //espressione che restituisce il valore 4.5  
Ore_Settimana //costante che restituisce un valore, e.g. 30
```

Ogni espressione deve produrre infatti un risultato. Se l'espressione contiene operatori, il suo risultato si determina applicando ogni operatore ai suoi operandi. Gli operatori rappresentano operazioni aritmetiche, logiche, matematiche o di manipolazione digitale (spostamenti, slittamenti ecc.) (Figura 3.1).

#### 3.1.1 Espressioni

Le più comuni sono forse le espressioni aritmetiche, che fanno uso degli operatori + (somma), - (sottrazione), \* (moltiplicazione), / (divisione) e % (resto della divisione intera); esse restituiscono un valore intero o reale a



**Figura 3.1** Struttura di un'espressione.

Espresione con un operatore e operandi.

seconda degli operandi. In generale un'espressione si può utilizzare in qualsiasi punto del codice in cui sia legale inserire un valore del tipo risultante dall'espressione stessa. Quindi le operazioni si possono innestare, cioè un operando può essere a sua volta un'espressione, a patto che quest'ultima restituisca un valore del tipo giusto per l'operando che essa rappresenta.

#### Esempio 3.1

Supponiamo definite e inizializzate due variabili di tipo intero, *i* e *s*:

```
int i=1, s=15;
```

quella che segue è una lista di espressioni (alcune sono anche istruzioni).

```
false      // restituisce il valore false
s          // restituisce il valore 15 (valore della variabile)
25         // restituisce il valore 25 (costante letterale)
s + 25    // restituisce il valore 40 (espressione aritmetica)
s == 40   // operatore di confronto, verifica l'uguaglianza dei
          // valori, in questo caso restituisce il valore false
          // perché 15 è diverso da 40
i++       // istruzione che utilizza l'operatore di post-incremento;
          // aumenta di 1 il valore della variabile i, ma restituisce
          // il valore che essa aveva prima dell'incremento, cioè 1
i = s + 5; // istruzione di assegnamento; anch'essa è un'espressione
          // e restituisce un "left value", cioè l'indirizzo della
          // variabile i (non restituisce 20)
(10+s)/(i*3) // espressione che restituisce 0
```



Si noti che un'espressione non è un'istruzione. Le istruzioni indicano al compilatore che realizzi un compito e terminano con un punto e virgola, mentre le espressioni specificano un calcolo. In un'istruzione vi possono essere varie espressioni.

### 3.1.2 Operatori

C/C++ sono linguaggi molto ricchi di operatori. L'operazione realmente compiuta da un operatore può dipendere dal tipo dei suoi operandi. In genere, se non si conosce il tipo di operando/i, non si può conoscere il significato dell'espressione. Per esempio,

$m + n$

può significare "somma di numeri interi", "somma di numeri reali" (che può sembrare la stessa cosa ma a livello macchina è un'operazione diversa e restituisce un valore di tipo diverso), concatenazione di stringhe o perfino un'altra operazione, tutto a seconda dei tipi di  $m$  e  $n$ .

Gli operatori possono essere unari, binari o ternari, a seconda che abbiano aritma (ovvero "numero di argomenti") 1, 2 o 3. Gli operatori unari, come gli operatori di pre-incremento o post-incremento ( $++$ ), hanno un solo operando. Gli operatori binari, come somma (+) e sottrazione (-) hanno due operandi. Gli operatori ternari, come l'operatore condizionale ( $? :$ ) hanno tre operandi.

Alcuni operatori possono essere sia unari sia binari; come l'operatore (-) che può fungere anche da operatore di segno se anteposto a un solo operando numerico.

Gli operatori impongono vincoli sul tipo degli operandi. Per esempio, un operatore binario richiede normalmente che i due operandi siano dello stesso tipo, o di tipi che si possano convertire a un tipo comune (per esempio, come vedremo si può convertire un intero in un reale e viceversa; ma non è possibile convertire un tipo puntatore a un reale).

### 3.1.3 Valutazione di espressioni composte

Quando le espressioni contengono diversi operatori si dicono "espressioni composte", e diventano molto importanti la precedenza e l'associatività degli operatori.

**Precedenza:** indica la priorità relativa dell'operatore rispetto ad altri nella stessa espressione composta.

**Associatività:** indica l'ordine in cui si valuta un'espressione composta costituita da più operatori con identica precedenza. L'associatività si dice "destra" se la catena di operatori va valutata da destra verso sinistra (è il caso, per esempio, dell'operatore di assegnamento  $=$ ); si dice "sinistra" se la catena di operatori va invece valutata da sinistra verso destra (è il caso comune degli operatori aritmetici).

#### Esempio 3.2

$m * n + p$	equivale a	$m * (n + p)$	l'operatore * ha maggiore precedenza di +
$m = n = p$	equivale a	$m = (n = p)$	associatività destra
$m + n + p$	equivale a	$(m + n) + p$	associatività sinistra

Gli operatori C++ possono essere:

- di visibilità;
- aritmetici;
- di incremento e decremento;
- di assegnamento;
- relazionali;
- logici;
- bit a bit;
- condizionali;
- di indirizzo o di indirezione;
- di dimensione (sizeof);
- di sequenza ( , );
- di conversione di tipo (static\_cast, reinterpret\_cast, const\_cast, dynamic\_cast);
- di memoria dinamica (new e delete).

Gli operatori C++ possono essere collocati in una tabella di 16 righe a priorità decrescente, dove ciascuna riga contiene operatori della stessa precedenza e della stessa associatività. Si noti che le parentesi tonde hanno la massima precedenza.

Precedenza e associatività determinano quindi come si valutano le espressioni composte, ma i programmatorei possono modificare queste regole inserendo opportunamente parentesi tonde nelle espressioni per trattare ogni sottoespressione come un'unità e poi applicare le normali regole di precedenza.

Priorità	Operatori	Associatività
17	<code>::</code> <code>a++ a--</code> <code>type() type[]</code>	risolutore di visibilità post-incremento, post-decremento "convertitore di tipo" funzionale
16	<code>a()</code> <code>a[]</code> <code>. -&gt;</code> <code>*a --a</code> <code>+a -a</code> <code>! ~</code> <code>(type)</code>	chiamata funzionale indice vettoriale accesso al membro pre-incremento, pre-decremento più e meno unari "not" logico, "not" bit-a-bit "convertitore di tipo" stile "C"
15	<code>*a</code> <code>&amp;a</code> <code>sizeof</code> <code>co_await</code> <code>new new[]</code> <code>delete delete[]</code>	operatore di "indirezione" "indirizzo-di" "dimensione-di" await-espressione (dal C++20) allocazione memoria dinamica deallocazione memoria dinamica

Priorità	Operatori	Associatività
14	<code>. * ~-&gt;</code>	puntatore-a-membro
13	<code>a*b a/b a%b</code>	moltiplicazione, divisione, resto
12	<code>a+b a-b</code>	addizione, sottrazione
11	<code>&lt;&lt; &gt;&gt;</code>	shift a sinistra, shift a destra
10	<code>&lt;&lt;&gt;&gt;</code>	confronto trimodale (dal C++20)
	<code>&lt; &lt;=</code>	minore, minore o uguale
9	<code>&gt; &gt;=</code>	maggiore, maggiore o uguale
8	<code>== !=</code>	uguale, diverso
7	<code>&amp;</code>	"and" bit-a-bit
6	<code>^</code>	"or" esclusivo bit-a-bit
5	<code> </code>	"or" bit-a-bit
4	<code>&amp;&amp;</code>	"and" logico
3	<code>  </code>	"or" logico
	<code>a?:b:c</code>	operatore condizionale
	<code>throw</code>	operatore "throw"
	<code>co_yield</code>	yield-espressione (dal C++20)
	<code>=</code>	assegnamento
2	<code>+= -=</code>	assegnamento composto
	<code>*= /= %=</code>	assegnamento composto
	<code>&lt;&lt;= &gt;&gt;=</code>	assegnamento composto
	<code>&amp;= ^=  =</code>	assegnamento composto
1	<code>,</code>	virgola
		sinistra

**Esempio 3.3**

Qual è il risultato di  $6 + 3 * 4 / 2 + 2$ ?

A seconda dell'ordine con il quale vengono valutate le sottoespressioni aritmetiche si può ottenere un risultato diverso. In questo caso potremmo avere 14 (risultato reale in C++), 36, 20 o 9.

$$6 + 3 * 4 / 2 + 2;$$

—————  
12  
↓      ↓  
6 +    6 + 2      risultato in C++ 14

In C++ si ottiene il risultato vero 14 perché la moltiplicazione e la divisione hanno precedenza più alta e quando coincidono nella stessa espressione si applica prima l'operatore di sinistra, dato che gli operatori aritmetici hanno associatività sinistra.

**Esempio 3.4**

```
cout << 6 + 3 * 4 / 2 + 2;           // visualizza 14
cout << ((6 + (3 * 4)) / 2) + 2;    // visualizza 11
```

```

cout << 6 + 3 * 4 / (2 + 2);           // visualizza 9
cout << (6 + 3) * (4 / 2 + 2);       // visualizza 36

```

Si noti che se più operatori hanno la stessa precedenza, hanno anche la stessa associatività. Per questo, per valutare una sottoespressione composta costituita da operatori diversi ma con identica precedenza si segue l'ordine stabilito dalla loro comune associatività.

## 3.2 Operatore di assegnamento

L'operatore di assegnamento (`=`) conferisce il valore dell'espressione alla sua destra alla variabile situata alla sua sinistra.

*variabile = espressione*

Per esempio:

```
x = 525;
y = 725;
```

Questo operatore è associativo a destra, cosa che permette di eseguire assegnamenti multipli:

`x = y = z = 525;`

equivale a

```
x = (y = (z = 525));
```

cioè alle tre variabili si assegna il valore 525.

A sinistra dell'operatore di assegnamento non può esserci altro che non sia il nome di una variabile.

### Esempio 3.5

```

int i, j, val_m;
const int ci = i;    // inizializzazione, non assegnamento
2040 = val_m;        // errore; 2040 non è il nome di una variabile
i + j = val_m;       // errore; i + j non è il nome di una variabile
ci = val_m;          // errore; ci non è il nome di una variabile

```

In C++ l'operatore di assegnamento si può comporre con operatori aritmetici. Gli operatori di assegnamento composti sono riportati nella Tabella 3.1.

Tabella 3.1 Operatori di assegnamento composto in C++

Simbolo	Uso	Equivalenti	Descrizione
<code>=</code>	<code>a = b</code>	<code>a = a</code>	assegna il valore di <code>b</code> alla variabile <code>a</code>
<code>+=</code>	<code>a += b</code>	<code>a = a + b;</code>	somma <code>a</code> e <code>b</code> e assegna il risultato alla variabile <code>a</code>
<code>-=</code>	<code>a -= b</code>	<code>a = a - b;</code>	sottrae <code>b</code> ad <code>a</code> e assegna il risultato alla variabile <code>a</code>
<code>*=</code>	<code>a *= b</code>	<code>a = a * b;</code>	moltiplica <code>a</code> per <code>b</code> e assegna il risultato alla variabile <code>a</code>
<code>/=</code>	<code>a /= b</code>	<code>a = a / b;</code>	divide <code>a</code> per <code>b</code> e assegna il risultato alla variabile <code>a</code>
<code>%=</code>	<code>a %= b</code>	<code>a = a % b;</code>	mette in <code>a</code> il resto della divisione intera di <code>a</code> per <code>b</code>

Questi operatori composti sono semplici notazioni abbreviate. Per esempio, se si vuole moltiplicare i per 10, si può scrivere

`i = i * 10;`

oppure:

`i *= 10;`

In maniera analoga l'operatore di assegnamento si può comporre anche con operatori bit-a-bit, in inglese *bitwise* (li vedremo in seguito):

`<=> & ^ |= // operatori bit-a-bit`

#### Esempio 3.6

*Equivale a*

<code>conto += 5;</code>	<code>conto = conto + 5;</code>
<code>totale -= sconto;</code>	<code>totale = totale - sconto;</code>
<code>cambio %= 100;</code>	<code>cambio = cambio % 100;</code>
<code>quantita *= c1 + c2;</code>	<code>quantita = quantita * (c1 + c2);</code>
<code>m -= 8;</code>	<code>m = m - 8;</code>
<code>m *= 6;</code>	<code>m = m * 6;</code>

### 3.3 Operatori aritmetici

Gli operatori aritmetici servono per le operazioni aritmetiche di base e ne osservano le ovvie regole di precedenza e associatività (Tabella 3.2).

Se gli operatori + e - non hanno un operando a sinistra (ma un operatore) s'intende che siano unari, con ovvio significato e precedenza più alta. A destra possono avere un valore intero o reale.

```
i = +75;      // associa il valore positivo 75 alla variabile i
i = -75;      // associa il valore negativo 75 alla variabile i
i = i * -75;  // associa alla variabile i il valore che attualmente
               // possiede moltiplicato per il valore negativo 75
```

#### Esempio 3.7

Qual è il risultato dell'espressione:  $7 * 10 - 5 \% 3 * 4 + 9$ ?

Ci sono tre operatori a precedenza più alta \*, % e \*. La valutazione procede da sinistra verso destra, quindi si esegue prima  $7 * 10$ :

Tabella 3.2 Operatori aritmetici

Operatore	Tipi interi	Tipi reali	Esempio
+	somma	somma	$4 + 5$
-	sottrazione	sottrazione	$7 - 3$
*	prodotto	prodotto	$4 * 8$
/	quoziente della divisione intera	divisione	$8 / 5$
%	resto della divisione intera		$12 \% 5$

$70 - 5 \% 3 * 4 + 9$

Rimangono alla precedenza più alta % e \* con associatività a sinistra, dunque si esegue  $5 \% 3$

$70 - 2 * 4 + 9$

e poi  $2 * 4$ , producendo

$70 - 8 + 9$

Le due operazioni restanti sono di uguale precedenza e associatività sinistra; si realizzerà prima la sottrazione:

$62 + 9$

e infine la somma producendo il risultato finale:

71

Come già osservato, si possono utilizzare le parentesi tonde per cambiare l'ordine di valutazione di un'espressione predeterminato dalle regole di precedenza e associatività. Come ovvio dall'aritmetica, si valutano prima le sottoespressioni tra parentesi e poi il resto secondo le normali regole. Se le parentesi sono annidate si eseguono prima le sottoespressioni più interne.

Per esempio, si consideri l'espressione  $(7 * (10 - 5) \% 3) * 4 + 9$ . Si valuta prima la sottoespressione  $(10 - 5)$  producendo

$(7 * 5 \% 3) * 4 + 9$

Poi, si valuta da sinistra a destra la sottoespressione  $(7 * 5 \% 3)$

$(35 \% 3) * 4 + 9$

seguita da

$2 * 4 + 9$

Infine la moltiplicazione, ottenendo

$8 + 9$

e la somma produce il risultato finale

17



Si deve fare attenzione nella scrittura di espressioni che contengano più operazioni per assicurarsi che queste vengano valutate nell'ordine previsto. A volte, anche quando le parentesi tonde non sarebbero necessarie è bene utilizzarle per rendere evidente all'occhio umano quale sia l'ordine di valutazione, così da poter scrivere abbastanza tranquillamente espressioni complicate in termini di molte espressioni più semplici. Ovviamente se le parentesi fossero squilibrate si produrrebbe errore in fase di compilazione.

Tabella 3.3 Operatori di incremento (++) e decremento (--)

	Sintassi	È equivalente a	Cioè è	Restituisce
Preincremento	<code>++x;</code>	<code>x += 1;</code>	<code>x = x + 1;</code>	la variabile <code>x</code>
Postincremento	<code>x++;</code>	<code>x += 1;</code>	<code>x = x + 1;</code>	il valore della variabile <code>x</code>
Predecremento	<code>--x;</code>	<code>x -= 1;</code>	<code>x = x - 1;</code>	la variabile <code>x</code>
Postdecremento	<code>x--;</code>	<code>x -= 1;</code>	<code>x = x - 1;</code>	il valore della variabile <code>x</code>
	<code>y=x++;</code>	<code>y = x;</code> <code>x += 1;</code>		
	<code>y=++x;</code>	<code>x += 1;</code> <code>y = x;</code>		

### 3.4 Operatori di incremento e decremento

Delle molte caratteristiche di C++ ereditate dal C, una delle più utili sono gli operatori di incremento (++) e decremento (--) che, rispettivamente, sommano e sottraggono 1 alla variabile che hanno come argomento. Di ciascuno esistono due importanti varianti, *pre* e *post*, il cui comportamento è descritto nella Tabella 3.3.

Poiché l'operatore di preincremento `++x` restituisce la variabile a cui si applica, esso si può innestare: `++(++x)`; mentre poiché l'operatore di postincremento restituisce il valore della variabile, esso non può essere innestato, cioè `(x++)++` è un errore perché non si può applicare un operatore a un valore (costante) ma solo a una variabile. Lo stesso discorso si applica agli operatori di pre e post decremento.

Se gli operatori ++ e -- si usano come prefissi, l'operazione di incremento/decremento si effettua prima dell'operazione di assegnamento; se invece gli operatori ++ e -- si usano come suffissi, l'assegnamento si effettua in prima dell'incremento/decremento.

#### Esempio 3.8

```
int a = 1, b, c;
b = a++;      //postincremento; b vale 1 ed a vale 2
c = a--;      //postdecremento; c vale 2 ed a vale 1

int a = 1, b, c;
b = ++a;      //preincremento; b vale 2 ed a vale 2
c = --a;      //predecremento; c vale 1 ed a vale 1
```

#### Esempio 3.9

Questo programma illustra il funzionamento degli operatori di incremento/decremento.

```

int main()
{
    int m = 46, n = 75;
    cout << "m = " << m << "n = " << n << endl;
    ++m;
    --n;
    cout << "m = " << m << "n = " << n << endl;
    m++;
    n--;
    cout << "m = " << m << "n = " << n << endl;
    return 0;
}

```

produce a schermo:

```

m = 46,   n = 75
m = 46,   n = 74
m = 47,   n = 73

```

### Esempio 3.10

Differenze tra operatori di preincremento e postincremento.

```

int main()
{
    int m = 99, n;
    n = ++m;
    cout << "m = " << m << "n = " << n << endl;
    n = m++;
    cout << "m = " << m << "n = " << n << endl;
    cout << "m = " << m++ << endl;
    cout << "m = " << m << endl;
    cout << "m = " << ++m << endl;
    return 0;
}

```

### Esecuzione

```

m = 100,   n = 100
m = 101,   n = 100
m = 101
m = 102
m = 103

```

### Esempio 3.11

Ordine di valutazione non predicibile.

```

int main()
{

```

```

int n = 5, t;
t = ++n * --n; // ERRORE ++n e --n non restituiscono valori
cout << "n = " << n << ", t = " << t << endl;
cout << ++n << " " << ++n << " " << ++n << endl;
}

```

## 3.5 Operatori relazionali

ANSI/ISO C++ supporta il tipo `bool` che può assumere solo i due valori `false` e `true`. Un'espressione si dice *booleana* se restituisce un valore di tipo `bool`.

Per rappresentare i valori `false` e `true` il C++ utilizza il tipo `int`: lo 0 per `false` e qualunque altro intero per `true`.

**false** zero  
**true** diverso da zero

Operatori come `>` e `==` che verificano una relazione tra due operandi si chiamano operatori relazionali e si utilizzano in espressioni della forma

*espressione<sub>1</sub>* *operatore\_relazionale* *espressione<sub>2</sub>*

*espressione<sub>1</sub>*, *espressione<sub>2</sub>* espressioni compatibili C++  
*operatore\_relazionale* un operatore della Tabella 3.7

Gli operatori relazionali si usano normalmente nelle istruzioni condizionali (`if`) e in quelle iterative (`while`, `for`) e servono a verificare una condizione. La Tabella 3.4 mostra gli operatori relazionali che si possono applicare a operandi di qualunque tipo di dato standard: `char`, `int`, `float`, `double` ecc.

Quando si utilizzano gli operatori in un'espressione, l'operatore relazionale produce 0 o 1 a seconda del risultato del confronto. Per esempio, se si scrive

`c = 3 < 7;`

la variabile `c` prende il valore 1 perché 3 è minore di 7 e quindi `3<7` è vera.

Tabella 3.4 Operatori relazionali in C++

Operatore	Significato	Esempio
<code>==</code>	<i>Uguale a</i>	<code>a == b</code>
<code>!=</code>	<i>Non uguale a</i>	<code>a != b</code>
<code>&gt;</code>	<i>Maggiore di</i>	<code>a &gt; b</code>
<code>&lt;</code>	<i>Minore di</i>	<code>a &lt; b</code>
<code>&gt;=</code>	<i>Maggiore o uguale di</i>	<code>a &gt;= b</code>
<code>&lt;=</code>	<i>Minore o uguale di</i>	<code>a &lt;= b</code>

**ATTENZIONE**

Un errore tipico, perfino tra programmati esperti, è quello di confondere l'operatore di assegnamento (`=`) con l'operatore di uguaglianza (`==`).

**Esempio 3.12**

Se `s`, `a`, `b` e `c` sono di tipo `double`, `numero` è `int` e `iniziale` è di tipo `char`, le seguenti espressioni booleane sono valide:

```
s < 5.76
b * b >= 5.0 * a * c
numero == 100
iniziale != '5'
```

Gli operatori relazionali si utilizzano normalmente per confrontare valori numerici. Così, se

`s = 3.1`

l'espressione

`s < 7.5`

produce il valore `true`. In modo simile, se

`numero = 27`

l'espressione

`numero == 100`

produce il valore `false`.

I caratteri si confrontano in base ai rispettivi codici (ASCII o UNICODE).

'A' < 'C' è `true` perché il codice 65 di A è minore del codice 67 di C

'a' < 'c' è `true` perché il codice 97 di a è minore del codice 99 di c

'b' < 'B' è `false` perché il codice 98 di b non è minore del codice 66 di B

Gli operatori relazionali hanno associatività sinistra e minore precedenza rispetto agli operatori aritmetici. Per esempio,

`m + 5 <= 2 * n`

equivale a

`(m + 5) <= (2 * n)`

**ATTENZIONE**

Per valutare una successione di diseguaglianze come

`s < z < e`

si deve utilizzare la seguente espressione

`(s < z) && (z < e)`

**Tabella 3.5** Tabelle di verità degli operatori logici

a	b	!	$\&&$	$\ $	a    b	a && b	(a    b) && (a && b)
true	true	false	true	true	false	true	false
true	false	false	false	true	false	false	false
false	true	true	false	true	true	false	false
false	false	false	false	false	false	false	false

## 3.6 Operatori logici

Gli *operatori logici* (anche detti *booleani*, in onore di George Boole, creatore dell'omonima algebra) sono: ! (not),  $\&&$  (and) e  $\|$  (or).

L'operatore ! produce false se il suo operando è true e viceversa.

L'operatore  $\&&$  produce true se e solo se entrambi gli operandi sono true.

L'operatore  $\|$  produce false se e solo se entrambi gli operandi sono false.

La Tabella 3.5 mostra le cosiddette tabelle di verità per questi tre operatori logici.

Gli operatori logici si utilizzano nelle istruzioni condizionali (if) e iterative (while o for) che vedremo poi. Anticipiamo qualche esempio:

```
if ((a < b) && (c > d)) cout << "I risultati non sono validi";
```

ovvero, se la variabile a è minore di b e, allo stesso tempo, c è maggiore di d, allora visualizza il messaggio: I risultati non sono validi.

```
if (( vendite > 50000) || (ore < 100)) bonus = 100000;
```

ovvero, se la variabile vendite ha valore maggiore di 50000 oppure la variabile ore è minore di 100, allora si assegnerà alla variabile bonus il valore 100000.

```
if (!( vendite < 2500)) bonus = 12500;
```

ovvero, se vendite è maggiore o uguale a 2500, si assegnerà a bonus il valore 12500.

L'operatore ! ha priorità più alta di  $\&&$ , che a sua volta ha precedenza su  $\|$ .

L'associatività è sinistra per tutti.

Gli operatori matematici hanno precedenza sugli operatori relazionali, e gli operatori relazionali hanno precedenza sugli operatori logici.

La seguente istruzione:

```
if ( vendite < sal_min * 3 && anni > 10 * iva)...
```

equivale a

```
if (( vendite < (sal_min * 3)) && (anni > (10 * iva)))...
```

In C++ gli operandi a sinistra di  $\&&$  e  $\|$  vengono sempre valutati per primi; se il loro valore è già sufficiente per determinare il valore dell'espressione, l'operando destro non viene valutato, per risparmiare tempo CPU. Ciò accade solo in due casi: quando il primo operando di  $\&&$  è false (in questo caso tutta

l'espressione è sicuramente false) e quando il primo operando di `||` è true (in questo caso tutta l'espressione è sicuramente true). Questa proprietà si dice **valutazione in cortocircuito**.

#### Esempio 3.13

*Supponiamo che si debba valutare l'espressione:*

`(s >= 0.0) && (sqrt(s) >= 2)`

se l'operando (`s >= 0.0`) è false (`s` è negativo) tutta l'espressione è sicuramente false, pertanto non è necessario valutare il secondo operando, la qualcosa evita di calcolare la radice quadrata di numeri (`s`) negativi.

La valutazione in **cortocircuito** ha due benefici importanti:

1. l'espressione *booleana* al primo membro può agire come sentinella per evitare di compiere un'operazione potenzialmente insicura al secondo membro;
2. l'espressione *booleana* al primo membro può far risparmiare una considerevole quantità di tempo nella valutazione di condizioni complesse al secondo membro.

#### Esempio 3.14

*I benefici citati si apprezzano nell'espressione booleana*

`(n != 0) && (s < 1.0 / n)`

che evita di dividere per zero il secondo membro, dato che se `n` è 0, allora la prima espressione

`n != 0`

è false e la seconda espressione

`s < 1.0 / n`

non viene valutata.

Lo stesso vantaggio si ha valutando l'espressione booleana duale della precedente:

`(n == 0) || (s >= 1.0 / n)`

dato che se `n` è 0, la prima espressione

`n == 0`

è true e non c'è bisogno quindi di valutare la seconda espressione (`s >= 1.0 / n`).

Si può assegnare direttamente un valore di tipo `bool` a una variabile di tipo `bool`.

**Esempio 3.15**

```
bool Maggiorenne = true;
Maggiorenne = (età > 18);
```

**Esempio 3.16**

*range* e *lettera* sono due variabili booleane. *range* è *true* se e solo se il valore di *n* va da -100 a 100; la variabile *lettera* è *true* se e solo se *car* è una lettera maiuscola o minuscola

```
bool range = (n > -100) && (n < 100);
bool lettera = (('A' <= car) && (car <= 'Z')) ||
    (('a' <= car) && (car <= 'z'));
```

### 3.7 Operatori di manipolazione dei bit

Gli operatori di manipolazione dei bit (*bitwise operators*) eseguono operazioni logiche sui bit di un operando oppure sulla coppia di bit in corrispondenza della posizione degli operandi. Le operazioni sono generalmente efficientissime perché eseguite direttamente da un singola istruzione del microprocessore. Questi operatori bit-a-bit si applicano soltanto ai tipi char e int (non ai reali). Le seguenti tabelle di verità descrivono le operazioni degli operatori bit-a-bit logici (Tabella 3.6).

**Esempio 3.17**

- (01001001 & 01001110) produce 01001000
- (0000000010011000 | 0000000000000101) produce 0000000010011101
- (0000000001010011 ^ 0000000011001100) produce 0000000010011101
- (~0000000001000001) produce 0000000001011110

Gli operatori bit-a-bit di spostamento verso destra (>) e verso sinistra (<)

```
variabile >> numero_di_posizioni;
variabile << numero_di_posizioni;
```

operano uno scivolamento verso destra o verso sinistra di tutti i bit dell'operando *variabile*, di *numero\_di\_posizioni*. I bit che entrano in *variabile* sono tutti a 0, mentre quelli che "fuoriescono" sono perduti.

**Tabella 3.6** Operatori logici bit a bit

		AND	OR inclusivo	OR esclusivo (XOR)	Negazione
a	b	a & b	a   b	a ^ b	~a
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

**Esempio 3.16**

```
int main()
{
    unsigned int i = 10, j = 11;
    cout << (i & j) << endl;
    cout << (i | j) << endl;
    cout << (i ^ j) << endl;
    cout << ~i << endl;
    cout << (i << 3) << endl;
    cout << (i >> 3) << endl;
}
```

produce a schermo:

```
10
11
1
4294967285
1
80
```

la seguente tabella spiega questi risultati:

Espressione	Valore in binario	Valore in decimale
i	00000000 00000000 00000000 00001010	10
j	00000000 00000000 00000000 00001011	11
i & j	00000000 00000000 00000000 00001010	10
i   j	00000000 00000000 00000000 00001011	11
i ^ j	00000000 00000000 00000000 00000001	1
-i	11111111 11111111 11111111 11110101	4.294.967.285
i >> 3	00000000 00000000 00000000 00000001	1
i << 3	00000000 00000000 00000000 01010000	80

Esistono anche operatori bit-a-bit composti (Tabella 3.7).

### 3.8 Operatore condizionale

L'operatore condizionale è un operatore ternario che restituisce un valore che dipende dal valore dell'espressione booleana al primo membro. È associativo a destra.

Tabella 3.7 Operatori bitwise composti

Simbolo	Uso	Descrizione
<<=	a <<= b	Sposta a sinistra b bit e assegna il risultato ad a
>>=	a >>= b	Sposta a destra b bit e assegna il risultato ad a
&=	a &= b	Assegna ad a il valore a & b
^=	a ^= b	Assegna ad a il valore a ^ b
=	a  = b	Assegna ad a il valore a   b

Il formato dell'operatore condizionale è:

*espressione\_booleana* ? *se\_vera* : *se\_falsa*

Si valuta *espressione\_booleana*, se è true viene restituito il valore dell'espressione *se\_vera* altrimenti viene restituito il valore dell'espressione *se\_falsa*.

#### Esempio 3.19

```
n >= 0 ? 1 : -1 // restituisce 1 se n è positivo, -1 se è negativo
m >= n ? m : n //restituisce il maggiore tra m ed n
```

### 3.9 Operatore virgola

L'operatore *virgola* (,) permette di combinare due o più espressioni che verranno valutate da sinistra verso destra. L'espressione più a destra restituisce il risultato globale.

*espressione\_1, espressione\_2, espressione\_3, ..., espressione\_n*

Per esempio, in

```
int i = 10, j = 25;
```

viene prima dichiarata e inizializzata la i e poi la j.

#### Esempio 3.20

```
int main()
{
    int i, j;
    j = (i = 12, i + 8);
    cout << j << endl;
}
```

produce a schermo:

20

### 3.10 Operatore sizeof

L'operatore *sizeof* serve per conoscere la dimensione in byte di un tipo di dato o di una variabile. Il formato dell'operatore unario è

*sizeof(name\_variabile tipo\_dato)*  
*sizeof espressione*

L'operatore può essere utile a migliorare la portabilità dei programmi. Infatti diverse architetture hardware possono dimensionare diversamente i vari tipi di dato e l'uso opportuno di questo operatore permette di gestire queste differenze. L'operatore viene valutato in fase di compilazione, quando il compilatore sostituisce ogni sua occorrenza nel programma con il valore *unsigned* da esso restituito.

**Esempio 3.21**

```
int main() {
    int i=10, j=11;
    cout << sizeof (char) << ',';
    cout << sizeof (unsigned int) << ',';
    cout << sizeof (float) << ',';
    cout << sizeof (double) << ',';
    cout << sizeof (i) << ',';
    cout << sizeof (i + j) << endl;
}
```

produce a schermo:

```
1, 4, 4, 8, 4, 4
```

### 3.11 Conversioni di tipo

In C/C++ si può convertire un valore di un tipo in un valore di un altro tipo. Tale azione si dice *conversione di tipo* (casting). Questa caratteristica è necessaria in numerose espressioni aritmetiche e assegnazioni in cui gli operatori binari richiedono operandi dello stesso tipo ma si trovano a operare su operandi di tipo diverso. Inoltre, quando si chiama una funzione, il tipo dell'argomento passatole deve coincidere con il tipo dell'argomento formale che la funzione si aspetta (vedremo poi); se non è così, il tipo dell'argomento passato si converte nel tipo atteso. C/C++ realizza automaticamente parecchie conversioni (*conversioni implicite*), ma ha anche operatori di conversione che permettono al programmatore di impostarle esplicitamente (*conversioni esplicite*). Tipicamente, C/C++ converte implicitamente i tipi quando:

- si assegna un valore di un tipo a una variabile di un altro tipo aritmetico;
- si combinano in espressioni di tipi diversi;
- si passano argomenti a funzioni.

#### Conversione implicita

Gli operandi di tipo a precisione più bassa si convertono nei tipi a precisione più alta (*promozione di tipo*). Per esempio, in:

```
pi = 3.141592 + 3;
```

si assegna a pi la somma di un intero e di un reale, ovvero valori di tipi differenti, quindi pi potrebbe assumere valore 6 o 6.141592 (che sarebbe più logico) a seconda che venga compiuta la "somma fra interi" (troncando 3.141592 a 3) o la "somma fra reali" (convertendo 3 nel reale 3.0). Il compilatore sceglie preservando, se possibile, la precisione, quindi convertendo il numero intero in numero reale (ottenendo 6.141592).

In:

```
int n = 0.0;
```

il compilatore deve assegnare il valore reale 0.0 a n, che è una variabile di tipo intero. Quando i tipi sinistro e destro di un assegnamento differiscono,

il valore di destra viene convertito al tipo della variabile a sinistra, ma se ciò può far perdere informazione (come in questo caso, in cui il valore reale si converte a int) il compilatore produrrà un messaggio di allerta (*warning*) che avviserà il programmatore.

### Rapporto

1. In un'espressione aritmetica, gli operatori binari richiedono operandi dello stesso tipo. Se questi non coincidono, il compilatore può compiere automaticamente conversioni di tipo implicite.
2. Nel caso di chiamata a una funzione, il tipo dell'argomento attuale deve corrispondere con il tipo dell'argomento formale, altrimenti il compilatore prova a convertire il tipo del parametro attuale a quello del parametro formale.
3. C/C++ ha operatori di *casting* di tipi (che permettono cioè di convertire i tipi esplicitamente), ma il compilatore può convertire automaticamente il tipo in certi casi ovvi.

### Conversioni aritmetiche

Le *conversioni aritmetiche* assicurano che gli operandi di un operatore binario aritmetico o logico si convertano a un tipo comune prima che si valuti l'espressione, e questo tipo comune sarà il tipo del risultato dell'espressione. La regola è quella predetta della *promozione di tipo*, cioè il compilatore cerca di convertire senza perdere informazione, convertendo il tipo dell'operando a precisione più bassa in quello del tipo a precisione più alta. La graduatoria di precisione, dalla più alta a quella più bassa, riguardo i principali tipi predefiniti è questa:

long double, double, float, long, int, short, char, bool

I tipi più piccoli di int (char, signed char, unsigned char, short e unsigned short) sono promossi a int. Il tipo bool si promuove a int nel senso che false si promuove a 0 e true a 1.

Quando un tipo unsigned è implicato in un'espressione, le regole di conversione preservano il valore degli operandi, ma in questi casi le dimensioni relative dipendono dalla macchina.

### Esempio 3.22

```
int main()
{
    bool flag = true;
    char car = 'a';
    short num_short = 2;
    int num_int = 1234567890;
    long num_long = 4294967296;
    float num_float = 3.14159265358979323846;
    double num_double = 3.14159265358979323846264338327950288419716939;
    //esempi di conversioni automatiche di tipi
    cout << num_long + num_int << endl;      // da int a long
```

```

cout << num_double + num_long << endl; // da long a double
cout << num_double + num_float << endl; // da float a double
cout << num_short + num_float << endl; // da short ad int e poi a float
cout << flag + num_int << endl; // da bool a int
cout << car + num_float << endl; // da car ad int e poi a float
cout << num_int + num_unsigned << endl; // da int ad unsigned OVERFLOW
num_int = num_unsigned; // da unsigned ad int OVERFLOW
cout << num_int << endl; // mostra OVERFLOW precedente
}

```

produce a schermo:

```

5529535186
4.29497e+09
6.28319
5.14159
1234567891
100.142
1234567889
-1

```

### Conversioni esplicite

Con il termine casting s'intende la conversione esplicita di tipi. C++ offre nuove forme di casting rispetto al C.

#### Notazioni compatibili con il C

- `tipo_in_cui_convertire(espressione)` //notazione stile funzione
- `(tipo_in_cui_convertire) espressione;` //notazione stile C

#### Notazioni compatibili con lo standard ANSI/ISO C++

C++ supporta i seguenti operatori di casting, ma si tratta di usi specifici che saranno trattati nei capitoli a seguire.

- `const_cast <tipo_in_cui_convertire> (espressione)`
- `dinamic_cast <tipo_in_cui_convertire> (espressione)`
- `reinterpret_cast <tipo_in_cui_convertire> (espressione)`
- `static_cast <tipo_in_cui_convertire> (espressione)`

#### Esempio 3.23

```

int main()
{
    int i = 7, j = 2;
    double m = i/j; // divisione fra interi -> troncatura m == 3.0
    cout << "m vale " << m << endl;
    m = (double) (i/j); // div fra interi m == (double) 3 == 3.0
    cout << "m vale ancora " << m << endl;
    m = (double) i/j; // i è conv in double quindi div fra double -> m double
    cout << "m adesso vale " << m << endl;
}

```

```
m = i/static_cast <float> (j); // j conv float quindi div fra float -> m float
cout << "m adesso vale ancora " << m << endl;
m = (float) i/ (double) j; // pura esagerazione - non serve
cout << "m adesso vale ancora invariabilmente " << m << endl;
}
```

produce a schermo:

```
m vale 3
m vale ancora 3
m adesso vale 3.5
m adesso vale ancora 3.5
m adesso vale ancora invariabilmente 3.5
```

### CONSIDERAZIONI

In questo capitolo abbiamo esaminato le regole di precedenza e associatività dei differenti operatori quando si combinano in espressioni. In dettaglio abbiamo esaminato gli operatori:

- di assegnamento =
- aritmetici +, -, \*, / e %
- di pre/post incremento e pre/post decremento (C++ permette di applicare questi operatori a variabili di tipo char, int e perfino float e double)
- relazionali >, >=, <, <=, ==, !=

- logici &&, ||, !
- di manipolazione di bit che realizzano operazioni bit-a-bit (bitwise), &, |, ^, !, << e >>
- di assegnamento composti
- virgola
- condizionale
- sizeof

Abbiamo anche introdotto il casting, che permette di forzare la conversione di tipi di un'espressione.

### ESERCIZI

- Espressione
- Operatore
- Priorità/precedenza
- Associatività
- Tipi di dato predefiniti
- Conversioni di tipi
- Conversione esplicita e implicita
- Valutazione in cortocircuito
- Incremento/decremento
- Manipolazione di bit

## ESERCIZI

**Esercizio 1** Scrivere un programma che valuti le seguenti espressioni aritmetiche sistemando il risultato in una variabile del tipo opportuno facendo più prove:

- $15/12$
- $15 \% 12$
- $24/12$
- $24 \% 12$
- $123/100$
- $200 \% 100$
- $10 * 14 - 3 * 2$
- $-4.0 + 5 * 2$
- $13 - (24 + 2 * 5)/4 \% 3$
- $(4 - 40/5) \% 3$
- $4 * (3 + 5) - 8 * 4 \% 2 - 5$
- $-3 * 10 + 4 * (8 + 4 * 7 - 10 * 3)/6$

**Esercizio 2** Scrivere un programma che calcoli il valore delle seguenti espressioni aritmetiche e inserisca il risultato in una variabile del tipo opportuno: per l'elevamento a potenza si utilizzi la funzione della libreria

```
<cmath>
double pow (double base, double exponent);
```

- $\frac{x}{y} + 1$
- $\frac{b}{c+d}$
- $\frac{xy}{1-4x}$
- $\frac{x+y}{x-y}$
- $(a+b)\frac{c}{d}$
- $\frac{xy}{mn}$
- $x + \frac{y}{z}$
- $[(a+b)^2]^2$
- $(x+y)^2$

**Esercizio 3** Scrivere un programma per scambiare i valori di due variabili  $x$  e  $y$  di un certo tipo di dato.

**Esercizio 4** Scrivere un programma che legga da tastiera due interi nelle variabili  $x$  e  $y$ , e, poi, visualizzi a schermo i valori di:

- $x/y$
- $X \% Y$

Eseguire il programma varie volte con diverse coppie di interi come input.

**Esercizio 5** Una temperatura data in gradi Celsius (centigradi) può essere trasformata in una temperatura equivalente Fahrenheit con la seguente formula:

$$F = \frac{9}{5}C + 32$$

Scrivere un programma che legga la temperatura in gradi centigradi e la converta a gradi Fahrenheit.

**Esercizio 6** La relazione tra i lati ( $a, b$ ) di un triangolo e l'ipotenusa ( $h$ ) viene data dalla formula:

$$a^2 + b^2 = h^2$$

Scrivere un programma che legga la lunghezza dei lati e calcoli l'ipotenusa.

**Esercizio 7** L'area di un triangolo i cui lati sono  $a, b, c$  si può calcolare mediante la formula:

$$A = \sqrt{p(p-a)(p-b)(p-c)}$$

dove  $p = (a + b + c)/2$ . Scrivere un programma che legga le lunghezze dei tre lati di un triangolo e ne calcoli l'area.

**Esercizio 8** Scrivere un programma che legga l'ora in notazione di 24 ore e la restituiscia in notazione di 12 ore. Per esempio, se l'input è 13:45, l'output sarà

1:45 PM

Il programma chiederà all'utente di introdurre esattamente cinque caratteri. Così, per esempio, le nove si introdurrebbe come

09:00

**Esercizio 1** Scrivere un programma che accetti date scritte nel modo usuale e le visualizzi come tre numeri. Per esempio, l'input

15 Febbraio 1989

produrrà l'output

15 2 1989

**Esercizio 2** Scrivere un programma che accetti un numero di tre cifre scritto in lettere e poi lo visualizzi come un valore di tipo intero. L'input si deve terminare con un punto. Per esempio, l'input

duecento venticinque

produrrà l'output

225

**Esercizio 3** Scrivere un programma che legga il raggio di un cerchio e poi visualizzi: circonferenza del cerchio, area del cerchio e diametro del cerchio.

**Esercizio 4** Scrivere un programma che accetti un anno scritto in cifre arabe e visualizzi l'anno scritto in numeri romani, dentro l'intervallo 1000 a 2000.

*Nota:* Ricordi che V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000.

V = 4      XL = 40      CM = 900

MCM = 1900    MCML = 1950    MCMX = 1960    MCMXL = 1940  
MCMLXXXIX = 1989

**Esercizio 5** Un file di dati contiene le quattro cifre, A, B, C, D, di un intero positivo N: si vuole arrotondare N al centinaio più prossimo e visualizzare l'output. Per esempio, se A è 2, B è 3, C è 6 e D è 2, allora N sarà 2.362 e il risultato arrotondato sarà 2.400. Se N è 2.342, il risultato sarà

2.300, e se N = 2.962, allora il numero sarà 3.000. Progettare il programma corrispondente.

**Esercizio 6** Un file contiene due date nel formato giorno (1 a 31), mese (1 a 12) e anno (intero di quattro cifre), corrispondenti alla data di nascita e alla data attuale, rispettivamente. Scrivere un programma che calcoli e visualizzi l'età dell'individuo. Se è la data di un bebè (meno di un anno di età), l'età si deve dare in mesi e giorni; in caso contrario, l'età si calcolerà in anni.

**Esercizio 7** Scrivere un programma che determini se un anno è bisestile. Un anno è bisestile se è multiplo di 4 (per esempio, 1984). Tuttavia, gli anni multipli di 100 sono bisestili solo quando sono anche multipli di 400 (per esempio, 1800 non è bisestile, mentre 2000 sì lo è).

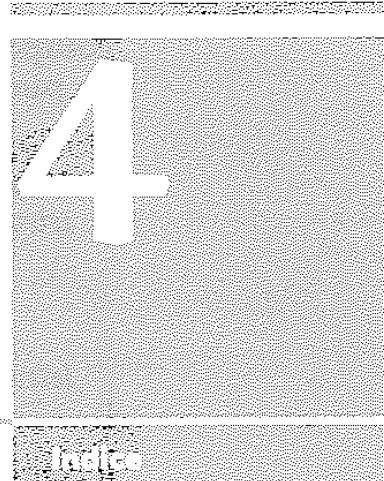
**Esercizio 8** Scrivere un programma che calcoli il numero di giorni di un mese, dati i valori numerici del mese e dell'anno.

**Esercizio 9** Si vuole calcolare il salario netto settimanale dei lavoratori di un'impresa che segue le norme elencate di seguito.

- Ore settimanali lavorate < 38 a un tasso dato.
- Ore extra (38 o più) a un tasso del 50% superiore all'ordinario.
- Gravame 0%, se il salario lordo è minore o uguale a 300 euro.
- Gravame 10%, se il salario lordo è maggiore a 300 euro.

**Esercizio 10** Costruire un programma che indichi se un numero introdotto da tastiera è positivo, uguale a zero, o negativo.

# La programmazione strutturata



- |            |   |             |  |
|------------|---|-------------|--|
| <b>4.1</b> | Strutture di controllo                      | <b>4.8</b>  | Istruzione while                       |
| <b>4.2</b> | Istruzione if                               | <b>4.9</b>  | Istruzione for                         |
| <b>4.3</b> | Istruzione condizionale<br>doppia: if else  | <b>4.10</b> | Precauzioni nell'uso del for           |
| <b>4.4</b> | Istruzioni if_else annidate                 | <b>4.11</b> | Istruzione do while                    |
| <b>4.5</b> | Istruzione switch                           | <b>4.12</b> | Confronto fra while, for e do<br>while |
| <b>4.6</b> | if else e operatore<br>condizionale ( ? : ) | <b>4.13</b> | Progetto di un'istruzione<br>ciclica   |
| <b>4.7</b> | Frequenti errori di<br>programmazione       | <b>4.14</b> | Cicli annidati                         |

## INTRODUZIONE

I programmi visti finora s'eseguono in modo *sequenziale*, cioè, un'istruzione dopo l'altra, dalla prima fino all'ultima istruzione, e ognuna viene eseguita una sola volta. Questa *programmazione sequenziale* risolve problemi molto semplici, ma per problemi normali bisogna avere la capacità di controllare e variare il flusso d'esecuzione del programma mediante altre *strutture di controllo*.

La cosiddetta "Programmazione Strutturata" contempla tre tipi di

strutture di controllo: oltre alla citata struttura *sequenziale*, vi sono quelle di *selezione* e quelle di *ripetizione*. Le *strutture selettive*, o *condizionali*, determinano quale sequenza di istruzioni eseguire in funzione della valutazione di una certa condizione di selezione; quest'ultima potrà essere booleana (nell'istruzione if) o multivaleore (nell'istruzione switch). Le *strutture di controllo iterative*, o *cicliche* (while, for e do while) consentono invece la ripetizione di una sequenza di istruzioni.

## 4.1 Strutture di controllo

Le **strutture di controllo** determinano il flusso d'esecuzione di un programma. Esse permettono di combinare più istruzioni in una semplice unità logica con un punto d'ingresso e uno d'uscita.

La **programmazione strutturata** contempla tre tipi di strutture di controllo: *sequenza*, *selezione* e *ripetizione*. Fino a ora è stata utilizzata solo la struttura sequenziale, cioè il concetto di **blocco** di istruzioni, o **istruzione**

**composta.** Si tratta semplicemente di una sequenza di istruzioni racchiuse tra parentesi graffe.

```
[  
    istruzione1;  
    istruzione2;  
  
    .  
  
    .  
  
    istruzioneN;  
]
```

Il controllo (della CPU) passa in maniera sequenziale dalla prima istruzione all'ultima.

Tuttavia, anche la soluzione di semplici problemi richiede spesso di valutare una certa condizione per scegliere fra percorsi alternativi.

## 4.2 Istruzione if

In tutti i linguaggi procedurali la struttura di selezione principale è l'istruzione if. Normalmente essa ha due forme possibili. In C++ il formato più semplice ha la seguente sintassi:

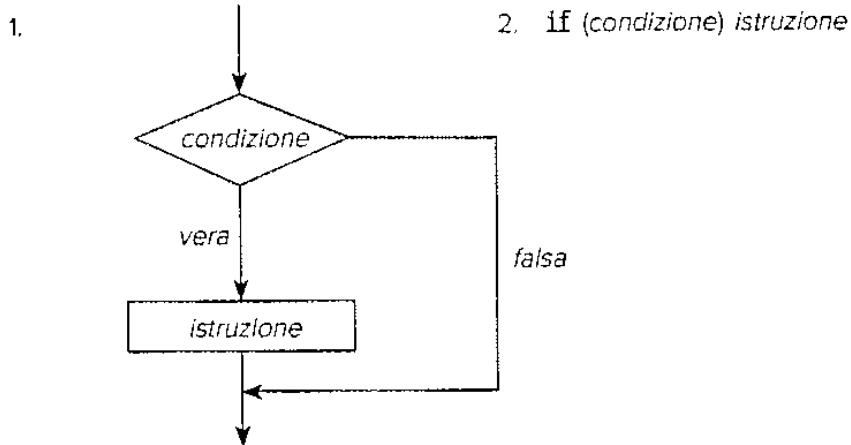
```
if (espressione booleana) istruzione
```

L'istruzione if valuta l'*espressione booleana* fra le parentesi; se è vera, esegue *istruzione* (che può essere un'istruzione composta), altrimenti non fa nulla. Dopodiché, in entrambi i casi l'esecuzione del programma continua con l'istruzione successiva all'if. La Figura 4.1 mostra il *diagramma di flusso* (flow chart) dell'if.

### Esempio 4.1

*Scrivere un programma per determinare se, dati due numeri letti in input, il primo è divisibile per il secondo.*

```
int main()  
{  
    int n, d;  
    cout << "Per favore, introduca due interi: ";  
    cin >> n >> d;  
    if (n%d == 0) cout << n << " è divisibile per " << d << endl;  
}
```



**Figura 4.1** Diagramma di flusso dell'`if`. Il diagramma illustra il flusso di controllo per l'esecuzione di un if.

### Esecuzione

Per favore, introduca due interi: **36 4**  
**36** è divisibile per **4**

Questo programma legge due numeri interi e verifica se il resto della divisione di `n` per `d` (cioè `n%d`) è zero, perché in tal caso `n` è divisibile per `d`. Se la condizione è verificata viene emesso un opportuno messaggio sul flusso di output.

### Esempio 4.2

*Scrivere un programma per determinare se un certo numero letto in input è minore, maggiore o uguale a zero.*

```

int main()
{
    float num;
    cout "Per favore, immetta un numero positivo o negativo: ";
    cin >> num;
    if (num > 0) cout << num << " è maggiore di zero" << endl;
    if (num < 0) cout << num << " è minore di zero" << endl;
    if (num == 0) cout << num << " è uguale a zero" << endl;
}
  
```

### Esecuzione

Per favore, immetta un numero positivo o negativo: **10.45**  
**10.45** è maggiore di zero

### Esempio 4.3

*Scrivere un programma per visualizzare il valore assoluto di un numero letto dalla tastiera.*

```

int main() {
    cout << "Per favore, immetta un numero intero: ";
    int valore;
    cin >> valore;
    if (valore < 0) valore = -valore;
    cout << "Il valore assoluto è " << valore << endl;
    return 0;
}

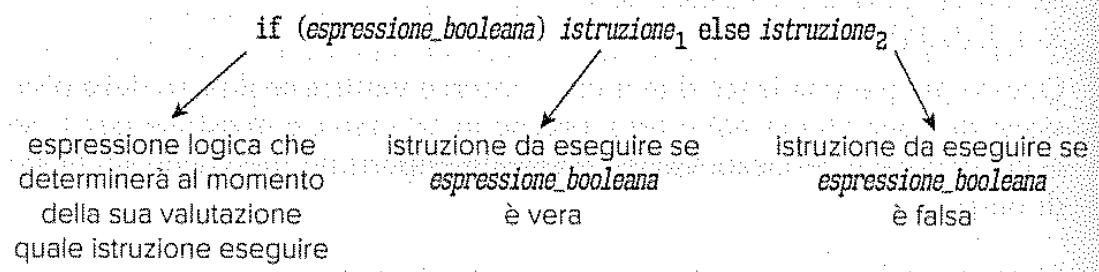
```

### Esecuzione

Per favore, immetta un numero intero: **-11**  
Il valore assoluto è **11**

### 4.3 Istruzione condizionale doppia: if else

La seconda forma dell'istruzione condizionale è l'if else. Questo formato dell'istruzione if ha la seguente sintassi:



*istruzione*<sub>1</sub> e *istruzione*<sub>2</sub> possono anche essere istruzioni composte. L'istruzione if else valuta *espressione booleana*. Se è vera esegue *istruzione*<sub>1</sub>, altrimenti esegue *istruzione*<sub>2</sub>. La Figura 4.2 mostra la semantica dell'istruzione if else.

#### Esempio 4.4

```

if (Voto >= 6) cout << "Sufficiente" << endl;
else cout << "Insufficiente" << endl;

```

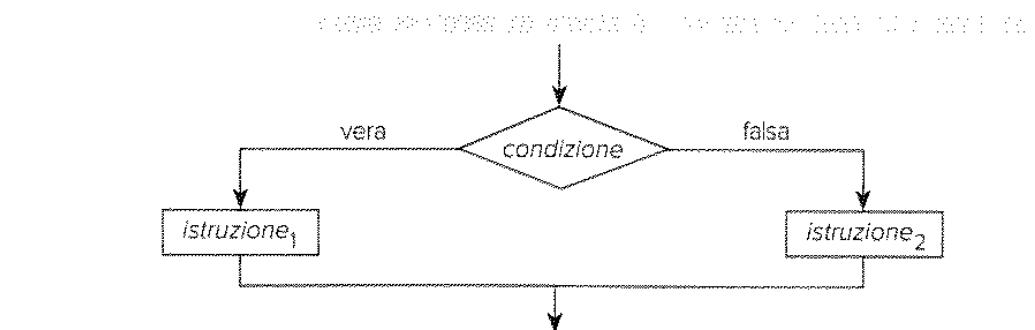


Figura 4.2

Diagramma di flusso della rappresentazione di un'istruzione if else.

**Esempio 4.5**

*Scrivere un programma per determinare se, dati due numeri letti in input, il primo è divisibile per il secondo (come l'Esempio 4.1 ma con la clausola else).*

```
int main()
{
    int n, d;
    cout << "Per favore, introduca due interi: ";
    cin >> n >> d;
    if (n%d == 0) cout << n << " è divisibile per " << d << endl;
    else cout << n << "non è divisibile per " << d << endl;
}
```

**Esecuzione**

Per favore, introduca due interi: 36 5  
36 non è divisibile per 5

**Esempio 4.6**

*Calcolare il maggiore di due numeri letti dalla tastiera e visualizzarlo sullo schermo.*

```
int main()
{
    int x, y;
    cout << "Per favore, introduca due interi: ";
    cin >> x >> y;
    cout << "Il maggiore fra " << x << " e " << y << " è ";
    if (x > y) cout << x << endl;
    else cout << y << endl;
}
```

**Esecuzione**

Per favore, introduca due interi: 13 54  
Il maggiore fra 13 e 54 è 54

**4.4 Istruzioni if else annidate**

Le istruzioni if hanno come condizione un'espressione booleana e quindi implicano una o due alternative. Se abbiamo più alternative possiamo "annidare" più istruzioni if una dentro l'altra.

Un'istruzione if è annidata quando si trova dentro uno dei due rami di un'altra istruzione if.

**Esempio 4.7**

*A seconda del valore della variabile intera x incrementare il contatore dei numeri positivi num\_pos o di quelli negativi num\_neg o degli zeri num\_zeri.*

if (x > 0) num_pos++;	
else	if (x < 0) num_neg++;
	else num_zeri++;

L'istruzione if annidata ha tre alternative: si incrementa di 1 il valore di una delle tre variabili, num\_pos, num\_neg e num\_zeri, a seconda se x è maggiore, minore o uguale a zero. I riquadri mostrano la struttura logica dell'istruzione; la seconda istruzione if (quella interna) viene eseguita se è falsa la condizione della prima. In pratica, si verifica la prima condizione ( $x > 0$ ); se è vera, num\_pos si incrementa di 1 e si salta il resto dell'istruzione if, altrimenti si verifica la seconda condizione ( $x < 0$ ); se è vera, si incrementa di 1 num\_neg, altrimenti si incrementa num\_zeri. È importante capire che la seconda condizione si verifica *soltanto se la prima condizione è falsa*.

Gli if si possono ovviamente annidare anche nel ramo positivo.

#### Esempio 4.8

```
• if (x > 0)
    if (y > 0) z = sqrt(x) + sqrt(y);
• if (x > 0)
    if (y > 0) z = sqrt(x) + sqrt(y);
    else cerr << "*** Impossibile calcolare z" << endl;
```

#### Esempio 4.9

*Scrivere un programma per giocare a indovinare un numero nascosto dando suggerimenti per tentativi successivi.*

```
#define NUMERO_NASCOSTO 15
int main()
{
    int num = 15;
    cout << "Indovini che numero ho pensato: ";
    cin >> num;
    if (num > NUMERO_NASCOSTO)
    {
        cout << num << " è troppo grande" << endl;
        cout << "provi con un numero più piccolo" << endl;
    }
    else if (num < NUMERO_NASCOSTO)
    {
        cout << num << " è troppo piccolo" << endl;
        cout << "provi con un numero più grande" << endl;
    }
    else
    {
        cout << "Bravo! Come ha fatto ad indovinare?" << endl;
    }
}
```

#### Esecuzione 1

```
Indovini che numero ho pensato: 23
23 è troppo grande
provi con un numero più piccolo
```

### Esecuzione 2

Indovini che numero ho pensato: **11**  
**11** è troppo piccolo  
 provi con un numero più grande

### Esecuzione 3

Indovini che numero ho pensato: **15**  
 Bravo! Come ha fatto ad indovinare?

Un'istruzione if annidata può rendersi anche come sequenza di istruzioni if. Per esempio, l'istruzione if dell'esempio precedente si può riscrivere come la seguente sequenza di istruzioni if.

```
if (x > 0) num_pos++;
if (x < 0) num_neg++;
if (x == 0) num_zeri++;
```

Le due soluzioni sono logicamente equivalenti ma questa è meno efficiente perché compie sempre tre valutazioni booleane, mentre l'if annidato dell'esempio precedente ne compie una sola se ( $x > 0$ ) e solo due se ( $x < 0$ ).

È bene indentare le istruzioni annidate in maniera da rendere il codice ben leggibile. Per esempio, questo codice è corretto ma incomprensibile:

```
if (a > 0) if (b > 0) ++a; else if (c > 0)
if (a < 5) ++b; else if (b < 5) ++c; else --a;
else if (c < 5) --b; else --c; else a = 0;
```

meglio sarebbe scriverlo così:

```
if (a > 0) // forma più leggibile
    if (b > 0) ++a;
else
    if (c > 0)
        if (a < 5) ++b;
        else
            if (b < 5) ++c;
            else --a;
    else
        if (c < 5) --b;
        else --c;
else
    a = 0;
```

o, meglio, così:

```
if (a > 0) // forma ancora più leggibile
    if (b > 0) ++a;
else if (c > 0)
    if (a < 5) ++b;
```

```

else if (b < 5) ++c;
else --a;
else if (c < 5) --b;
else --c;
else a = 0;

```

### Esempio 4.10

*Scrivere un programma per calcolare il maggiore di tre numeri interi.*

```

int main()
{
    int a, b, c, maggiore;
    cout << "Introduca tre numeri interi:";
    cin >> a >> b >> c;
    if (a > b)
        if (a > c) maggiore = a;
        else maggiore = c;
    else
        if (b > c) maggiore = b;
        else maggiore = c;
    cout << "Il maggiore è " << maggiore << endl;
}

```

### Esecuzione

```

Introduca tre numeri interi: 77 54 85
Il maggiore è 85

```

Se  $a > b$  e  $a > c$  allora  $\text{maggior} = a$ .

L'istruzione `else maggiore=c` si riferisce all'`if` a lei più vicino, cioè `if (a>c)`. Quindi se  $a > b$  e  $a <= c$  allora  $\text{maggior} = c$ .

L'istruzione `else` successiva si riferisce invece al primo `if (a>b)`. Se  $a <= b$  e  $b > c$  allora  $\text{maggior} = b$ .

La successiva `else maggiore=c` si riferisce all'`if` a lei più vicino, cioè `if (b>c)`. Quindi se  $a <= b$  e  $b <= c$  allora  $\text{maggior} = c$ .

## 4.5 Istruzione switch

Dall'esercizio e dagli esempi precedenti risulta chiaro come l'uso degli `if` annidati possa essere fonte di confusione. L'istruzione `switch` è stata introdotta proprio per selezionare una tra molteplici alternative. La condizione da valutare non sarà più quindi booleana ma potrà essere un'espressione semplice denominata *espressione di controllo* o *selettore*. Il valore di quest'espressione deve essere un tipo *ordinale* (per esempio, `int`, `char`, `bool` ma non `float`, `double` o `string`). L'istruzione ha la seguente sintassi:

```

switch (selettore)
{
    case etichetta1 : istruzione1; [break];
    case etichetta2 : istruzione2; [break];
    :
    case etichettan : istruzionin; [break];
    [default: istruzionid];
}

```

in cui la notazione [] indica opzionalità; per esempio, [break;] significa che l'inserimento del break è opzionale.

Ogni *etichetta<sub>n</sub>* è un valore costante diverso dalle altre. Se il valore del selettore è uguale a una delle etichette case allora si eseguirà la corrispondente sequenza di istruzioni e si continuerà a eseguire le successive sequenze di istruzioni fino a incontrare la prima istruzione break o fino alla fine dello switch se non s'incontrano istruzioni break.

Il tipo di ogni etichetta deve essere lo stesso del *selettore*.

Le espressioni sono permesse come etichette, ma solo se ogni operando dell'espressione è una costante.

Se il valore del selettore non è uguale a un'etichetta case, non si eseguirà alcuna istruzione, a meno che sia presente l'ultima etichetta opzionale default. Benché l'etichetta default sia opzionale, se ne consiglia l'uso, a meno che non si sia assolutamente sicuri che tutti i valori che il *selettore* potrà assumere siano contemplati nelle etichette case.

#### Esempio 4.11

```

char giocata_totocalcio;
cin >> giocata_totocalcio;
switch (giocata_totocalcio)
{
    case '1':
        cout << "vittoria in casa" << endl; break;
    case '2':
        cout << "vittoria fuori casa" << endl; break;
    case 'x':
        cout << "pareggio" << endl; break;
    default:
        cout << "non è una giocata corretta" << endl;
}

```

#### Esempio 4.12

```
unsigned opzione;
```

```
....
```

```

switch (opzione)
{
    case 0:
    case 1:
    case 2: cout << "minore di 3"; break;
    case 3: cout << "uguale a 3"; break;
    default: cout << "maggiore di 3";
}

```

#### Esempio 4.13

*Scrivere un programma per passare dalla valutazione del sistema didattico anglosassone a quello nominale italiano.*

```

int main()
{
    char voto;
    cout << "Introdurre un voto nel sistema anglosassone (A-H): ";
    cin >> voto;
    switch (voto)
    {
        case 'A': cout << "Ottimo\n"; break;
        case 'B': cout << "Distinto\n"; break;
        case 'C': cout << "Buono\n"; break;
        case 'D': cout << "Sufficiente\n"; break;
        case 'F': cout << "Insufficiente\n"; break;
        default: cout << "Non è un voto";
    }
    return 0;
}

```

L'istruzione switch valuta voto. Se voto è uguale al valore di un'etichetta, allora si trasferisce il flusso di controllo alle istruzioni associate a quell'etichetta. Se nessuna etichetta coincide con il valore di voto, si eseguono le istruzioni associate all'etichetta default. L'istruzione break fa sì che il flusso del programma esca dallo switch. Se una sequenza di istruzioni non è chiusa dal break, si eseguono anche le istruzioni delle etichette successive fino al successivo break (o fino alla fine se non si incontrano break).

#### Esecuzione di prova 1

Introdurre un voto nel sistema anglosassone (A-H): A  
Ottimo

#### Esecuzione di prova 2

Introdurre un voto nel sistema anglosassone (A-H): H  
Non è un voto

È possibile associare la stessa sequenza di istruzioni a più etichette case. Per esempio, si può scrivere:

```

switch(c) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        num_cifre++; break;
    case ' ': case '\t': case '\n':
        num_spazi++; break;
    default: num_altri++;
}

```

Lo switch è particolarmente adatto a implementare "menu", cioè a stampare in output liste di alternative da presentare all'utente del programma.

#### 4.6 if else e operatore condizionale ( ? : )

Le istruzioni di selezione if else possono essere sostituite da opportuni usi dell'operatore condizionale.

##### Esempio 4.14

```
a == b ? funzione1() : funzione2();

equivale all'istruzione:

if (a == b) funzione1();
else funzione2();
```

##### Esempio 4.15

Nel seguente pezzo di codice si utilizza in maniera molto efficiente l'operatore condizionale per assegnare alla variabile minore il più piccolo di due valori letti in input.

```
int input1, input2, minore;
cin >> input1 >> input2;
minore = (input1 <= input2) ? input1 : input2
```

##### Esempio 4.16

*Scrivere un programma per mostrare che si può individuare l'ordinamento fra due numeri letti in input sia con l'istruzione if else sia con l'operatore condizionale.*

```
int main()
{
    float n1, n2;
    cout << "Per favore, introduca due numeri, positivi o negativi: ";
    cin >> n1 >> n2;
    cout << "con if else" << endl;
    if (n1 > n2) cout << n1 << " > " << n2 << endl;
    else cout << n2 << " > " << n1 << endl;
    cout << "con operatore condizionale" << endl;
    n1 > n2 ? cout << n1 << " > " << n2 : cout << n2 << " < " << n1;
}
```

## Esecuzione

Per favore, introduca due numeri, positivi o negativi: -2.7 +3.14

con if else

3.14 > -2.7

con operatore condizionale

3.14 < -2.7

## 4.7 Frequenti errori di programmazione

1. Uno degli errori più comuni dell'if consiste nell'utilizzare l'operatore di assegnamento (`=`) invece dell'operatore di uguaglianza (`==`) nella condizione.
2. Come già detto, in un'istruzione `if` annidata, ogni clausola `else` corrisponde alla `if` precedente più vicina, ma un'indentazione approssimativa può confondere le idee; per esempio, a quale `if` è associata l'`else` nel seguente esempio? e qual'è o quali sono le istruzioni dell'`else`?

```
if (a > 0)
if (10 > 0)
c = a + b;
else
c = a + abs(b);
d = a * b * c;
```

Il sistema migliore per evitare errori è quello di fare buon uso dell'indentazione, per far capire al primo sguardo a quale `if` corrisponde la clausola `else`.

```
if (a > 0)
    if (b > 0)
        c = a + b;
    else
        c = a + abs(b);
d = a * b * c;
```

3. Poiché i numeri reali sono approssimati, non si dovrebbe mai metterli a confronto con l'operatore di uguaglianza `==`. Per esempio, se anche se le espressioni reali seguenti sono equivalenti:

`a * (1 / a)` equivale a 1.0

l'espressione

`a * (1 / a) == 1.0`

potrebbe restituire valore false. Per questo stesso motivo il selettore di un'istruzione `switch` non può essere di tipo reale.

4. Quando in un'istruzione `switch` o in un blocco di istruzioni manca una delle parentesi graffe appare un messaggio di errore del tipo:

Errore ...: Compound statement missing ] in function

Se non si scrive il codice con buon stile d'indentazione può essere difficile localizzare la parentesi mancante.

## 4.8 Istruzione while

Un **ciclo** è la ripetizione controllata di una sequenza di istruzioni. La sequenza si dice **corpo** del ciclo e ogni ripetizione si chiama **iterazione** del ciclo. L'iterazione è ovviamente controllata da una condizione, cioè un'espressione booleana che prima o poi dovrà diventare falsa se si vuole evitare di avviare una ripetizione all'infinito. Il corpo può essere un'istruzione singola o un'istruzione composta (una sequenza di istruzioni racchiuse fra parentesi graffe).

L'istruzione ciclica `while` ha la *condizione* posta davanti al corpo del ciclo; questo significa che si valuta *prima* la condizione e poi si esegue eventualmente il corpo del ciclo. Se la condizione è falsa già in partenza il corpo del ciclo non verrà eseguito neanche una volta (esso sarà quindi eseguito *zero o più volte*). L'esecuzione del corpo si ripete *fin tanto che* la condizione del ciclo rimane vera e termina quando essa si riscontra falsa. La Figura 5.1 rappresenta il flow chart del ciclo `while`.

<code>while</code>	<i>parola riservata C++</i>
<code>condizione_ciclo</code>	<i>espressione logica o booleana</i>
<code>istruzione</code>	<i>istruzione semplice o composta</i>

Il *comportamento* o *funzionamento* di un'istruzione (ciclo) `while` è la seguente.

1. Si valuta la *condizione\_ciclo*.
2. Se la *condizione\_ciclo* è vera:
  - a) si esegue l'istruzione specificata, denominata **corpo** del ciclo;
  - b) il controllo torna al passo 1.
3. In caso contrario il controllo passa all'istruzione successiva al `while`.

Il **corpo** del ciclo si ripete **finché** l'espressione logica (condizione del ciclo) è vera. Quando l'espressione logica risulta falsa si termina e si esce dal ciclo.

### Esempio 4.17

Il seguente frammento di codice conta da 1 a 10

```
int x = 1;
while (x <= 10)
    cout << "x: " << ++x;
```

### Esempio 4.18

Il seguente frammento di codice visualizza n asterischi

```
contatore = 0;
while (contatore < n)
{
    cout << " * ";
    contatore++; //incrementa il contatore
}
```

La variabile **di controllo del ciclo** deve essere:

1. *inizializzata*;
2. *verificata*;
3. *aggiornata* dal corpo del ciclo o da eventi esterni (altrimenti il ciclo diventa **infinito**, cosa normalmente non desiderata dal programmatore).

Un ciclo infinito *indesiderato* si produce quando la condizione del ciclo, pur contenendo almeno una variabile, non viene mai falsificata in alcuna iterazione. Per esempio:

```
contatore = 1;
while (contatore < 100)
{
    cout << contatore << endl;
    contatore--; // decrementa il contatore invece che incrementarlo
}
```

#### Esempio 4.19

```
int main()
{
    int contatore = 0; // inizializza la variabile di controllo
    while(contatore < 5) // condizione di controllo
    {
        contatore++; // corpo del ciclo
        cout << "contatore: " << contatore << endl;
    }
    cout << "Terminato. Contatore = " << contatore << endl;
    return 0;
}
```

#### Esecuzione

```
contatore: 1
contatore: 2
contatore: 3
contatore: 4
contatore: 5
Terminato. Contatore = 5
```

#### Esempio 4.20

*Scrivere un programma per calcolare le calorie assunte durante il giorno.*

```
int main()
{
    int num_pasti, contatore, calorie_per_alimento, calorie_totali;
    cout << "Quanti pasti ha fatto oggi? ";
    cin >> num_pasti;
```

```

calorie_totali = 0;
contatore = 1;
cout << "Introdurre il numero di calorie di ognuno dei "
<< num_pasti << " pasti:\n";
while (contatore++ <= num_pasti)
{
    cin >> calorie_per_alimento;
    calorie_totali = calorie_totali + calorie_per_alimento;
}
cout << "Le calorie totali assunte oggi sono = "
<< calorie_totali << endl;
return 0;
}

```

### Esecuzione

```

Quanti pasti ha fatto oggi? 4
Introdurre il numero di calorie di ognuno dei 4 pasti:
100 254 198 32
Le calorie totali assunte oggi sono = 584

```

Un errore tipico nel while è l'omissione delle parentesi graffe per inglobare la sequenza di istruzioni del suo corpo in un'unica istruzione composta. Il codice seguente:

```

contatore = 1;
while (contatore < 25)
    cout << contatore << endl;
    contatore++;

```

visualizzerà infinite volte il valore 1 perché non verrà mai aggiornata la variabile di controllo contatore, quindi la condizione contatore < 25 non verrà mai falsificata. L'istruzione di output è cioè l'unica istruzione di questo (evidentemente errato) ciclo while.

Come già detto, se si utilizza una condizione di controllo falsa già in partenza, il ciclo non viene mai avviato. Questo a volte può essere voluto, perché la condizione può essere falsa in partenza solo in particolari casi. Se però la condizione è sempre e sicuramente falsa in partenza allora il ciclo non ha alcun senso e certamente siamo di fronte a un errore di progetto dell'algoritmo. Per esempio:

```

contatore = 10
while (contatore > 100)
{
    ...
}

```

Qui alla variabile di controllo contatore viene assegnato un valore (10) immediatamente prima del ciclo, ma questo valore falsifica la condizione (contatore > 100), quindi il corpo del while non sarà mai eseguito. Siamo sicuramente di fronte a un errore.

### 4.3.1 Cicli controllati "da sentinella" o "da contatore"

Nell'esempio precedente il ciclo è controllato da una variabile *contatore* che viene incrementata a ogni iterazione e la condizione verifica che il suo valore non superi un certo limite. Questo significa che sappiamo in partenza quante volte dovrà essere ripetuto il ciclo, ma non sempre siamo così fortunati, perché il numero delle iterazioni da effettuare potrebbe non essere noto in partenza. In questi casi per terminare il ciclo si usa la tecnica della *sentinella*. Si definisce un particolare valore del dato da elaborare come *sentinella*, cioè come quello destinato a indicare la terminazione del ciclo, e si imposta la condizione di controllo in maniera da verificare che il dato da elaborare nella ripetizione corrente abbia valore diverso dal valore sentinella. Per esempio, se dovessimo sommare un numero ignoto di numeri positivi letti da tastiera, potremmo pensare di utilizzare un numero negativo come sentinella per indicare la chiusura del ciclo di lettura:

```
int numero, somma=0;
const int sentinella = -1;
cout << "Per favore, immetta il primo numero: ";
cin >> numero;
while (numero != sentinella)
{
    somma += numero;
    cout << "Per favore, immetta il numero successivo: ";
    cin >> numero;
}
cout << "La somma dei numeri letti è " << somma << endl;
```

questo programma avrebbe un'esecuzione come questa:

```
Per favore, immetta il primo numero: 23
Per favore, immetta il numero successivo: 34
Per favore, immetta il numero successivo: 45
Per favore, immetta il numero successivo: 67
Per favore, immetta il numero successivo: -1
La somma dei numeri letti è 169
```

Per controllare l'esecuzione di un ciclo si possono utilizzare anche variabili tipo *bool* come *indicatori di stato* o *flag* (bandierina). Il valore del *flag* si inizializza (normalmente a *false*) prima del ciclo e si modifica nel ciclo quando accade un certo specifico evento. Il ciclo si ripeterà fino a quando non accadrà quell'evento che cambierà il valore del flag.

#### Esempio 4.21

*Scrivere un programma per leggere caratteri alfabetici introdotti da tastiera fino a quando viene battuto un carattere cifra.*

Si può utilizzare un flag *letta\_cifra* per indicare che il carattere battuto da tastiera è una cifra e inizializzarne il valore a *false*. Il corpo del ciclo deve

contenere un'istruzione che ne cambi il valore a true quando viene introdotto un carattere cifra (eventualmente come primo carattere). Poiché il ciclo va avanti fin tanto che la condizione di controllo ha valore true essa dovrà essere scritta come `!letta_cifra` che rimane true fintanto che `letta_cifra` è false. Il ciclo while sarà:

```
char car;
bool letta_cifra = false;
while (!letta_cifra)
{
    cout << "Immetta un carattere:";
    cin >> car;
    letta_cifra = ((‘0’<= car) && (car <= ‘9’));
    ...
}
```

Il ciclo funziona così:

1. s'inizializza `letta_cifra` (in questo caso a false);
2. la condizione del ciclo `!letta_cifra` è true, quindi si entra nel ciclo;
3. si legge un dato dalla tastiera e lo si mette nella variabile `car`;
4. il flag `letta_cifra` diventa true se il codice ASCII del carattere letto è compreso fra quello del carattere '0' e quello del carattere '9', altrimenti rimane a true;
5. il ciclo termina solo quando `letta_cifra` diventa true.

#### **Metodo per uscire da un ciclo**

1. Settare la flag sentinella a «falso» (o a «vero») prima di entrare nel ciclo `while`.
2. Fin tanto che la la sentinella rimane a «falso» (o a «vero»):
  - 2.1. eseguire le istruzioni del corpo del ciclo;
  - 2.2. se si produce la condizione di uscita, allora modificare il valore della flag, al fine di falsificare la condizione di controllo e così terminare il ciclo.

#### **4.3.2 Cicli infiniti e istruzione break**

L'istruzione `break` non termina solo l'istruzione `switch` ma anche il ciclo `while`. Essa può essere utilizzata per uscire da un ciclo infinito (la cui condizione è cioè sempre true) quando accade un certo evento. Per esempio:

```
int main()
{
    int contatore = 0;
    while (true);
    {
        contatore++;
        if (contatore > 10) break;
    }
}
```

```

    cout << "Contatore: " << contatore << "\n";
    return 0;
}

```

al termine il programma scrive a schermo:

Contatore: 11

Il ciclo while ha una condizione che non può essere falsificata, ne risulterebbe un ciclo infinito! Esso però incrementa la variabile contatore e poi verifica se è maggiore di 10. Se è così, l'istruzione break termina il ciclo while.

#### Esempio 4.22

*Scrivere un programma per calcolare la media aritmetica di un predefinito NUMERO di numeri.*

```

#define NUMERO 6
int main()
{
    int Contatore = 0;
    float Numero, Media, Somma = 0;
    cout << "Per favore, immetta " << NUMERO << " numeri\n";
    while (Contatore < NUMERO)
    {
        cin >> Numero;
        Somma += Numero; // incrementa la somma parziale
        ++Contatore; // incrementare il contatore
    }
    Media = Somma / Contatore;
    cout << "Media: " << Media << endl;
    return 0;
}

```

#### Esecuzione

```

Per favore, immetta 6 numeri
3.14 5.23 4.32 6.28 5.15 1.1
Media: 4.2033

```

### 4.9 Istruzione for

Ogni ciclo while:

```

inizializzazione;
while (condizione)
{
    corpo del ciclo;
    passo;
}

```

può venire convenientemente reso da una specifica istruzione ciclica denominata **for**

```
for (inizializzazione; condizione; passo)
    corpo del ciclo
```

L'uso del **for** è particolarmente comodo nei cicli controllati dal contatore. Per esempio, utilizzando il **for**, il programma dell'esercizio precedente diventa:

```
#define NUMERO 6
int main()
{
    float Numero, Media, Somma = 0;
    cout << "Per favore, immetta " << NUMERO << " numeri\n";
    for (int Contatore = 0; Contatore < NUMERO; Contatore++)
    {
        cin >> Numero;
        Somma += Numero; // incrementa la somma parziale
    }
    Media = Somma / NUMERO;
    cout << "Media: " << Media << endl;
    return 0;
}
```

L'esempio mostra come la variabile di controllo possa essere definita dentro il **for** stesso al momento della sua inizializzazione; questo ci porta a enunciare un principio di buona programmazione: è sempre bene definire le variabili il più localmente possibile; se una variabile è usata solo da una funzione non ha senso definire quella variabile globale, così se una variabile serve solo come contatore di un **for** dentro una funzione, non ha senso definire quella variabile come locale alla funzione (tutt'altro come variabile globale).

Il passo del ciclo può essere un incremento della variabile di controllo (*for ascendente*) o un decremento (*for descendente*).

Esempio di **for ascendente**:

```
for (int n = 1; n <= 10; n++) cout << n << '\t' << n*n << endl;
```

La variabile di controllo è **n** (definita dentro il **for**), il suo valore iniziale è 1, il valore limite è 10 e il passo è l'incremento **n++**. Ciò significa che il ciclo viene eseguito per ogni valore di **n** da 1 a 10. Nella prima iterazione **n** prenderà il valore 1; nella seconda il valore 2 e così via fino al valore 10. Sul canale di output sarà visualizzato:

```
1   1
2   4
3   9
4  16
5  25
```

```

6   36
7   49
8   64
9   81
10  100

```

Esempio di formato discendente:

```
for (int n = 10; n > 5; n--) cout << n << '\t' << n*n << endl;
```

L'uscita di questo ciclo è:

```

10  100
9   81
8   64
7   49
6   36

```

perché il valore iniziale della variabile di controllo è 10, la condizione di controllo è  $n > 5$  e il passo è l'operatore di decremento  $n--$ .

L'incremento/decremento della variabile di controllo può essere di qualsiasi valore e non necessariamente 1, cioè, 5, 10, 20, -4.... Per esempio, il ciclo

```
for (int n = 0; n <= 100; n += 20) cout << n << '\t' << n*n << endl;
```

utilizza come passo l'incremento:

```
n += 20
```

e in output verrà stampato:

```

0   0
20  400
40  1600
60  3600
80  6400
100 10000

```

#### Esempio 4.23

```

// esempio 1
for (int i=9; i>=0; i-=3) cout << (i*i) << endl;

// esempio 2
for (int i=0,j=MAX; i<j; i++,j--) cout << (i+2*j) << endl;

```

Il primo esempio mostra un ciclo discendente che inizializza la variabile di controllo a 9. Il ciclo va avanti fintanto che  $i$  si mantiene positivo; poiché il passo è un decremento di 3, il ciclo viene eseguito quattro volte con  $i$  che prende i valori 9, 6, 3 e 0.

Il secondo esempio dichiara due variabili di controllo,  $i$  e  $j$ , inizializzandole una a 0 e l'altra a MAX.  $i$  viene incrementata di uno, mentre  $j$  viene decrementata di uno. Il ciclo andrà avanti fintanto che  $i$  rimane minore di  $j$ .

**Esempio 4.24**

*Scrivere un programma per sommare i primi dieci numeri naturali.*

```
int main()
{
    unsigned somma = 0;
    for (unsigned n = 1; n <= 10; n++) somma = somma + n;
    cout << "La somma dei numeri da 1 a 10 è " << somma << endl;
    return 0;
}
```

L'uscita del programma è:

La somma dei numeri da 1 a 10 è 55

Ovviamente le variabili hanno visibilità solo nel blocco in cui sono definite, quindi quando un ciclo for dichiara una variabile di controllo, essa rimane accessibile solo dentro il for. Per esempio, nel rendere l'Esercizio 4.12 con il for invece che con il while, non abbiamo potuto mantenere il parallelo perfetto:

```
for (int Contatore = 0; Contatore < NUMERO; Contatore++)
{
    cin >> Numero;
    Somma += Numero; // incrementa la somma parziale
}
Media = Somma / Contatore;
```

perché, pur essendo l'algoritmo concettualmente corretto, il compilatore dà errore poiché la variabile Contatore è definita dentro al for e quindi non è visibile al suo esterno.

Le variabili di controllo possono essere di tipo float o double e si possono incrementare/decrementare di quantità non intere.

```
for (float lung = 0.75; lung <= 5; lung += 0.05)
    cout << "lunghezza è ora uguale a " << lung << endl;
```

In realtà il passo può essere una qualunque istruzione (non necessariamente un incremento/decremento). Per esempio, supponendo inclusa la libreria <cmath>:

```
for (float x = pow(5, 3.0); x > 2.0; x = sqrt(x))
    cout << "x è ora uguale a " << x << endl;
```

Naturalmente, se la variabile di controllo non è di tipo int, si avranno meno garanzie di precisione.

#### 4.10 Precauzioni nell'uso del for

Come per il ciclo while, anche per il for si deve porre attenzione a che l'inizializzazione e il passo siano tali per cui la condizione del ciclo venga prima

o poi falsificata. In particolare sarebbe da evitare che *il corpo del ciclo modifichi variabili utilizzate come termini di ciclo nella condizione di controllo*, perché questo potrebbe portare a strane interferenze fra il passo e il corpo del ciclo con rischi di incappare in cicli infiniti. Per esempio, l'esecuzione di

```
int limite = 1;
for (int i = 0; i <= limite; i++)
{
    cout << i << endl;
    limite++;
}
```

produce una sequenza teoricamente infinita di interi dato che, a ogni iterazione, l'istruzione del corpo `limite++` incrementa `limite` di 1, prima che il passo `i++` incrementi `i` (questo `for` in realtà terminerà quando `i` assumerà il valore `MAXINT` e `limite` il valore `MAXINT+1` cioè `MININT`, ma sarebbe questo comunque un risultato probabilmente non voluto).

Nel caso del `for` sarebbe anche da evitare che *il corpo del ciclo modifichi le variabili di controllo utilizzate nel passo*; per esempio, il ciclo:

```
int limite = 1;
for (int i = 0; i <= limite; i++)
{
    cout << i << endl;
    i--;
}
```

produrrà infiniti zeri poiché l'istruzione `i--` del corpo del ciclo decrementa `i` di 1 prima che il passo `i++` la incrementi. Come risultato `i` è sempre 0 a ogni iterazione del `for`.

Come nel `while` l'istruzione `break` può essere utilizzata per uscire da un `for` infinito, cioè senza inizializzazione, né condizione né passo, ovvero:



**ATTENZIONE!** Se si esegue un `for` senza inizializzazione, né condizione né passo, l'istruzione si esegue all'infinito, a meno che non si utilizzi un'istruzione `break` o `return` (normalmente dentro un `if`).

per esempio:

```
int num;
for (;;)
{
    ...
    cout << "Introduca un numero";
    cin >> num;
    if (num == -999) break;
    ...
}
```

Attenzione a non collocare un punto e virgola dopo le parentesi tonde del ciclo `for`, perché questo significherebbe che esso non ha il corpo e quindi non farebbe alcunché. Per esempio, il ciclo:

```
for (int i = 1; i<= 10; i++);
    cout << "Stampa qualcosa" << endl;
```

non stampa proprio nulla e il compilatore non produce alcun messaggio di errore. La frase `Stampa qualcosa` si stampa soltanto una volta (e non dieci), dopo le dieci iterazioni vuote del ciclo `for`.

#### 4.11 Istruzione `do while`

Se si ha la sicurezza che il corpo del ciclo deve essere eseguito almeno una volta può essere conveniente utilizzare l'istruzione `do while`.

<code>do</code>	parola riservata C++
<code>istruzione</code>	istruzione semplice o composta
<code>while</code>	condizione espressione logica o booleana

Il ciclo `do while` comincia eseguendo `istruzione`, e solo dopo averla eseguita valuta la `condizione`. Se questa è vera, allora `istruzione` viene ripetuta e così via, fino a quando `condizione` diventa falsa. Quest'istruzione è utile per proporre menù di istruzioni all'utente; di seguito ne viene presentato un esempio.

#### Esempio 4.25

*Scrivere un programma che proponga continuamente qualcosa all'utente.*

```
int main()
{
    char opzione;
    do
    {
        cout << "Salve!" << endl;
        cout << "Vuole un altro saluto?\n";
        << "Prema s per si e n per no,\n"
        << "e poi prema invio: ";
        cin >> opzione;
    } while (opzione == 's'|| opzione == 'S');
    cout << "Addio!\n";
    return 0;
}
```

#### Esecuzione di prova

```
Salve!
Vuole un altro saluto?
Prema s per si e n per no
e poi prema invio: S
```

```

Salve!
Vuole un altro saluto?
Prema s per si e n per no
e poi prema invio: N
Addio!

```

#### 4.12 Confronto fra while, for e do while

La Tabella 4.1 spiega quando usare ognuno dei tre cicli. In C++, il ciclo for è il più usato. È relativamente facile riscrivere un ciclo do while come while, inserendo un'assegnazione iniziale della variabile condizionale. Tuttavia, non tutti i cicli while si possono esprimere come do while, dato che il corpo di quest'ultimo si esegue sempre almeno una volta, mentre quello del while può non essere eseguito. Il while viene quindi generalmente preferito, a meno che non sia chiaro che si deve eseguire almeno un'iterazione.

Proponiamo un confronto sinottico schematico delle tre istruzioni:

```

contatore = valore_iniziale;
while (contatore < valore_fermata)
{
    // corpo del ciclo
    contatore++;
}

for (contatore=valore_iniziale; contatore<valore_fermata; contatore++)
{
    // corpo del ciclo
}

contatore = valore_iniziale;
if (valore_iniziale < valore_fermata)
do
{
    // corpo del ciclo
    contatore++;
} while (contatore < valore_fermata);

```

**Tabella 4.1** Formatì dei cicli

while	È particolarmente usato quando l'iterazione è controllata da sentinella. Poiché il corpo del ciclo può non essere eseguito, si utilizza quando si vuole saltare all'istruzione successiva se la condizione è falsa in partenza.
for	È particolarmente usato quando l'iterazione è controllata da contatore. La condizione di controllo precede l'esecuzione del corpo del ciclo.
do while	È usato quando si deve eseguire almeno un'iterazione. La condizione di controllo segue l'esecuzione del corpo del ciclo.

### 4.13 Progetto di un'istruzione ciclica

Il progetto di un'istruzione ciclica richiede quindi la definizione di tre parti:

1. il corpo del ciclo;
2. le istruzioni di inizializzazione;
3. le condizioni per la terminazione del ciclo.

I cicli vengono spesso impiegati per calcolare sommatorie e produttorie di serie di numeri. Se si conosce la lunghezza della serie, la sommatoria può essere facilmente calcolata con un ciclo a contatore inizializzando la variabile sommatoria (esterna al ciclo per poter essere poi utilizzata al di fuori di esso) a 0. Per esempio, con un ciclo for:

```
int sommatoria = 0;
for (int contatore=1; contatore<=lunghezza_della_serie; contatore++)
{
    ... produrre il dato successivo
    sommatoria += dato successivo;
}
```

Se invece di una sommatoria si deve calcolare una produttoria bisogna ovviamente inizializzarla a 1 invece che a 0:

```
int produttoria = 1;
for (int contatore=1; contatore<=lunghezza_della_serie; contatore++)
{
    ... produrre il dato successivo
    produttoria *= dato successivo;
}
```

Esistono vari modi per terminare un ciclo di prelievo dati dal canale di input.

#### Terminazione predeterminata dal numero dei dati da elaborare

Se si può determinare in anticipo il numero n di dati da elaborare, magari chiedendolo all'utente, si può utilizzare un ciclo gestito dal contatore.

#### Proporre la terminazione prima di ogni successiva iterazione

Si tratta semplicemente di chiedere all'utente, dopo ogni iterazione del ciclo, se esso deve essere iterato o no. Per esempio:

```
sommatoria = 0;
cout << "Ci sono ancora dati da elaborare?: S\N\n";
char risposta;
cin >> risposta;
while ((risposta == 'S') || (risposta == 's'))
{
    cout << "Introduca un numero: ";
    cin >> numero;
    sommatoria += numero;
```

```

cout << "Ci sono altri numeri?: S\N\n";
cin >> risposta;
}

```

### Ciclo controllato da sentinella

Il modo più pratico ed efficiente per terminare un ciclo di lettura da tastiera è tramite un valore sentinella. Si tratta di un valore diverso da tutti gli altri in lettura che possa servire da indicatore del termine del ciclo. Per esempio, se si deve leggere una lista di numeri positivi, come sentinella si può utilizzare un numero negativo.

```

cout << "Introduca una sequenza di interi positivi" << endl;
      << "Termini la sequenza con un numero negativo" << endl;
sommatoria = 0;
cin >> numero;
while (numero >= 0)
{
    sommatoria += numero;
    cin >> numero;
}
cout << "La somma è: " << sommatoria;

```

### Terminazione per chiusura del flusso di input

Quando si legge da un file, si può terminare il ciclo quando il file è finito. Normalmente il C++ utilizza l'identificatore standard eof per denotare la fine del file. Se il flusso di input proviene dalla tastiera (standard input) come codice di fine flusso si può utilizzare la combinazione CTRL-D. Supponiamo, per esempio, di dover leggere una serie di numeri dallo standard input:

```

sommatoria = 0;
while (cin >> numero) sommatoria += numero;
cout << "La somma è: " << sommatoria;

```

Questo ciclo termina quando viene falsificata la strana condizione (cin>>numero). L'espressione viene valutata, quindi viene eseguita l'operazione di lettura dall'input, e restituisce l'indirizzo dell'oggetto cin, che è un valore diverso da zero, quindi tale valore viene interpretato come true, di conseguenza il ciclo itera in continuazione; smette di iterare quando l'utente preme da tastiera CTRL-D o CTRL-C, a seconda del sistema operativo utilizzato. A quel punto viene chiuso il flusso di input e la valutazione dell'espressione cin>>numero restituisce false.

#### 4.13.1 Istruzioni di salto

A volte è conveniente uscire da un ciclo durante l'iterazione corrente (ovvero non al termine né all'inizio dell'iterazione corrente). Lo si può fare inserendo dentro un if una di queste tre istruzioni di salto, elencate in ordine di "gravidità" crescente:

1. `continue;` // si salta all'iterazione successiva
2. `break;` // si salta all'istruzione successiva
3. `goto etichetta;` // si salta al punto etichettato

Queste istruzioni di salto possono risultare comode ma non sono necessarie, come dimostrato dal teorema di Böhm-Jacopini nel 1966. Il loro problema è che tendono a minare la struttura modulare del programma rendendolo confuso e intrecciato (*spaghetti code*).

#### Esempio 4.26

##### Esempio continue

```
int main()
{
    int anno = 2015;
    while (anno < 2025)
    {
        anno++;
        cout << "\nNel " << anno;
        if (anno == 2020) continue;
        cout << " va tutto bene!" ;
    }
    if (anno == 2025) return 0;
    cout << " scoppia la pandemia e";
    paperino: cout << " mi faccio il vaccino!\n";
    return 0;
}
```

#### Esecuzione

```
Nel 2016 va tutto bene!
Nel 2017 va tutto bene!
Nel 2018 va tutto bene!
Nel 2019 va tutto bene!
Nel 2020
Nel 2021 va tutto bene!
Nel 2022 va tutto bene!
Nel 2023 va tutto bene!
Nel 2024 va tutto bene!
Nel 2025 va tutto bene!
```

#### Esempio 4.27

##### Esempio break

```
int main()
{
```

```

int anno = 2015;
while (anno < 2025)
{
    anno++;
    cout << "\nNel " << anno;
    if (anno == 2020) break;
    cout << " va tutto bene!" ;
}
if (anno == 2025) return 0;
cout << " scoppia la pandemia e";
paperino: cout << " mi faccio il vaccino!\n";
return 0;
}

```

**Esecuzione**

Nel 2016 va tutto bene!  
 Nel 2017 va tutto bene!  
 Nel 2018 va tutto bene!  
 Nel 2019 va tutto bene!  
 Nel 2020 scoppia la pandemia e mi faccio il vaccino!

**Esempio 4.28****Esempio goto**

```

int main()
{
    int anno = 2015;
    while (anno < 2025)
    {
        anno++;
        cout << "\nNel " << anno;
        if (anno == 2020) goto paperino;
        cout << " va tutto bene!" ;
    }
    if (anno == 2025) return 0;
    cout << " scoppia la pandemia e";
    paperino: cout << " mi faccio il vaccino!\n";
    return 0;
}

```

**Esecuzione**

Nel 2016 va tutto bene!  
 Nel 2017 va tutto bene!  
 Nel 2018 va tutto bene!  
 Nel 2019 va tutto bene!  
 Nel 2020 mi faccio il vaccino!

#### 4.14 Cicli annidati

È possibile *annidare* cicli, cioè metterli uno dentro l'altro. A ogni iterazione del ciclo esterno, quello interno viene rieseguito completamente.

##### Esempio 4.29

*Scrivere un programma che visualizzi a schermo un triangolo isoscele fatto di asterischi.*

```
*  
***  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

Risolviamo il problema tramite un ciclo esterno per disegnare le righe (10 nel nostro esempio) e due cicli interni: il primo per disegnare gli spazi (da 9 a 0 nel nostro esempio) e il secondo per disegnare gli asterischi (da 1 a 19 nel nostro esempio); al termine di ogni riga si va a capo.

```
#define RIGHE 10  
int main()  
{  
    cout << endl;  
    // scrive riga; ciclo esterno  
    for (int riga = 1; riga <= RIGHE; riga++)  
    {  
        // scrive spazi; primo ciclo interno alla riga  
        for (int spazio = RIGHE-riga; spazio > 0; spazio--) cout << ' ';  
        // scrive asterischi; secondo ciclo interno alla riga  
        for (int asterisco = 1; asterisco < 2*riga; asterisco++) cout << '*';  
        cout << endl; // termina la riga  
    }  
    return 0;  
}
```

##### Esempio 4.30

*Scrivere un programma che visualizzi a schermo un rettangolo di dimensioni fornite in input, così come anche in input venga letto il carattere con cui scriverlo.*

```
int main()
```

```

{
    int righe, colonne;
    char c;
    cout << "Disegno un rettangolo\nDi quante righe? ";
    cin >> righe;
    cout << "Di quante colonne? ";
    cin >> colonne;
    cout << "Con quale carattere? ";
    cin >> c;
    for (int i = 0; i < righe; i++)
    {
        for (int j = 0; j < colonne; j++) cout << c;
        cout << "\n";
    }
    return 0;
}

```

### Esecuzione di prova

```

Disegno un rettangolo
Di quante righe? 10
Di quante colonne? 20
Con quale carattere? +
+++++
+++++
+++++
+++++
+++++
+++++
+++++
+++++
+++++
+++++
+++++
+++++
+++++
+++++
+++++
+++++

```

Nell'esempio precedente il programmatore ha utilizzato lo stesso identificatore (*i*) per indicare sia la variabile di controllo del ciclo esterno sia la variabile di controllo del ciclo interno! Si tratta di una scelta temeraria che va evitata perché fonte di confusione; la variabile del ciclo interno, avendo lo stesso nome di quella del ciclo esterno, maschera quest'ultima, con la conseguenza che, se fosse necessario nel ciclo interno riferirsi alla variabile del ciclo esterno (ma non è stato questo il caso), ciò non si potrebbe fare. Dando invece un nome diverso alla variabile interna (per esempio, *j*) sarebbe possibile nel ciclo interno usufruire di espressioni che le utilizzino entrambe (per esempio, *i<j*).

**CONCETTO**

In questo capitolo abbiamo affrontato i temi centrali della cosiddetta "programmazione strutturata", cioè le "strutture di controllo" del flusso di programma, che sono l'istruzione *composta* (il già trattato "blocco di istruzioni" che, eseguite in sequenza, equivalgono a una singola istruzione), le istruzioni *condizionali* e quelle *cicliche*.

Le istruzioni condizionali sono l'*if* e lo *switch*. L'*if* seleziona la sequenza di istruzioni da compiere in base al risultato di un test (la sua "condizione"). Esso ha due forme, la semplice e la composta, cioè l'*if* *else*; la prima decide se eseguire o no una sequenza di istruzioni, la seconda quale sequenza eseguire fra due. Quindi per entrambi gli *if* la condizione è di tipo booleano. Lo *switch* invece consente di scegliere fra più sequenze alternative, pertanto la sua condizione sarà la valutazione di un'espressione semplice che potrà restituire più risultati a ciascuno dei quali il programmatore avrà fatto corrispondere l'istruzione da eseguire.

Le istruzioni cicliche si eseguono ripetutamente fintanto che rimane

vera una certa condizione di controllo, tipicamente un *contatore* (per contare le ripetizioni del ciclo) oppure una *sentinella* (quando non si sa in anticipo il numero delle ripetizioni che bisogna fare). Il C/C++ contempla tre istruzioni cicliche:

- Il *while* verifica prima la sua condizione booleana e, se è true, esegue le istruzioni del corpo del ciclo;
- Il *do while* esegue prima il corpo del ciclo e poi verifica la condizione per sapere se ripeterlo un'altra volta;
- Il *for* che è un *while* opportunamente mascherato in maniera da renderlo più comodo da usare nei cicli a contatore.

L'integrità della programmazione strutturata può essere violata in C/C++ per opera delle cosiddette "istruzioni di salto": *break*, *continue* e *goto*. L'istruzione *continue* salta alla fine dell'iterazione corrente, l'istruzione *break* salta alla fine del ciclo corrente, l'istruzione *goto* identificatore\_posizione salta a un punto del programma etichettato da quell'identificatore di posizione.

**ESERCIZI SVOLTI**

- Struttura di controllo
- Istruzione *break*
- Istruzione *composta*
- Istruzione *if*
- Istruzione *switch*
- Tipo di dato *bool*
- Ciclo, Iterazione/ripetizione
- Controllo di cicli
- Ottimizzazione di cicli
- Istruzione *do-while*
- Istruzione *for*
- Istruzione *while*
- Terminazione di un ciclo
- Confronto di *while*, *for* e *do while*

## Esercizi

**Esercizio 1:** Che valore si assegna alla variabile consumo nell'istruzione if seguente se la velocità è 120?

```
if (velocita > 80) consumo = 10.00;
else if (velocita > 100) consumo = 12.00;
else if (velocita > 120) consumo = 15.00;
```

**Esercizio 2:** Spiegare le differenze tra le istruzioni della colonna di sinistra e quelle della colonna di destra. Per ognuna di esse dedurre il valore finale di x se il suo valore iniziale è 0.

```
if (x >= 0) x = x+1;           if (x >= 0) x = x+1;
else if (x >= 1) x = x+2;    if (x >= 1) x = x+2;
```

**Esercizio 3:** Cosa c'è di scorretto nei seguenti pezzi di codice?

- `if (x = 0) cout << x << " = 0\n";`  
else cout << x << " != 0\n";
- `if (x < y < z) cout << x << "<" << y << "<" << z << endl;`
- `if x > 25.0 y = x`  
else y = z;

**Esercizio 4:** Qual è l'errore di questo codice?

```
cout << "Per favore, immetta un numero positivo
n:";
cin >> n;
if (n < 0)
    cout << "Questo numero è negativo. Provvi di
    nuovo.\n";
    cin >> n;
else
    cout << "Grazie! n= " << n << endl;
```

**Esercizio 5:** Scrivere un'istruzione if else che classifichi un intero x in una delle seguenti categorie:

`x < 0 oppure s x <= 100 oppure x > 100`

**Esercizio 6:** Scrivere un programma che determini il maggiore fra tre numeri.

**Esercizio 7:** Che output produrrà il codice seguente, quando si inserisce in un programma completo?

```
int prima_opzione = 1;
switch (prima_opzione + 1)
{
    case 1: cout << "Agnello arrosto\n"; break;
```

```
case 2: cout << "Cotoletta abbacchio\n";
break;
case 3: cout << "Bistecca\n";
case 4: cout << "Dolce di pasticcio\n";
break;
default: cout << "Buon appetito\n";
```

}

**Esercizio 8:** Scrivere un programma che scriva la qualifica corrispondente al voto numerico secondo il seguente criterio:

0 ≤ voto < 5.0	Insufficiente
5 ≤ voto < 6.5	Sufficiente
6.5 ≤ voto < 8.5	Buono
8.5 ≤ voto < 10	Distinto
voto = 10	Ottimo

**Esercizio 9:** Scrivere un programma che determini se il carattere introdotto da tastiera corrisponda a un carattere alfabetico, una cifra, un carattere di interpunkzione, un carattere speciale o un carattere non stampabile.

**Esercizio 10:** Si consideri il seguente codice:

```
for (i = 0; i < n; ++i) —n;
cout << i << endl;
```

- qual è l'uscita se n è 0?
- qual è l'uscita se n è 1?
- qual è l'uscita se n è 3?

**Esercizio 11:** Qual è l'output dei seguenti cicli?

```
int n, m;
for (n = 1; n <= 10; n++)
    for (m = 10; m >= 1; m--)
        cout << n << " volte " << m << " = " << n *
        m << endl;
```

**Esercizio 12:** Scrivere un programma che calcoli e visualizzi in output serie numeriche del tipo:

$$1 + 2 + 3 + \dots + (n-1) + n$$

dove n è un valore letto in input. In particolare lo si applichi alla serie:

$$\pi = 2 \cdot \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdots$$

che restituisce la costante *pigreco* (3,1441592...), e alla serie:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} = \sum_{i=0}^n \frac{x^i}{i!}$$

che restituisce il valore di  $e^x$ .

**Esercizio** Scrivere un ciclo while che visualizzi tutte le potenze di un intero *n*, minori di un valore specificato letto in input.

**Esercizio** Compreso il seguente ciclo while, riscrivere con for e do while.

```
num = 10;
while (num <= 100){
    cout << num << endl;
    num += 10;
}
```

**Esercizio** Scrivere un programma che visualizzi a schermo rombi come questo:

```
*
 *
 * *
 *
 * * *
 *
 * * * *
 *
 * * * *
 *
 * * *
 *
 *
```

di dimensione letta in input.

**Esercizio** Descrivere l'output dei seguenti cicli:

```
a) for (int i = 1; i <= 5; i++){
    cout << i << endl;
    for (int j = i; j >= 1; j--)
        cout << j << endl;
}
b) for (int i = 3; i > 0; i--)
    for (int j = 1; j <= i; j++)
        for (int k = i; k >= j; k--)
            cout << i << j << k << endl;
c) for (int i = 1; i <= 3; i++)
    for (int j = 1; j <= 3; j++)
        for (int k = 1; k <= j; k++)
            cout << i << j << k << endl;
        cout << endl;
```

**Esercizio** Data una serie di numeri interi letti in input da un file, calcolare il fattoriale di ciascuno di loro.

**Esercizio** Scrivere e verificare un programma che risolva equazioni di secondo grado ( $ax^2 + bx + c = 0$ ).

**Esercizio** Scrivere un programma che, letti in input tre interi, indichi se stanno in ordine numerico oppure no.

**Esercizio** Scrivere un programma che, letti due interi e un carattere fra *x*, *-*, *\**, */* e *%* calcoli la corrispondente espressione aritmetica interpretando i due interi come operandi e il carattere come operatore.

**Esercizio** Scrivere un programma che legga interi in input e conti quanti sono positivi, negativi o zero.

**Esercizio** Scrivere un programma che legga interi in input e ne trovi il maggiore e il minore.

**Esercizio** Implementare l'algoritmo di *Euclide* che trova il massimo comune divisore di due numeri interi e positivi.

**Esercizio** Scrivere un programma che calcoli e visualizzi il più grande, il più piccolo e la media di N numeri letti in input.

**Esercizio** Il matematico Leonardo Fibonacci propose il seguente problema. Supponiamo che una coppia di conigli abbia un paio di coniglietti ogni mese e ogni nuova coppia diventi fertile all'età di un mese. Se si dispone di una coppia fertile e nessuno dei conigli muore, quante coppie vi saranno dopo un anno? Migliorare il problema calcolando il numero di mesi necessari per produrre un numero dato di coppie di conigli.

**Esercizio** Calcolare il fattoriale di un numero intero letto in input utilizzando le istruzioni while, do while e for.

**Esercizio** Trovare il numero maggiore di una serie di numeri.

**Esercizio** Scrivere un programma che visualizzi a schermo il seguente output:

```
1
1   2
1   2   3
1   2   3   4
1   2   3
1   2
1
```

 Un numero perfetto è un intero positivo che è uguale alla somma di tutti i suoi divisori (escluso lui stesso). Il primo numero perfetto è 6, dato che i suoi divisori sono 1,

2, 3 che sommano 6. Scrivere un programma che trovi tutti i numeri perfetti minori di un numero letto in input.

# Funzioni

# 5

[Indice](#)

- |            |  |             |                                     |
|------------|--|-------------|-------------------------------------|
| <b>5.1</b> | Concetto di funzione                     | <b>5.8</b>  | Specificatore<br>di accesso auto    |
| <b>5.2</b> | Struttura di una funzione                | <b>5.9</b>  | Funzioni di libreria                |
| <b>5.3</b> | Prototipi delle funzioni                 | <b>5.10</b> | Compilazione modulare               |
| <b>5.4</b> | Passaggio di parametri alla<br>funzione  | <b>5.11</b> | Ricorsione                          |
| <b>5.5</b> | Argomenti di default                     | <b>5.12</b> | Sovraccaricamento delle<br>funzioni |
| <b>5.6</b> | Funzioni in linea (inline)               | <b>5.13</b> | Template di funzioni                |
| <b>5.7</b> | Visibilità e "storage classes" in<br>C++ |             |                                     |

## Introduzione

Le funzioni sono programmi che possono essere mandati in esecuzione da altri programmi. La loro enorme importanza sta nel fatto che esse evitano ripetizioni di codice e facilitano la programmazione rendendo possibile riutilizzare programmi già fatti all'interno di programmi nuovi.

Ogni funzione ha un nome che serve al programma chiamante per mandarla in esecuzione; ogni programma C/C++ è esso stesso una funzione, chiamata `main()`.

Le funzioni definite all'interno di un programma possono essere mandate in esecuzione anche da altri programmi. Se si raggruppano funzioni ben collaudate in librerie tematiche, altri programmi potranno utilizzarle facilmente comprimendo

enormemente i tempi di sviluppo del software e rendendolo più affidabile.

La maggior parte dei programmati non costruiscono librerie, ma le usano tutti. Qualunque compilatore C++ include più di cinquecento funzioni di libreria, che appartengono alla *libreria standard ANSI* (*American National Standards Institute*). Fra le numerose funzioni di librerie, non sarà sempre facile trovare quella che serve, perciò è utile munirsi del manuale delle funzioni di libreria del compilatore. La potenza di un linguaggio sta anche nella sua libreria.

Compilazione separata, sovraccaricamento e ricorsione sono concetti la cui conoscenza è essenziale per progettare un software efficiente, modulare e manutenibile.

### 5.1 Concetto di funzione

Anche il C/C++ consente la *programmazione modulare*: il programma si compone di vari sottoprogrammi specifici denominati *funzioni*. Per esempio, se si sta scrivendo un programma per leggere una lista di caratteri dalla tastiera, metterli in ordine alfabetico e scriverli poi a schermo, si potrebbe far svolgere tutti questi compiti allo stesso unico programma (che sarà esso stesso una funzione, la `main()`).

```
int main()
{
    // Codice per leggere dall'input una sequenza di caratteri
    ...
    // Codice per ordinare i caratteri in senso alfanumerico
    ...
    // Codice per inviare in output la sequenza ordinata
    ...
    return 0
}
```

Tuttavia, un modo migliore per scrivere il programma è quello di definire funzioni indipendenti per ogni sottocompito:

```
int main()
{
    acquisire_caratteri(); // Chiamata a una funzione che legge i
                           // caratteri dall'input
    ordinare_caratteri(); // Chiamata alla funzione che ordina
                           // alfabeticamente i caratteri letti
    visualizzare_caratteri(); // Chiamata alla funzione che
                               // scrive la sequenza a schermo
    return 0; // ritorno al sistema operativo

void acquisire_caratteri()
{
    // Codice C++ per acquisire dall'input una lista di caratteri
}

void ordinare_caratteri()
{
    // Codice C++ per ordinare alfanumericamente i caratteri
}

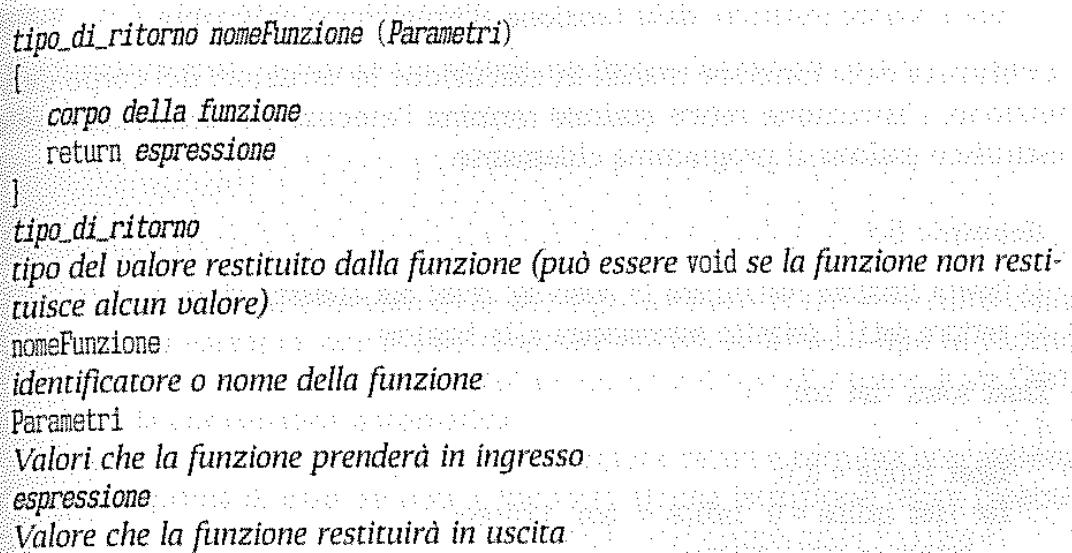
void visualizzare_caratteri()
{
    // Codice C++ per visualizzare a schermo la sequenza ordinata
}
```

 Evitare di scrivere funzioni più lunghe di una schermata.

## 5.2 Struttura di una funzione

In C/C++ le funzioni possono essere mandate in esecuzione in qualunque punto di qualunque funzione (se una funzione chiama sé stessa si parla di "chiamata ricorsiva"), ma sono globali, quindi non possono essere definite dentro funzioni (neanche dentro la `main()`).

La Figura 5.1 mostra la struttura di una funzione in C++.



Gli aspetti più importanti nella progettazione di una funzione sono i seguenti.

- *Il Tipo del risultato.* È il tipo del dato che la funzione restituisce al programma che la manda in esecuzione.
  - *Argomenti formali.* È la lista dei parametri (tipizzati) che la funzione richiede al programma che la chiama; vengono scritti nel formato seguente:
- tipo1 parametro1, tipo2 parametro2, ...*
- *Corpo della funzione.* È il codice del sottoprogramma; si racchiude tra parentesi graffe senza punto e virgola dopo quella di chiusura.

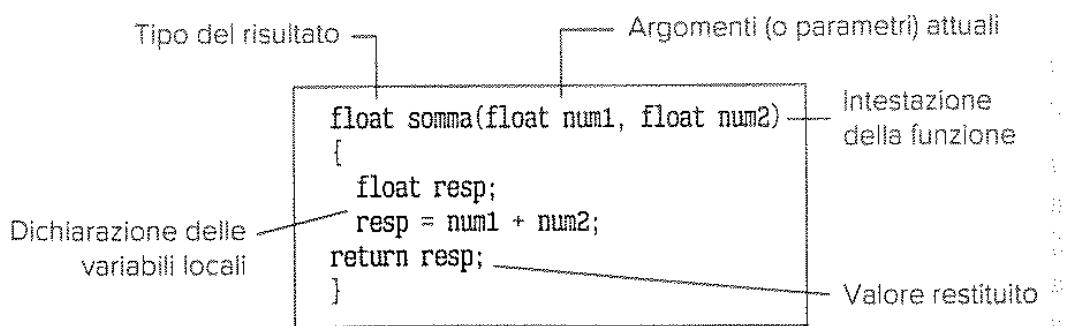


Figura 5.1  
Struttura di una funzione.

- *Passaggio di parametri.* Quando viene mandata in esecuzione una funzione le si passano i suoi argomenti "attuali" e, come vedremo, questo passaggio in C++ avviene "per valore" o "per riferimento".
- *Dichiarazioni locali.* Gli argomenti formali, le costanti e le variabili definite dentro la funzione sono locali a essa, esistono solo mentre la funzione è in esecuzione e non sono accessibili fuori di essa.
- *Valore restituito dalla funzione.* Mediante la parola riservata `return` si può ritornare il valore restituito dalla funzione al programma chiamante.

La chiamata della funzione manda in esecuzione le istruzioni del corpo della funzione. L'istruzione `return qualcosa` termina l'esecuzione della funzione e restituisce `qualcosa` al programma chiamante.

#### Esempio 5.1

```
// Questa funzione restituisce la somma dei primi num elementi di un
// vettore dati[] definito esternamente alla funzione

float somma (int num)
{
    float totale = 0.0;
    for (int indice = 0; indice <= num; indice++)
        totale += dati[indice];
    return totale;
}
```

#### 5.2.1 Nome di una funzione

Un nome di una funzione comincia con una lettera o un underscore (`_`) e può contenere lettere, cifre o underscore. C/C++ è «case sensitive», ovvero lettere maiuscole e minuscole sono caratteri diversi.

```
int max (int x, int e)           // nome della funzione max
double media (double x1, double x2) // nome della funzione media
```

#### 5.2.2 Tipo del dato di ritorno

Il tipo restituito può essere uno dei tipi semplici, come `int`, `char` o `float`, un *puntatore* (lo vedremo più avanti) a qualunque tipo C++, o un tipo `struct`.

```
int max(int x, int y) // ritorna un tipo int
double media(double x1, double x2) // ritorna un tipo double
float funz0() [...] // ritorna un float
char* funz1() [...] // ritorna un puntatore a char
int* funz3() [...] // ritorna un puntatore a int
char* funz4() [...] // ritorna un puntatore a un array char
int funz5() [...] // ritorna un int [è opzionale ma consigliato]
struct InfoPersona CercareRegistro(int num_registro);
```

Molte funzioni non restituiscono risultati e si utilizzano solo come *subroutine* per realizzare compiti concreti. Una funzione che non restituisce un risultato si dice *procedura* e si specifica utilizzando la parola riservata `void` come speciale tipo di dato.

```
void VisualizzareRisultati(float Totale, int num_elementi);
```

### 5.2.3 Risultati di una funzione

Una funzione può restituire un valore mediante l'istruzione `return` la cui sintassi è:

```
return(espressione);
return;
```

`espressione` deve essere ovviamente del tipo definito come restituito dalla funzione; per esempio, non si può restituire un valore `int` se il tipo di ritorno è un puntatore. Tuttavia, se si restituisce un `int` e il tipo di ritorno è un `float`, si realizza la conversione automatica.

Una funzione può avere più di un'istruzione `return` e termina non appena s'esegue la prima di esse. Se non si incontra alcuna istruzione `return` l'esecuzione continua fino alla parentesi graffa finale del corpo della funzione.

Se il tipo di ritorno è `void`, l'istruzione `return` si può omettere:

```
void funz1()
{
    cout << "Questa funzione non prende e non restituisce valori";
}
```

Il valore restituito si racchiude solitamente tra parentesi tonde, ma il loro uso è opzionale. I sistemi operativi accettano un valore restituito dalla funzione principale, cioè dalla `main()`, che se tutto è andato bene dovrebbe essere il numero 0. In C++ tale tipo deve essere `int`.

```
int main()
{
    cout << "Ciao mondo" << endl;
    return 0;
}
```

#### Ricordiamo

Un errore tipico è quello di dimenticare l'istruzione `return` o di situarla dentro una sezione di codice che non verrà eseguita. In questi casi il risultato della funzione è imprevedibile e probabilmente porterà a risultati scorretti. Per esempio, se si mette l'istruzione `return` dentro un'istruzione condizionale come:

```
if (Totale >= 0.0) return Totale;
```

se Totale è minore di zero non si esegue l'istruzione `return` e il risultato restituito dalla funzione è un valore casuale (comunque in questi casi C++ genera di solito un messaggio di warning che aiuta a riconoscere questo errore).

#### 5.2.4 Chiamata a una funzione

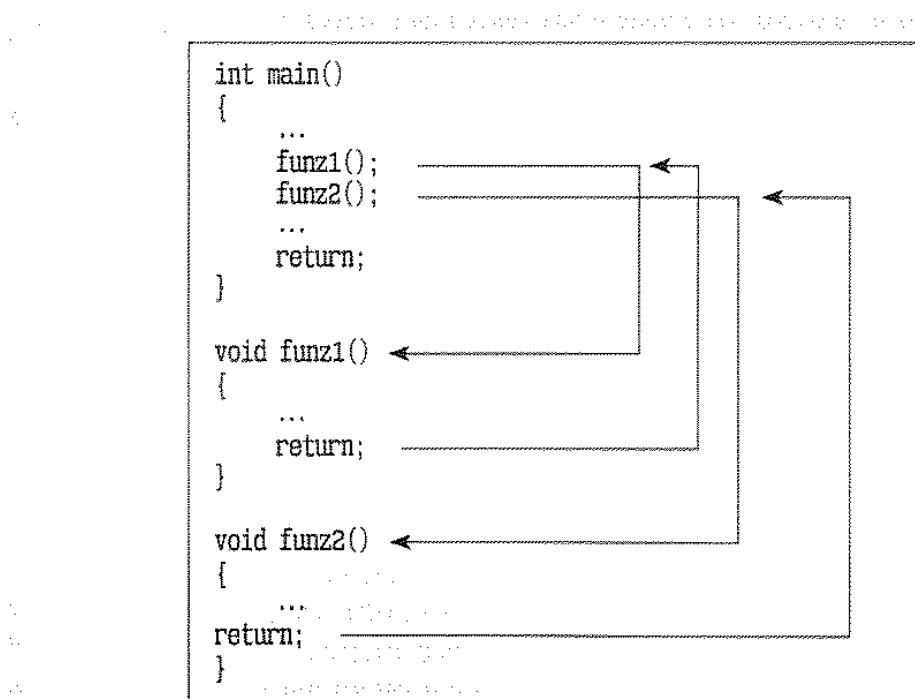
Una funzione va in esecuzione quando viene *chiamata* (o *invocata*) dalla `main()` o da un'altra funzione.



La funzione che chiama un'altra funzione è detta *funzione chiamante* e la funzione controllata si denominated *funzione chiamata*.

La funzione chiamata inizia con la prima istruzione dopo la parentesi graffa aperta e termina con la prima istruzione `return` o con l'ultima istruzione prima della parentesi graffa chiusa (Figura 5.2).

```
void funz1()
{
    cout << "Seconda funzione \n";
}
void funz2()
{
    cout << "Terza funzione \n";
}
```



**Figura 5.2**  
Traccia di chiamate di funzioni.

```

int main()
{
    cout << "Prima funzione chiamata: main() \n";
    funz1();
    funz2();
    cout << "main termina";
    return 0;
}

```

L'output di questo programma è:

```

Prima funzione chiamata main()
Seconda funzione
Terza funzione
main termina

```

Il formato `funz()` senza argomenti indica espressamente che la funzione non accetta argomenti.

### Esempio 5.2

Scrivere una funzione `int max(int, int)` che restituisca il maggiore fra due numeri interi letti in input. Scrivere inoltre un programma per testarne le funzionalità.

```

int max(int x, int y)
{
    if (x < y) return y;
    else return x;
}

int main()
{
    int m, n;
    cout << "Per uscire immetta 0\n";
    do {
        cout << "Per favore immetta due numeri\n";
        cin >> m >> n;
        cout << "Il maggiore fra i due è " << max(m, n) << endl;
    } while(m != 0);
    return 0;
}

```

### Esecuzione di prova

```

Per uscire immetta 0
Per favore immetta due numeri
23 45
Il maggiore fra i due è 45
Per favore immetta due numeri

```

```
-23 -12
```

Il maggiore fra i due è -12  
Per favore immetta due numeri

```
0 9
```

Il maggiore fra i due è 9

### Esempio 5.3

Scrivere una funzione `double media(double, double)` per calcolare la media aritmetica di due numeri letti in input.

```
double media(double x1, double x2)
{
    return(x1 + x2)/2;
}

int main()
{
    double num1, num2;
    cout << "Introdurre due numeri reali:";
    cin >> num1 >> num2;
    cout << "Il valore medio è " << media(num1, num2) << endl;
    return 0;
}
```

### Esecuzione di prova

```
Introdurre due numeri reali: 32.14 45.23
Il valore medio è 38.685
```

## 5.3 Prototipi delle funzioni

Per poter mandare in esecuzione una funzione bisogna prima averla definita. A volte però la funzione potrebbe essere definita in altri programmi che verranno poi collegati a quello dove la si vuole utilizzare; ma come potrebbe il compilatore accettare la chiamata a una funzione definita altrove? La soluzione è nel concetto di "dichiarazione": se non è già stata definita, la funzione deve almeno essere "dichiarata". La dichiarazione di una funzione si dice *prototipo* della funzione ed è simile alla definizione, ma:

1. non ha il corpo (perché esso è definito altrove);
2. deve specificare il tipo dei parametri formali ma non il loro nome (perché tanto non ha il corpo e quindi i nomi non servono);
3. l'intestazione deve terminare con il punto e virgola (;

**Prototipo**

```
tipo Ritorno nome_funzione (parametri_formali);
```

`tipo Ritorno` tipo del valore restituito dalla funzione

*nome\_funzione* nome della funzione  
*parametri\_formali* tipi degli argomenti formali della funzione, separati per virgole

Il prototipo fornisce sufficienti informazioni al compilatore per verificare che la funzione viene chiamata correttamente rispetto al numero e al tipo dei parametri che utilizza e rispetto al tipo restituito. Se c'è qualche incongruenza il compilatore la segnala come errore. È obbligatorio mettere un punto e virgola alla fine.

```
double Fahr2Celsius(double tempFahr); // prototipo valido anche se
                                         // tempFahr non è necessario
int max(int, int); // prototipo valido
```

Le dichiarazioni si collocano normalmente all'inizio del programma, prima della definizione della `main()`, mentre le definizioni si collocano normalmente dopo la `main()`. C/C++ distingue bene i concetti di *dichiarazione* e *definizione*. Quando si *dichiara* un'entità, se ne fornisce il nome e se ne elencano le caratteristiche, mentre quando si *definisce* un'entità si riserva anche spazio di memoria per essa. Rispetto alla dichiarazione, che informa solo dell'esistenza di una certa entità, la definizione la crea in memoria. Nel caso di una funzione, la *dichiarazione* ne contiene solo l'intestazione, mentre la *definizione* ne contiene anche il codice; per esempio, il programma dell'esercizio precedente andrebbe scritto così:

```
double media(double, double);

int main()
{
    double num1, num2;
    cout << "Introdurre due numeri reali:";
    cin >> num1 >> num2;
    cout << "Il valore medio è " << media(num1, num2) << endl;
    return 0;
}

double media(double x1, double x2)
{
    return(x1 + x2)/2;
}
```

### Dichiarazione

- Prima che una funzione possa essere invocata, deve essere almeno *dichiaratà* (se non completamente *definita*).
- La dichiarazione di una funzione (detta anche *prototipo* della funzione) è l'intestazione seguita da un punto e virgola, magari senza nominare i parametri formali.

*tipo\_risultato nome (tipo\_param1, tipo\_param2, ...);*

## 5.4 Passaggio di parametri alla funzione

Un parametro attuale può essere un valore costante, una variabile o un'espressione generica. In C++ ci sono due modi per passare variabili come argomenti attuali alle funzioni: "per valore" o "per riferimento". Supponiamo di avere una procedura `cerchio` con tre argomenti:

```
void cerchio(int x, int e, int diametro);
```

quando si invoca `cerchio` si debbono passarle tre parametri *attuali*, per esempio:

```
cerchio(25, y, giri*4);
```

dove il secondo parametro è una variabile. Il dubbio che può nascere è: cosa viene realmente passato alla funzione, la variabile o il valore che essa contiene nel momento in cui viene chiamata la funzione? Nel primo caso parremmo di passaggio per riferimento, nel secondo di passaggio per valore.

### 5.4.1 Passaggio di parametri per valore

In C++, come in C, le variabili vengono passate essenzialmente "per valore". *Passaggio per valore* significa che all'atto della chiamata la funzione riceve i valori delle variabili, non le variabili stesse. Questo significa che se la funzione modificherà il valore del suo argomento formale, queste modifiche non interesseranno in alcun modo la variabile che le era stata passata.

La Figura 5.3 esemplifica il passaggio di un argomento per valore; non viene passata la variabile reale *i* bensì il suo valore, 6.

È importante sottolineare che la funzione chiamata non può modificare il valore della variabile che le è stata passata come argomento attuale. Questo esempio mostra come, passando variabili per valore, le modifiche al para-

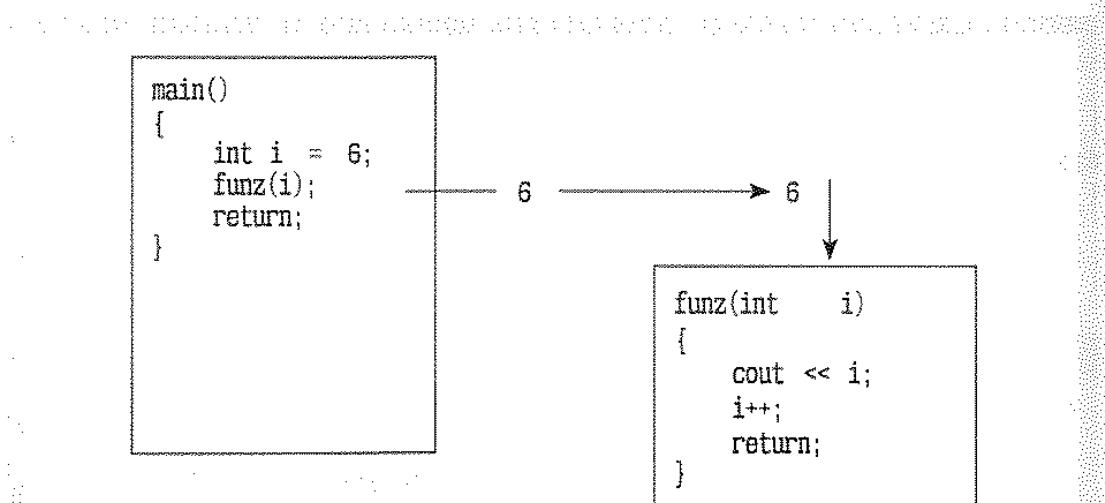


Figura 5.3

Passaggio per valore della variabile *i*.

metri effettuate dalla funzione non hanno alcun effetto sulla variabile passata alla funzione:

```
void ProvaPerValore(int);
int main(void)
{
    int n = 10;
    cout << "Prima di chiamare ProvaPerValore, n = " << n << endl;
    ProvaPerValore(n);
    cout << "Dopo la chiamata a ProvaPerValore, n = " << n << endl;
}

void ProvaPerValore(int i)
{
    cout << "Dentro ProvaPerValore, i = " << i << endl;
    i = 999;
    cout << "Adesso cambio il valore, i = " << i << endl;
}
```

## Esecuzione

```
Prima di chiamare ProvaPerValore, n = 10
Dentro ProvaPerValore, i = 10
Adesso cambio il valore, i = 999
Dopo la chiamata a ProvaPerValore, n = 10
```

### 5.4.2 Passaggio di parametri per riferimento

Supponiamo di voler scrivere una procedura per scambiare fra loro i valori di due variabili che passeremo come argomenti attuali alla chiamata. La procedura potrebbe avere questa struttura:

```
void scambia (int a, int b)
{
    int ausiliare = a;
    a = b;
    b = ausiliare;
}
```

e potremmo mandarla in esecuzione con una chiamata del tipo:

```
int i = 3, j = 4;
cout << "i = " << i << " e j = " << j << endl;
scambia (i, j);
cout << "i = " << i << " e j = " << j << endl;
```

Ma il risultato sarebbe frustrante:

```
i = 3 e j = 4
i = 3 e j = 4
```

Questo perché agli argomenti formali *a* e *b* non vengono passate le variabili *i* e *j* ma i valori che loro contengono all'atto della chiamata.

Quando una funzione deve modificare il valore della variabile passatagli come argomento si deve utilizzare il metodo del passaggio per *riferimento*. Con questo metodo non viene in realtà passata la variabile ma *il valore del suo indirizzo in memoria*, cioè il suo riferimento. Dentro la funzione esso sarà poi utilizzato per indirizzare la variabile esterna che dovrà essere modificata.

Questa tecnica può essere implementata in due modi: mediante i puntatori o mediante i riferimenti. La tecnica dei *puntatori* è ereditata dal C, mentre la tecnica dei *riferimenti* è stata introdotta dal C++ (derivandola dal Pascal del 1971). Illustriamo le due tecniche con l'esempio di una funzione che deve scambiare fra loro i valori di due variabili prese in ingresso come parametri attuali.

#### Metodo dei puntatori

```
void scambia_con_puntatori(int* a, int* b)
{
    int ausiliare = *a;
    *a = *b;
    *b = ausiliare;
}
```

La funzione *scambia\_con\_puntatori(int\* a, int\* b)* ha due argomenti formali, *a* e *b*, che non sono di tipo *intero* ma di tipo **puntatore a intero**. I puntatori sono variabili destinate a contenere indirizzi di memoria. "Puntatore a interi" significa "che contengono indirizzi di variabili di tipo *int*". Il corpo della funzione utilizza poi le espressioni *\*a* e *\*b* per **dereferenziare** i puntatori, cioè per leggere gli indirizzi dentro i puntatori e, tramite essi, risalire alle variabili di tipo *int* (in questo caso) da loro referenziate. Alla chiamata non verranno passate le variabili (ovvero i loro nomi) ma i loro **indirizzi** in memoria. L'indirizzo di memoria di una variabile si indica anteponendo l'operatore **&** (*ampersand*) al nome della variabile.

```
int i = 3, j = 50;
cout << "i = " << i << " e j = " << j << endl;
scambia_con_puntatori(&i, &j);
cout << "i = " << i << " e j = " << j << endl;
```

#### Metodo dei riferimenti

```
void scambia_con_riferimenti(int& m, int& n)
{
    int ausiliare = m;
    m = n;
    n = ausiliare;
}
```

La funzione *scambia(int& m, int& n)* ha come argomenti formali due **riferimenti a interi**, *m* e *n*. Un riferimento a una variabile è semplicemente un altro nome

della variabile stessa. Se ora chiamiamo la funzione passandole due variabili, per esempio, i e j, la funzione userà m e n come *alias* di i e j. Qualunque modifica effettuata dalla funzione su m e n avrà effetto su i e j.

```
int i = 3, j = 50;
cout << " i = " << i << " e j = " << j << endl;
scambia_con_riferimenti(i, j);
cout << "i = " << i << " e j = " << j << endl;
```

Cerchiamo di riassumere il comportamento delle tre procedure con un esempio:

```
void scambia(int, int);
void scambia_con_puntatori(int*, int*);
void scambia_con_riferimenti(int&, int&);

int main()
{
    int i=10, j=20;
    scambia(i, j);
    cout << "scambio errato i=" << i << " e j=" << j << endl;
    int i=10, j=20;
    scambia_con_puntatori(&i, &j);
    cout << "scambio puntatori i=" << i << " e j=" << j << endl;
    int i=10, j=20;
    scambia_con_riferimenti(i, j);
    cout << "scambio riferimenti i=" << i << " e j=" << j << endl;
}

void scambia(int a, int b)
{
    int ausiliare = a;
    a = b;
    b = ausiliare;
}

void scambia_con_puntatori(int* a, int* b)
{
    int ausiliare = *a;
    *a = *b;
    *b = ausiliare;
}

void scambia_con_riferimenti(int& a, int& b)
{
    int ausiliare = a;
    a = b;
    b = ausiliare;
}
```

L'esecuzione del programma produrrà:

```
scambio errato i=10 e j=20
scambio puntatori i=20 e j=10
scambio riferimenti i=20 e j=10
```

Di qualunque tipo essi siano, si può aggiungere lo specificatore `const` alla descrizione di un parametro formale per indicare al compilatore che esso è di sola lettura. Se il codice della funzione tenta di modificare questo parametro la compilazione produce un messaggio d'errore.

```
void falso(const int item, const char& car)
{
    item = 123;           // Errore a tempo di compilazione
    car = 'A';           // Errore a tempo di compilazione
}
```

## 5.5 Argomenti di default

Normalmente alla funzione si passa un numero di argomenti pari a quello definito nella sua intestazione. Tuttavia, in C++ è possibile definire funzioni in cui alcuni argomenti assumono un valore di *default* (per difetto di specifica). Se all'atto della chiamata non viene passato alcun valore per quel parametro, allora la funzione assumerà per lui il valore di default stabilito nell'intestazione. L'unica restrizione è che questi argomenti di default devono raggrupparsi a destra nell'intestazione. Il valore di default deve essere un'espressione costante.

L'esempio seguente mostra come utilizzare argomenti di default:

```
void asterischi(int fila, int col, int num, char c = "*");
```

Si può chiamare la funzione `asterischi` in due modi:

```
asterischi(4, 0, 40);
```

se si vuole utilizzare il quarto argomento di default, oppure

```
asterischi(4, 0, 40, '#');
```

se si vuole cambiare il carattere di default (\*) in #.

Un altro esempio più indicativo:

```
char funzdef(int arg1=1, char c='A', float f_val=45.7f);
```

Si può invocare `funzdef` con qualunque delle seguenti istruzioni:

```
funzdef(9, 'Z', 91.5); //Annulla tutti e tre gli argomenti di default
funzdef(25, 'W');    //Annulla i primi due argomenti di default
funzdef(50);         //Annulla il primo argomento di default
funzdef();           //Utilizza i tre argomenti di default
```

che sono equivalenti a:

```
funzdef(9, 'Z', 91.5);
funzdef(25, 'W', 45.7);
funzdef(50, 'A', 45.7);
funzdef(1, 'A', 48.7);
```

Il vincolo è che se si omette un argomento bisogna omettere anche tutti quelli alla sua destra. Per esempio, la seguente chiamata alla funzione non è corretta:

```
funzdef( , 'Z', 99.99);
```

Si deve quindi avere cura di collocare gli argomenti in un ordine per cui non capiti mai che un argomento di default stia a destra di un argomento specificato.

```
f()
f(a);
f(a, b);
f(a, b, c);
```

#### Esempio 5.4

La procedura scrive\_car ha due parametri. Il primo indica il carattere da scrivere a schermo e il secondo indica il numero di volte che quel carattere deve essere visualizzato.

```
void scrive_car(char c, int num = 1)
{
    for(int i = 1; i <= num; i++) cout << c;
}
```

Al parametro num si dà per default il valore 1. Quindi se non si specifica un valore per questo parametro, il carattere passato verrà visualizzato una sola volta. Sono chiamate valide:

```
scrive_car('x', 4)      // scrive quattro 'x'
scrive_car('e');        // scrive una 'e'
```

#### Esempio 5.5

```
void f(int a = 10, int b = 20, int c = 30)
{
    cout << "a = " << a << " b = " << b << " c = " << c << endl;
}

int main(void)
{
    f();
    f(1);
    f(1, 5);
    f(1, 2, 3);
}
```

### Esecuzione

```
a = 10 b = 20 c = 30
a = 1 b = 20 c = 30
a = 1 b = 5 c = 30
a = 1 b = 2 c = 3
```

- Gli argomenti di default si debbono passare per valore (e non per riferimento).
- I valori degli argomenti di default *non possono essere variabili*. In altre parole, se si dichiara `int n` e poi `int x = n`, si rifiuterà `n` come parametro per difetto. Tuttavia, se `n` si dichiara con `const int n=1` allora si accetta la definizione di default.
- Gli argomenti a destra di un argomento di default debbono essere anch'essi di default.

## 5.6 Funzioni in linea (inline)

Le funzioni in linea servono per aumentare la velocità del programma. Il loro uso è conveniente quando la funzione si richiama parecchie volte nel programma, per esempio all'interno di un ciclo, e il suo codice è breve.

Una funzione normale è un blocco di codice mandato in esecuzione da un'altra funzione. L'indirizzo dell'istruzione successiva alla chiamata della funzione viene memorizzato nello "stack" in maniera tale da potervi far ritorno quando l'esecuzione della funzione sarà conclusa. Dopodiché il flusso del programma salta all'indirizzo della prima istruzione della funzione. Il blocco di codice della funzione terminerà con un'istruzione speciale che recupererà dallo stack l'indirizzo di ritorno e manderà in esecuzione l'istruzione successiva alla chiamata di funzione nel programma chiamante.

Per una *funzione in linea* il compilatore ricopia realmente il codice della funzione in ogni punto in cui essa viene invocata. Il programma verrà così eseguito più velocemente perché non si dovrà eseguire il codice associato alla chiamata alla funzione. Tuttavia, ogni ripetizione della funzione richiede memoria, perciò il programma aumenta la sua dimensione. Per esempio, se si invoca dieci volte una funzione in linea, il compilatore inserisce dieci copie della funzione nel programma. Se questa occupa 1K, la dimensione del programma cresce di 10K.

Per creare una funzione in linea si deve inserire la parola riservata `inline` all'inizio dell'intestazione, come mostrato nel prossimo esempio.

### Esempio 5.6

Scrivere una funzione in linea `VolCono()` che calcoli il volume della figura geometrica *Cono* ( $V = 1/3 \pi r^2 h$ ) e la si testi in un programma principale.

```
const float PiGreco = 3.141592;
```

```

inline float VolCono(float raggio, float altezza)
{return ((PiGreco * (raggio * raggio) * altezza) / 3.0);}

int main()
{
    float raggio, altezza;
    cout << "Introduca raggio del cono";
    cin >> raggio;
    cout << "Introduca altezza del cono";
    cin >> altezza;
    cout << "Il volume del cono è: " << VolCono(raggio, altezza);
    return 0;
}

```

### Esecuzione di prova

Introduca raggio del cono 23  
 Introduca altezza del cono 23  
 Il volume del cono è: 12741.2

## 5.7 Visibilità e “storage classes” in C++

La *visibilità* di una variabile è la zona del programma in cui essa è accessibile.  
 Esistono quattro tipi di visibilità:

1. *di programma*;
2. *di file*;
3. *di funzione*;
4. *di blocco*.

Normalmente la visibilità è determinata dal punto in cui la variabile viene definita all'interno del programma, ma gli specificatori (*storage classes*)

static  
 extern  
 auto  
 register

possono modificarla.

### 5.7.1 Visibilità di programma

Le variabili *globali*, cioè quelle definite al di fuori di qualunque funzione, hanno *visibilità di programma* e possono essere riferite da qualunque funzione del programma (a partire dal suo punto di definizione) nello stesso file sorgente, ma anche da altre funzioni che fossero a esso collegate (dal linker) a far parte dello stesso programma eseguibile, a patto che in questi altri file quelle variabili siano state definite con lo specificatore *extern*.

### 5.7.2 Visibilità di funzione

Le variabili dichiarate dentro il corpo della funzione si dicono *locali* alla funzione e non possono essere riferite dal di fuori della funzione stessa. Esse esistono in memoria solo dal momento che viene lanciata la funzione e solo fintantoché la funzione rimane in esecuzione. Quindi due o più funzioni possono dare lo stesso nome a una propria variabile locale. Si esamini la Figura 5.4. Ci sono due funzioni che definiscono `x1` come variabile locale, tuttavia, qualunque operazione su `x1` dentro `funzione1()` non riguarda la `x1` di `funzione2()` e viceversa, perché sono a tutti gli effetti variabili diverse che esistono in memoria in diversi punti e anche (se la CPU è una sola) in diversi intervalli di tempo. Le modifiche che una funzione fa sulla sua variabile locale non hanno alcun effetto sull'omonima variabile dell'altra funzione, che in quel momento non esiste neanche.

Ma questo è vero a meno che una delle due funzioni non definisca la sua variabile locale utilizzando lo specificatore `static`. Una variabile locale, definita come `static`, nasce quando viene chiamata la funzione, ma non muore al termine della funzione e mantiene il suo valore tra diverse chiamate della

```
int x0;
funzione1();      // prototipo funzione1
funzione2();      // prototipo funzione2
```

```
main
```

```
{
```

```
...
```

```
funzione1()
```

```
[
```

```
    int x1;      // variabile locale
```

```
...
```

```
]
```

```
funzione2()
```

```
[
```

```
    int x1;
```

```
...
```

```
]
```

Figura 5.4

`x0` è globale al programma completo; `x1` è locale tanto a `funzione1()` come a `funzione2()`, ma si trattano come variabili indipendenti.

stessa funzione. Quindi, al contrario delle normali variabili locali, una variabile locale statica viene inizializzata una volta sola (la prima).

#### Esempio 5.7

```
void test()
{
    static int var = 0; // var è statica
    ++var;
    cout << var << endl;
}

int main()
{
    test();
    test();
    test();
    return 0;
}
```

z  
1  
2  
3

se si toglie lo specificatore **static**:

#### Esecuzione

1  
1  
1

Un uso frequente è quello di utilizzare una variabile statica per accumulare somme parziali attraverso successive chiamate di funzione, per esempio:

```
void RisultatiTotali(float Valore)
{
    static float Sommatoria;
    Sommatoria += Valore;
}
```

#### 5.7.3 Visibilità di file (sorgente)

Un altro uso dello specificatore **static** è quello di *occultare variabili globali da altri file sorgenti*. Per rendere una variabile globale privata al file sorgente (pertanto non utilizzabile da altri moduli di codice in altri file sorgenti) la si può far precedere dalla keyword **static**. Così facendo essa rimarrà privata al file sorgente in cui è definita e non la si potrà dichiarare come **extern** in un altro file sorgente. Le variabili con questa visibilità si possono riferire dal

punto in cui sono dichiarate fino alla fine del file sorgente. Se un file sorgente ha più di una funzione, tutte le funzioni che seguono la dichiarazione della variabile possono riferirla.

#### 5.7.4 Visibilità di blocco

Una variabile dichiarata in un blocco ha *visibilità di blocco* e può essere referenziata in qualunque parte del blocco, dal punto in cui è dichiarata fino alla fine. Per esempio, le variabili locali dichiarate dentro una funzione hanno visibilità nel blocco corpo della funzione, dal punto in cui sono definite in poi.

Una variabile locale dichiarata in un blocco annidato è visibile solo all'interno di quel blocco.

```
void funz(int j)
{
    auto int j;
    if (j > 3)
    {
        int i;
        for (i = 0; i < 50; i++) funz2(i);
    }
    //qui i non è visibile ma j si
}
```

Le variabili locali a una funzione si dicono anche *automatiche* (auto) perché si assegna loro spazio in memoria automaticamente alla chiamata della funzione e si rimuovono automaticamente all'uscita della funzione. La parola riservata auto nell'esempio precedente è inutile.

## 5.8 Specificatore di accesso auto

Lo specificatore auto è stato quello più inutilizzato dai primi C++ poiché se una variabile è definita dentro una funzione essa è «automaticamente» auto e quindi non c'è bisogno di specificarlo. Ma dal C++11 in poi esso è passato dall'essere il più inutile a essere il più usato poiché è invalso l'uso della **tipizzazione automatica**. La deduzione di tipo automatico è oggi una delle caratteristiche più importanti e utilizzate nel C++. I nuovi standard del linguaggio hanno reso possibile l'uso dello specificatore auto per indicare che *il programmatore vuole lasciare al compilatore la facoltà di stabilire il tipo effettivo*, sia per le variabili locali alla funzione sia per il tipo restituito dalla funzione stessa. Le versioni standard future amplieranno forse l'uso di auto a un numero ancora maggiore di casi. L'uso di auto in questi contesti ha diversi importanti vantaggi. Gli sviluppatori dovrebbero esserne consapevoli e preferire auto quando possibile. Esempi:

```
auto i = 42; // il compilatore assegna automaticamente int a i
auto i; // ERRORE: il compilatore non può desumere il tipo
auto d = 42.5; // il compilatore assegna double
```

```
auto s = "testo"; // il compilatore assegna char const *
double funzione();
    auto d = funzione(); // il compilatore assegna double a d
```

### 5.8.1 Specificatore di accesso extern

In C++ una funzione può utilizzare una variabile globale definita in un altro file sorgente semplicemente dichiarandola localmente con la parola riservata **extern**. In questo modo si indica al compilatore che la variabile è definita in un altro file sorgente che sarà *collegato* assieme (dal linker).

file ester1.cpp - variabile globale

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

void leggiReale();

float f;

int main()
{
    leggiReale();
    cout << "Valore di float =" << f;
    return 0;
}
```

file ester2.cpp - variabile esterna

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

void leggiReale()
{
    extern float f;
    cout << "Introduca valore in virgola mobile";
    cin >> f;
}
```

Nel file ester2.cpp la dichiarazione esterna di float f indica al compilatore che f è stata definita in un altro file. Quando questi file verranno collegati assieme dal linker le dichiarazioni si combineranno in modo da riferirsi alla stessa posizione di memoria.

Se una definizione di variabile comincia con la parola riservata **extern**, essa non è in realtà una definizione ma solo una **dichiarazione** di variabile, ovvero non comporta allocazione di memoria (perché essa è già stata allocata con una definizione in altro punto del codice del programma). Ogni definizione di variabile è anche una dichiarazione di variabile, ma non viceversa. Una *variabile si definisce una volta sola*, ma si può *dichiarare più volte*.

### 5.8.2 Specificatore di accesso register

Con lo specificatore register si chiede al compilatore di porre la variabile in uno dei registri hardware del microprocessore. Si tratta di un suggerimento, non un ordine, e quindi nel caso di microprocessori con pochi registri interni (per esempio, la famiglia di microprocessori Intel) il compilatore può decidere di ignorare la richiesta. L'uso tipico di una variabile registro è come variabile di controllo di un ciclo:

```
register int k;
for (i = 0; i < 1000; i++)...
```

in questa maniera si riduce il tempo che la CPU richiede per cercare il valore della variabile in memoria. Una variabile registro non può essere globale.

## 5.9 Funzioni di libreria

Per default, **tutte le funzioni sono globali e sono quindi visibili da altri moduli di programma**. Si può comunque dichiarare una funzione static in modo da impedire di utilizzare quella funzione in altri moduli sorgente del programma. Questo torna utile in grandi progetti, quando si fa ampio uso contemporaneo sia di funzioni di librerie varie sia di funzioni native del programma, con rischi di conflitti di identificatori, sia di variabili sia di funzioni.

Tutte le versioni del linguaggio C/C++ contengono una grande raccolta di funzioni di libreria. Queste funzioni predefinite si dividono in gruppi; le funzioni che appartengono allo stesso gruppo sono raccolte e dichiarate nello stesso *header file*. L'inserimento di queste librerie nel programma è specificato mediante la direttiva #include.

Alcune librerie sono standard e il compilatore sa dove trovarle nel file system, e i relativi header file si includono con le parentesi angolari:

```
#include <iostream>
```

Altre librerie sono specifiche dell'applicazione che si sta compilando, e di queste il compilatore non sa nulla; nell'includere l'header file bisogna indicare tutto il *pathname* fra doppi apici (per significare che non si tratta di libreria standard) invece che fra parentesi angolari:

```
#include "PATH/mia_libreria"
```

ovviamente se l'header file si trova nella stessa cartella del file sorgente che si sta compilando il percorso è nullo e si scrive semplicemente:

```
#include "mia_libreria"
```

Alcune librerie standard ereditate dal C sono obsolete, mentre quelle del C++ sono in continua evoluzione a causa del succedersi degli standards. Non è possibile descrivere in forma compiuta la libreria standard del C++ non solo perché evolve (lo standard è passato dal ISO-ANSI C++98 al C++11, poi al

C++11, al C++14, quindi al C++17 e ora siamo al C++20), ma anche perché essa è enorme: esistono in commercio libri di più di mille pagine interamente dedicati.

Per chiamare una funzione bisogna conoscerne numero, sequenza e tipo degli argomenti, nonché il tipo del suo valore di ritorno, informazioni queste tutte contenute nel prototipo della funzione. Quello che qui possiamo fare è elencare alcune delle librerie più usate.

### **<cctype>**

definisce un gruppo di funzioni per la manipolazione di caratteri.

**isalpha(c)** restituisce true se e solo se c è una lettera maiuscola o minuscola.

**islower(c)** restituisce true se e solo se c è una lettera minuscola.

**isupper(c)** restituisce true se e solo se c è una lettera maiuscola.

**isdigit(c)** restituisce true se e solo se c è una cifra (cioè un carattere da 0 a 9).

**isxdigit(c)** restituisce true se e solo se c è una cifra esadecimale (da 0 a 9, da A a F, da a a f).

**isalnum(c)** restituisce true se e solo se c è una cifra o un carattere alfabetico (maiuscolo o minuscolo).

**iscntrl(c)** restituisce true se e solo se c è un *carattere di controllo* (codici ASCII da 0 a 31).

**isgraph(c)** restituisce true se e solo se c è un carattere stampabile (non di controllo) eccetto lo spazio.

**isprint(c)** restituisce true se e solo se c è un carattere stampabile (ASCII da 21 a 127) incluso lo spazio.

**ispunct(c)** restituisce true se e solo se c è qualunque carattere di interpunzione.

**isspace(c)** restituisce true se e solo se c è uno spazio, nuova riga (\n), ritorno di carrello (\r), tabulazione (\t) o tabulazione verticale (\v).

**tolower(c)** converte la lettera c in minuscola, se non lo è già.

**toupper(c)** converte la lettera c in maiuscola, se non lo è già.

### Esempio 5.8

```
#include <cctype>
int main()
{
    char risp;
    cout << "È maschio o femmina (m/f)?";
    cin >> risp;
```

```

risp=toupper(risp);
switch (risp)
{
    case 'M': cout << "È maschio \n"; break;
    case 'F': cout << "È femmina \n"; break;
    default: cout << "Non è né maschio né femmina \n";
}
return 0;
}

```

&lt;cmath&gt;

**definisce un gruppo di funzioni matematiche****ceil(x)** arrotonda all'intero maggiore.**floor(x)** arrotonda all'intero minore.**fabs(x)** restituisce il valore assoluto di x.**fmod(x, e)** calcola il resto in virgola mobile per la divisione x/e.**Esempio 5.9**

```

#include <cmath>
int main ()
{
    cout << "fmod(5.3 / 2) = " << fmod (5.3,2) << endl;
    cout << "fmod(18.5 / 4.2) = " << fmod (18.5,4.2) << endl;
    return 0;
}

```

**Esecuzione**

```

fmod(5.3 / 2) = 1.3
fmod(18.5 / 4.2) = 1.7

```

**pow(x, y)** calcola x elevato a e ( $x^y$ ). Se x è minore o uguale a zero, y deve essere un intero. Se x è uguale a zero, y non può essere negativo.**sqrt(x)** restituisce la radice quadrata di x. x deve essere maggiore o uguale a zero.**acos(x)** restituisce l'arco coseno di x; x deve essere compreso fra -1 e 1.**asin(x)** restituisce l'arco seno di x; x deve essere compreso fra -1 e 1.**atan(x)** restituisce l'arco tangente di x.**atan2(x, e)** restituisce l'arco tangente di x diviso e.**cos(x)** restituisce il coseno dell'angolo x (x si esprime in radianti).

`sin(x)` restituisce il seno dell'angolo  $x$  ( $x$  si esprime in radianti).

`tan(x)` restituisce la tangente dell'angolo  $x$  ( $x$  si esprime in radianti).

`exp(x)`, `expl(x)` restituiscono l'esponenziale  $e^x$ , dove  $e$  è il numero di Nepero, base dei logaritmi naturali (valore approssimato 2.718282); `expl(x)` restituisce  $e^x$  su un long double.

`log(x)`, `logl(x)` restituiscono il logaritmo naturale di  $x$ ; `logl(x)` su un long double.

`log10(x)`, `log10l(x)` restituiscono il logaritmo decimale di  $x$ ; `log10l(x)` su un long double.  $x$  deve essere positivo.

#### <cstdlib>

definisce diverse funzioni di uso generale, tra cui la gestione dinamica della memoria, la generazione di numeri casuali, la comunicazione con l'ambiente, l'aritmetica dei numeri interi, la ricerca, l'ordinamento e la conversione.

`rand()` restituisce un numero aleatorio che varia da 0 a `RAND_MAX`, dove la costante `RAND_MAX` è definita in `<cstdlib>`; ogni volta che si richiama `rand()` nello stesso programma, si ottiene un numero differente, tuttavia il programma restituisce la stessa sequenza di numeri aleatori a ogni sua esecuzione; un modo per superare questo problema è quello di chiamare la funzione `srand()`.

`srand(seme)` inizializza il generatore di numeri aleatori in base al valore dell'argomento `seme`; s'intende che questo valore sia esso stesso casuale o letto in input.

#### Esempio 5.10

```
#include <cstdlib> // srand, rand
#include <ctime> // time
int main ()
{
    int pensato, tentato;
    srand (time(NULL)); // inizializza il seme
    /* genera il numero segreto da 1 a 10: */
    pensato = rand() % 10 + 1;
    do {
        cout << "Indovina che numero ho pensato, da 1 a 10: ";
        cin >> tentato;
        if (pensato < tentato) cout << "Troppo grande!\n";
        else if (pensato > tentato) cout << "Troppo piccolo\n";
    } while (pensato != tentato);
    cout << "Congratulazioni! C'hai azzeccato!";
    return 0;
}
```

## Esecuzione di prova

```
Indovina che numero ho pensato, da 1 a 10: 3
Troppo piccolo
Indovina che numero ho pensato, da 1 a 10: 5
Troppo piccolo
Indovina che numero ho pensato, da 1 a 10: 7
Troppo grande!
Indovina che numero ho pensato, da 1 a 10: 6
Congratulazioni! C'hai azzeccato!
```

### `<ctime>`

definisce strutture e funzioni per manipolare date e ora. La data si riferisce al calendario gregoriano.

Le funzioni `time`, `clock`, `_strdate` e `_strtime`, restituiscono, rispettivamente, l'ora attuale come numero di secondi trascorsi dalla mezzanotte dell'1 gennaio 1970 (ora universale, GMT), il tempo di CPU impiegato dal processo invocante, la data e l'ora attuale.

La libreria definisce la struttura dati:

```
struct tm
{
    int tm_sec;      /* secondi */
    int tm_min;      /* minuti */
    int tm_hour;     /* ore */
    int tm_mday;     /* giorno del mese 1 a 31 */
    int tm_mon;      /* mese, 0 per Gen, 1 per Feb,... */
    int tm_year;     /* anno da 1900 */
    int tm_wday;     /* giorni della settimana da domenica (0-6) */
    int tm_yday;     /* giorno dell'anno dal 1 Gen (0-365) */
    int tm_isdst;    /* sempre 0 per gmtime */
};
```

nonché il tipo `time_t`, alias di tipo aritmetico fondamentale in grado di rappresentare i tempi. Per ragioni storiche, è generalmente implementato come valore integrale che rappresenta il numero di secondi trascorsi dalle ore 00:00, 1 gennaio 1970 UTC.

`clock()` restituisce il tempo di processore trascorso dall'inizio dell'esecuzione del programma; il valore restituito è espresso in *clock tick*, che sono unità di tempo di lunghezza costante ma specifica del sistema (funzione della costante `CLOCKS_PER_SEC`). L'epoca utilizzata come riferimento per l'orologio varia da un sistema all'altro, ma è correlata all'esecuzione del programma (generalmente il suo lancio). Per calcolare il tempo di elaborazione effettivo di un programma, il valore restituito da `clock` deve essere confrontato con un valore restituito da una precedente chiamata alla stessa funzione.

```
inizio = clock();
```

```
fine = clock();
```

```
tempo_di_esecuzione = fine - inizio;
```

`time(ora)` restituisce in ora il numero di secondi trascorsi dalla mezzanotte (00:00:00) dell'1 gennaio 1970. Il prototipo della funzione è: `time_t time(time_t *ora);`

`localtime(ora)` converte data e ora in una struttura di tipo `tm`. Il prototipo è: `struct tm *localtime(const time_t *tptr);`

`mkttime(t)` converte l'ora a un formato di calendario. Prende l'informazione dell'ora locale contenuta nella struttura `*tptr` e determina i valori di tutti i membri nel formato di tempo del calendario. Il suo prototipo è: `time_t mkttime(struct tm *tptr);`

#### Esempio 5.11

```
#include <ctime> // time, localtime, mkttime, struct tm
using namespace std;
int main ()
{
    time_t istante;
    struct tm * informazioni;
    int anno, mese, giorno;
    const char * giornoDellaSettimana[] = { "Domenica", "Lunedì",
                                            "Martedì", "Mercoledì",
                                            "Giovedì", "Venerdì", "Sabato"};
    cout << "Che anno era: "; cin >> anno;
    cout << "Che mese era: "; cin >> mese;
    cout << "Che giorno era: "; cin >> giorno;
    time ( &istante );
    informazioni = localtime ( &istante );
    informazioni->tm_year = anno - 1900;
    informazioni->tm_mon = mese - 1;
    informazioni->tm_mday = giorno;
    mkttime ( informazioni );
    cout << "Quel giorno era un "
        << giornoDellaSettimana[informazioni->tm_wday];
    return 0;
}
```

#### Esecuzione di prova

Che anno era: 2021

Che mese era: 1

Che giorno era: 14

Quel giorno era un Giovedì

## 5.10 Compilazione modulare

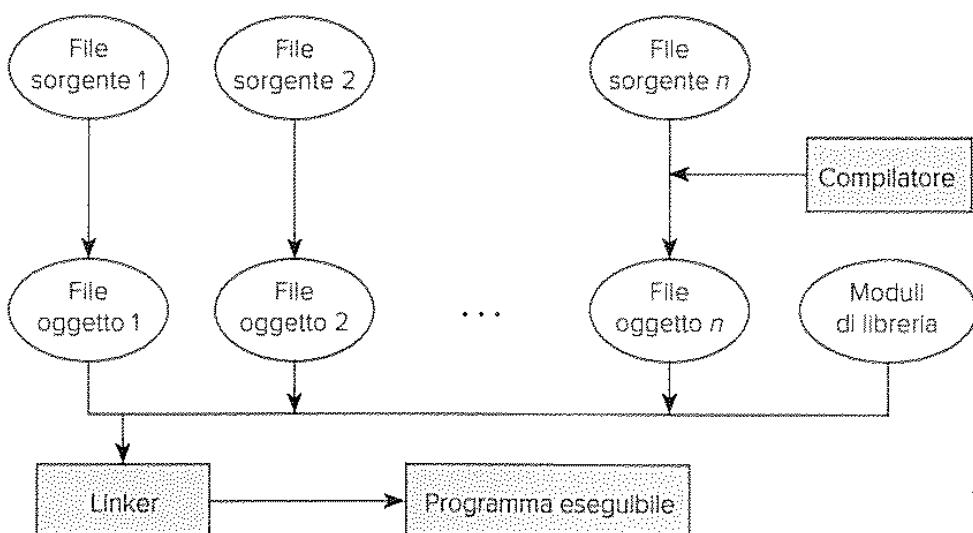
Quasi tutti gli esempi visti finora si trovavano in un unico file sorgente. I programmi si dividono in vari file sorgenti, anche chiamati *moduli*, ognuno dei quali può contenere una o più funzioni. Questi moduli verranno poi compilati separatamente ma collegati (*linked*) assieme. A ogni ricompilazione verranno in realtà ricompilati solo i moduli che sono stati modificati. La Figura 5.5 mostra come il linker produce l'eseguibile a partire dai vari moduli oggetto, ognuno dei quali si ottiene compilando il corrispondente modulo sorgente.

In ogni modulo si può referenziare funzioni definite in altri moduli. Come già detto, al contrario delle variabili, le funzioni sono globali per default, ma si può restringere la visibilità di una funzione alle sole funzioni dello stesso modulo utilizzando la parola riservata *static* per ridurre il rischio di avere funzioni con lo stesso nome. Per esempio, immaginiamo un programma molto grande che consta di parecchi moduli, nel quale si cerca un errore prodotto da una variabile globale; se la variabile è *static*, si può restringere la ricerca al solo modulo in cui è definita; altrimenti bisogna estendere l'indagine a tutti i moduli in cui essa può essere stata dichiarata (mediante la parola riservata *extern*).

Come regola generale, sono preferibili le variabili locali a quelle globali.

### Esempio 5.12

Supponiamo che il programma sia su due file: *MODULO1* e *MODULO2*.



**Figura 5.5**  
Compilazione modulare.

## MODULO1.cpp

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

int main()
{
    void f(int i), g(void); // Dichiarazioni (no definizioni)
    extern int n;          // Dichiarazione di n (no definizione)
    f(8);
    n++;
    g();
    cout << "Fine del programma. \n";
}
```

## MODULO2.cpp

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

int n = 100;           // Definizione di n (anche dichiarazione)
static int m = 7;
void f(int i)          // Definizione
{
    n += i + m;
}
void g()               // Definizione
{
    cout << "n = " << n << endl;
}
```

f e g si *definiscono* nel modulo 2 e si *dichiarano* nel modulo 1. Come compilare modularmente dipende dal compilatore e dalla configurazione IDE utilizzata. Senza IDE, utilizzando GCC da console si compila e manda in esecuzione come segue:

**Compilazione ed Esecuzione**

```
$ g++ MODULO1.cpp MODULO2.cpp -o prova
$ ./prova
n = 116
Fine del programma.
```

**5.11 Ricorsione**

Una *funzione ricorsiva* è una funzione che chiama sé stessa, direttamente o indirettamente. La *ricorsione diretta* è il processo per il quale una funzione invoca sé stessa nel proprio corpo. La *ricorsione indiretta* implica più di una

funzione, per esempio, una funzione `uno()` che chiama una funzione `due()` che a sua volta richiama `uno()`.

Un processo ricorsivo deve avere una condizione di terminazione, altrimenti si richiama all'infinito.

Un algoritmo tipicamente ricorsivo è il calcolo del fattoriale di un numero  $n$  ( $n!$ ).

$$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$$

perché il fattoriale di  $n$  si ottiene moltiplicando per  $n$  il fattoriale di  $n-1$ . La Figura 5.6 mostra la sequenza di invocazioni ricorsive della funzione fattoriale.

#### Esempio 5.13

Implementare in maniera ricorsiva la funzione fattoriale e testarne il funzionamento.

```
long Fattoriale(int numero)
{
    if (numero > 1)
        return numero * Fattoriale(numero-1);
    return 1;
}
int main()
{
```

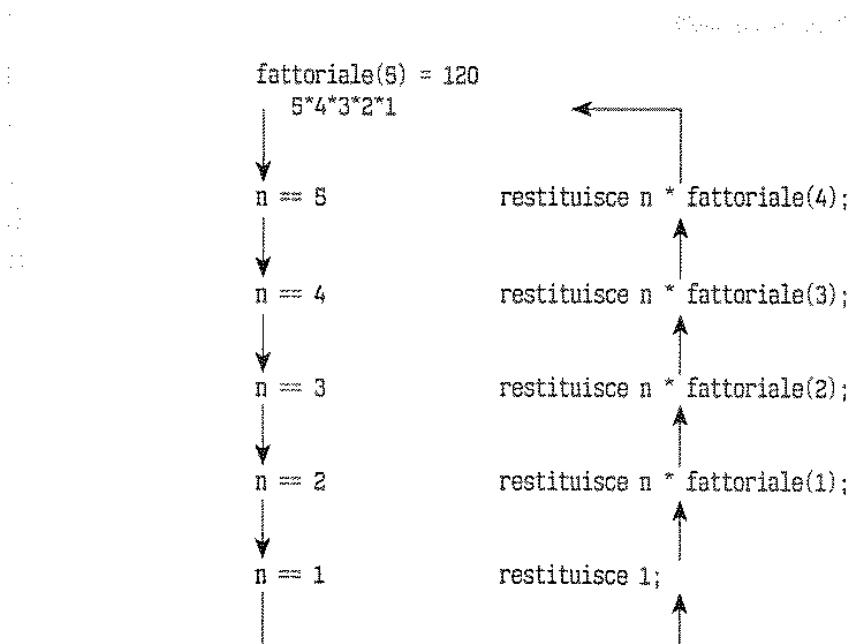


Figura 5.6

Chiamate a funzioni ricorsive per `fattoriale(5)`.

```

int num;
cout << "Per favore immetta un numero: ";
cin >> num;
cout << "Il fattoriale di " << num << " è " << Fattoriale(num) << endl;
return 0;
}

```

### Esecuzione

Per favore immetta un numero: 20  
Il fattoriale di 20 è 2432902008176640000

## 5.12 Sovraccaricamento delle funzioni

C++ supporta una delle funzionalità più importanti della programmazione: *il sovraccaricamento (overheading) delle funzioni*. Esso permette di utilizzare più funzioni con lo stesso nome, ma con almeno un argomento di tipo diverso e/o con un diverso numero di argomenti. Per capirne l'importanza si consideri, per esempio, questa funzione:

```

int quadrato(int x)
{
    return x * x;
}

```

Se si vuole implementare una funzione simile per processare un valore d'altro tipo, per esempio long o double, e il linguaggio non supporta l'overheading di funzioni, si debbono definire funzioni indipendenti con un nome diverso per ogni tipo, per esempio:

```

long quadratol(long x) {...};
double quadratod(double x) {...};
.....

```

grazie all'*overheading* possiamo conservare per esse lo stesso nome:

```

int quadrato(int x) {...};
long quadrato(long x) {...};
double quadrato(double x) {...};

```

Per mandare in esecuzione la funzione giusta si usa l'argomento del tipo giusto:

```

long raggio = 42500;
risultato = quadrato(raggio);

```

C++ determina quale tra le funzioni sovraccaricate deve chiamare, in funzione del numero e della sequenza dei tipi di parametro passati. Per risolvere ogni ambiguità C++ richiede che ogni funzione sovraccaricata abbia almeno un argomento di tipo diverso dalle altre (per corrispondenza di posizione),

altrimenti darà errore in fase di compilazione. Le regole che il C++ segue per selezionare la giusta funzione sovraccaricata sono:

1. se esiste, si seleziona la funzione che mostra la corrispondenza esatta tra il numero e i tipi dei parametri formali e attuali;
2. se tale funzione non esiste, si seleziona la funzione in cui il matching dei parametri avviene tramite una conversione automatica di tipo (come da int a long, o da float a double);
3. la corrispondenza dei tipi degli argomenti può venire anche forzata mediante *casting*.

#### Esempio 5.14

```
int Prova(int);
int Prova(int, int);
float Prova(float, float);
int main()
{
    int i = 7;
    int x = 4, y = 5;
    float a = 6.0, b = 7.0;
    cout << "Quadrato di " << i << " = " << Prova(i);
    cout << "\nProdotto " << x << " per " << y << " = " << Prova(x, y);
    cout << "\nMedia fra " << a << " e " << b << " = " << Prova(a, b);
}

int Prova(int valore) /* Quadrato */
{
    return (valore * valore);
}

int Prova(int valore1, int valore2) /* Prodotto */
{
    return(valore1 * valore2);
}

float Prova (float valore1, float valore2) /* Media */
{
    return ((valore1 + valore2) / 2);
}
```

#### Esecuzione

```
Quadrato di 7 = 49
Prodotto 4 per 5 = 20
Media fra 6 e 7 = 6.
```

#### Esempio 5.15

```
void seletipo(int i);
```

```

void seletipo(char *);
void seletipo(float f);
void seletipo(char);
void seletipo(int i, int j);
int main()
{
    seletipo(5);
    seletipo("Lancia Stratos");
    seletipo(3.65f);
    seletipo('A');
    seletipo(4, 5);
}
void seletipo(int i)
{
    cout << i << " è un valore di tipo intero \n";
}
void seletipo(char s[])
{
    cout << s << " è un valore di tipo stringa \n";
}
void seletipo(float f)
{
    cout << f << " è un valore di tipo tipo float \n";
}
void seletipo(char c)
{
    cout << c << " è un valore di tipo carattere \n";
}
void seletipo(int i, int j)
{
    cout << i << " e " << j << " sono 2 valori di tipo intero\n";
}

```

### Esecuzione

5 è un valore di tipo intero  
 Lancia Stratos è un valore di tipo stringa  
 3.65 è un valore di tipo tipo float  
 A è un valore di tipo carattere  
 4 e 5 sono 2 valori di tipo intero

Il sovraccaricamento delle funzioni è una forma di polimorfismo. Come si vedrà più avanti, il **polimorfismo** è uno dei concetti fondamentali della programmazione orientata agli oggetti.

### 5.13 Template di funzioni

Grazie al sovraccaricamento possiamo definire funzioni diverse con lo stesso nome, ma se il codice è lo stesso perché definire funzioni diverse solo perché i tipi di dato sono diversi?

I **template** (*function template*) forniscono un meccanismo per creare *funzioni generiche* (o, meglio, *per tipi generici*). Una funzione generica può supportare simultaneamente differenti tipi di dato per i suoi parametri. I *template* costituiscono una delle innovazioni del C++.

Consideriamo, per esempio, una funzione `abs(argomento)` che restituiscia il valore assoluto del suo argomento. Per implementare `abs()` in modo che possa accettare parametri `int`, `float`, `long` e `double`, si dovrebbero scrivere quattro funzioni diverse ma con lo stesso codice.

```
int abs(int x)
{
    return (x < 0) ? -x : x;
}

long abs(long x)
{
    return (x < 0) ? -x : x;
}

float abs(float x)
{
    return (x < 0) ? -x : x;
}

double abs(double x)
{
    return (x < 0) ? -x : x;
}
```

In C++ al posto delle quattro funzioni `abs()` si può scrivere un unico *template di funzione*:

```
template <class tipo> tipo abs(tipo x)
{
    return (x < 0) ? -x : x;
};
```

La parola riservata `template` indica al compilatore che è stato definito uno schema di funzione, il token `tipo` indica al compilatore che può essere sostituito dal tipo di dato vero appropriato: `int`, `float` ecc.

Un template di funzione ha il seguente formato:

```
template <class tipo> tipo_restituito func-name(lista parametri) [
    // corpo della funzione
]
```

**Esempio 5.16**

max dichiarata con un unico tipo generico T:

```
template <class T> T max (T a, T b)
{
    return a > b ? a : b;
}
```

**Esempio 5.17**

max dichiarata con due tipi generici T1 e T2:

```
template <class T1, class T2> T1 max (T1 a, T2 b)
{
    return a > b ? a : b;
}
```

Per capire un po' meglio il funzionamento consideriamo la funzione Funz()

```
template <class Tipo> Tipo Funz(Tipo arg1, Tipo arg2)
{
    // corpo della funzione Funz()
}

int main()
{
    int i, j;      // Definisce variabili intere
    float a, b;    // Definisce variabili float
    Funz(i, j); // uso del template Funz() con valori interi
    Funz(a, b); // uso del template Funz() con variabili float
}
```

Il programma dichiara un template Funz() e ne utilizza la funzione con argomenti interi (int) o reali (float). Quando il compilatore esamina la prima chiamata a Funz() vede che si tratta di un template con tipi generici e costruisce la corrispondente funzione vera che utilizza argomenti di tipo intero:

```
int Funz(int arg1, int arg2)
{
    // corpo della funzione Funz()
}
```

e lo stesso fa quando esamina la seconda chiamata, ma ovviamente costruisce la funzione con tipi diversi:

```
float Funz(float arg1, float arg2)
{
    // corpo della funzione Funz()
}
```

A motivo della loro genericità, i template sono candidati a divenire funzioni di libreria, come nell'Esempio 5.18 dove supponiamo di aver sistemato due template di funzione `min()` e `max()` in un header file.

#### Esempio 5.18

```
minmax.h

#ifndef MINMAX_H      // header guards
#define MINMAX_H      // evitano inclusioni multiple

template <typename T> T Max (T a, T b) {
    return a < b ? b: a;
}

template <typename T> T Min (T a, T b) {
    return a >= b ? b: a;
}

#endif
```

Segue un programma che utilizza i precedenti template (includendo il file che li contiene), tramite prototipi che il compilatore utilizza per scrivere i corpi delle funzioni reali.

```
#include "minmax.h"

int main () {
    int i = 39, j = 20;
    cout << "Il maggiore fra " << i << " e " << j << " è "
        << Max(i, j) << endl;

    double f1 = 13.5, f2 = 20.7;
    cout << "Il minore fra " << f1 << " e " << f2 << " è "
        << Min(f1, f2) << endl;

    char c1 = 'H', c2 = 'W';
    cout << "Il maggiore fra " << c1 << " e " << c2 << " è "
        << Max(c1, c2) << endl;

    return 0;
}
```

#### Esecuzione

```
Il maggiore fra 39 e 20 è 39
Il minore fra 13.5 e 20.7 è 13.5
Il maggiore fra H e W è W
```

SOLUZIONI

In questo capitolo abbiamo visto cosa sono e come si definiscono le funzioni, che costituiscono la base della programmazione modulare. Si utilizzano funzioni per suddividere problemi grandi in moduli più piccoli per facilitarne la progettazione e la manutenzione. In particolare abbiamo appreso come si dichiara, come si definisce e come si manda in esecuzione una funzione. Abbiamo visto come la funzione restituisce un risultato al programma chiamante e come accetta da esso argomenti su cui lavorare. Abbiamo visto che gli argomenti vengono passati per valore, ma anche che il C++ ha recuperato la tecnica dei riferimenti, consentendo quindi di passare argomenti a funzioni anche tramite riferimenti. Inoltre abbiamo approfondito:

- il concetto e l'uso di prototipi, obbligatori in C++;
- il concetto di argomenti per default e come utilizzarli nelle funzioni;
- il vantaggio di utilizzare funzioni **inline** per aumentare la velocità di esecuzione.

Abbiamo intravisto che la libreria standard C++ include una grande quantità di funzioni.

Altri temi approfonditi sono stati:

- la *visibilità* o le *regole di visibilità* di funzioni e variabili;
- l'*ambiente* di un programma ha quattro tipi di visibilità: di programma, file sorgente, funzione e blocco;
- le *variabili globali* e quelle *locali*;
- le *variabili statiche* che mantengono la loro informazione dopo che la funzione ha finito di lavorare;
- le *variabili registro* che si utilizzano quando si vuole aumentare la velocità di processamento di certe variabili;
- le *funzioni ricorsive* sono quelle che chiamano sé stesse;
- il *sovraffunzionamento delle funzioni* che permette di utilizzare lo stesso nome per funzioni diverse;
- i *template di funzioni* ovvero schemi di funzione che possono supportare differenti tipi di dato.

Soluzioni degli esercizi sul sito web [www.mheducation.it](http://www.mheducation.it)

SOLUZIONI

- Visibilità
- Argomento
- Libreria di funzioni
- Compilazione separata
- Funzione
- Parametro referenza
- Parametro valore

- Template di funzioni
- Prototipo
- Ricorsività
- Sovraccarico di funzioni
- Visibilità

## ESERCIZI

Scrivere una funzione logica *Cifra* che determini se un carattere è una delle cifre da 0 a 9.

Scrivere una funzione *Arrotondamento* che accetti un valore reale *Quantità*

e un valore intero *Decimali* e restituisca il valore *Quantità* arrotondato al numero specificato di decimali. Per esempio, Arrotondamento (20,563, 2) restituisce il valore 20,56, e Arrotondamento (20,563, 1) restituisce 20,5.

**Esercizio 1** Scrivere una funzione ricorsiva che calcoli i primi  $N$  numeri primi.

**Esercizio 2** Scrivere una funzione che abbia un argomento di tipo intero e che restituisca la lettera  $P$  se il numero è positivo, e la lettera  $N$  se è zero o negativo.

**Esercizio 3** Scrivere una funzione logica di due argomenti interi, che restituisca `true` se uno divide l'altro e `false` in caso contrario.

**Esercizio 4** Scrivere una funzione che converta una temperatura data in gradi Celsius a gradi Fahrenheit. La formula di conversione è:

$$F = 9/5 \cdot C + 32$$

**Esercizio 5** Scrivere una funzione logica `Vocale` che determini se un carattere è una vocale.

**Esercizio 6** Scrivere una funzione che riceva una stringa di caratteri e la restituisca in ordine inverso ('ciao' si converte in 'oain').

**Esercizio 7** Scrivere una funzione che determini se una stringa di caratteri è palindroma (cioè si legge allo stesso modo in entrambi i sensi; per esempio, radar).

**Esercizio 8** Scrivere una funzione che calcoli quanti punti di coordinate intere esistono dentro un triangolo di cui si conoscono le coordinate dei suoi tre vertici.

**Esercizio 9** Scrivere una funzione che abbia come parametro due numeri interi positivi `num1` e `num2`, e calcoli il resto della divisione intera del maggiore di loro tra il minore mediante somme e sottrazioni.

**Esercizio 10** Scrivere una funzione che abbia come parametri due numeri interi positivi `num1` e `num2` e calcoli il quoziente della divisione intera del primo tra il secondo mediante somme e sottrazioni.

**Esercizio 11** Un numero intero  $n$  si dice perfetto se la somma dei suoi divisori includendo 1 ed escludendo sé stesso coincide con sé stesso. Scrivere una funzione che decida se un numero è perfetto. Per esempio, 6 è un numero perfetto  $1 + 2 + 3 = 6$ .

**Esercizio 12** Scrivere una funzione che decida se due numeri interi positivi sono amici. Due numeri sono amici se la somma dei divisori diversi da sé stesso di ognuno di loro coincide con l'altro numero. Per esempio, 220 e 284 sono due numeri amici.

**Esercizio 13** Dato il valore di un angolo, scrivere una funzione che mostra i valori di tutte le funzioni trigonometriche.

**Esercizio 14** Scrivere una funzione che decida se un numero intero positivo è primo.

**Esercizio 15** Scrivere funzioni che calcolino il massimo comune divisore e il minimo comune multiplo di due numeri interi.

**Esercizio 16** Scrivere funzioni per leggere frazioni e visualizzarle, rappresentando le frazioni con due numeri interi numeratore e denominatore.

**Esercizio 17** Scrivere un programma che tramite un menù permetta di gestire le operazioni somma, sottrazione, prodotto e quoziente di frazioni.

**Esercizio 18** La funzione seno viene definita tramite il seguente sviluppo in serie.

$$\sin(x) = \sum_{i=0}^n \frac{(-1)^i \cdot x^{2i+1}}{(2i+1)!}$$

Scrivere una funzione che riceva come parametri il valore di  $x$  e un limite di errore, e poi calcoli il seno di  $x$  con un errore minore del limite che le si passi. Esegua la funzione in un programma con diversi valori di collaudo.

**Esercizio 19** La funzione coseno viene definita tramite il seguente sviluppo in serie di Taylor.

$$\cos(x) = \sum_{i=0}^n \frac{(-1)^i \cdot x^{2i}}{(2i)!}$$

Scrivere una funzione che riceva come parametro il valore di  $x$  e un limite di errore, e calcoli il coseno di  $x$  con un errore minore del limite che le si passi.

# Vettori e strutture

6

- |            |  |             |   |
|------------|--|-------------|---|
| <b>6.1</b> | Array                                  | <b>6.7</b>  | Accesso ai singoli campi delle strutture  |
| <b>6.2</b> | Inizializzazione di un array           | <b>6.8</b>  | Strutture annidate                        |
| <b>6.3</b> | Array di caratteri e stringhe di testo | <b>6.9</b>  | Array di strutture                        |
| <b>6.4</b> | Array multidimensionali                | <b>6.10</b> | Utilizzazione di strutture come parametri |
| <b>6.5</b> | Passaggio di vettori come parametri    | <b>6.11</b> | Funzioni membri di strutture              |
| <b>6.6</b> | Strutture                              | <b>6.12</b> | Unioni                                    |

Indice

## Introduzione

Finora sono state descritte le caratteristiche dei tipi di dato fondamentali (carattere, numero intero e in virgola mobile). Abbiamo anche imparato a definire e utilizzare costanti simboliche utilizzando `const`, `#define` e il tipo `enum`. In questo capitolo continueremo l'esame dei tipi di dato del C++ introducendo due strutture dati fondamentali nella programmazione strutturata: gli *array* (vettori) e le *struct* (strutture, records).

L'*array* è una struttura che contiene dati dello stesso tipo, come interi, numeri in virgola mobile o caratteri. È molto importante anche per il fatto che viene utilizzato per rappresentare le *stringhe*, cioè le sequenze di caratteri.

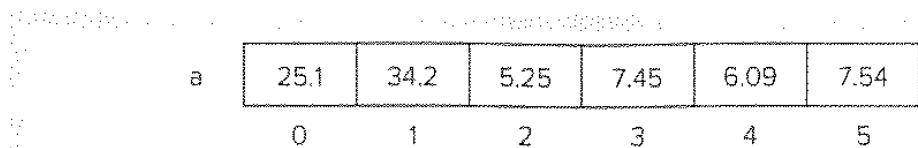
Gli elementi di un array possono essere a loro volta degli array, e in questo caso si rappresentano *matrici bidimensionali*, o anche array di array di array (matrici tridimensionali) e così via.

Gli elementi di un array possono essere di tipo *struct*, e in questo caso quel vettore diventa una *tabella*, cioè la componente fondamentale delle cosiddette "basi di dati".

A differenza del vettore, una *struttura* può aggregare dati di tipo diverso. La struttura importante per la creazione di basi di dati o altre applicazioni che richiedano grandi quantità di dati. Come vedremo in seguito, essa è anche alla base del concetto di *oggetto*.

## 6.1 Array

Un *array* (o *vettore*) è una sequenza di oggetti dello stesso tipo. Gli oggetti si chiamano elementi dell'*array* e si numerano consecutivamente 0, 1, 2, 3... Il

**Figura 6.1***Array di sei elementi.*

tipo degli elementi dell'*array* può essere qualsiasi predefinito (come *char*, *int* o *float*) ma anche tipi di dato definiti dall'utente, come vedremo più avanti. Un *array* può contenere, per esempio, l'età degli allievi di una classe, le temperature di ogni giorno del mese, o il numero dei tamponi positivi per ogni regione in occasione di pandemie.

Come detto prima, gli *elementi* di un array si numerano consecutivamente a partire dallo 0. Questi numeri si dicono *indici* dell'array, e il loro ruolo è quello di localizzare la posizione di ogni elemento dentro l'array, fornendo *accesso diretto* a esso.

Se il nome del vettore è *a*, allora *a[0]* è il nome del primo elemento, *a[1]* è il nome del secondo elemento ecc. L'elemento *i*-esimo si trova quindi nella posizione *i*-1, e se l'array ha *n* elementi, i loro nomi sono *a[0]*, *a[1]*, ..., *a[n-1]*.

La Figura 6.1 rappresenta graficamente un array *a* con sei numeri reali: *a[0]* contiene 25.1, *a[1]* contiene 34.2, *a[2]* contiene 5.25, *a[3]* contiene 7.45, *a[4]* contiene 6.09 e *a[5]* contiene 7.54. Questa figura rappresenta realmente una regione della memoria del computer perché un *array* si immagazzina sempre sequenzialmente su posizioni di memoria contigue.

### 6.1.1 Definizione di un array

Come qualunque altro tipo di variabile, l'array deve essere dichiarato prima di essere utilizzato. Un array si definisce in modo simile agli altri tipi di dato, ma si deve indicare tra parentesi quadre la sua dimensione (o *lunghezza*). La *sintassi* per dichiarare un array di una dimensione determinata è:

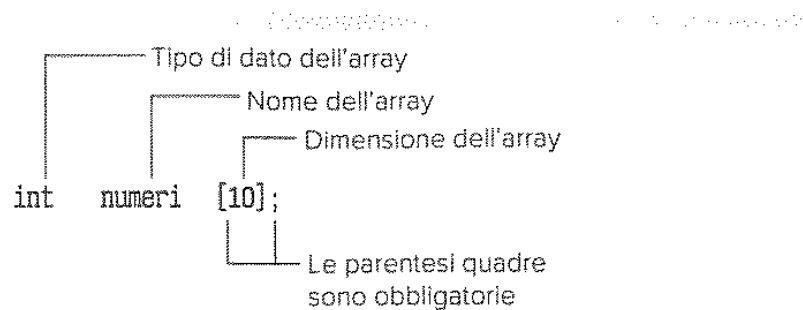
*tipo nomeArray[numeroDiElementi];*

*numeroDiElementi* dimensione dell'array, deve essere un valore intero (come 10), un valore const, o un'espressione costante (come 5 \* sizeof(int)) o una variabile che però sia stata istanziata prima della definizione dell'array.

Per esempio, per creare un array (lista) di dieci variabili intere, si scrive (Figura 6.2):

```
int numeri[10]; //Crea un array di 10 elementi int
```

Questa definizione fa sì che il compilatore riservi spazio sufficiente per contenere dieci numeri interi. In C++ gli interi occupano normalmente 4 byte, così che un array di dieci interi occupa 40 byte di memoria.



**Figura 6.2.**  
Sintassi della definizione di un array.

### 6.1.2 Accesso agli elementi di un array

L'utilità dell'array sta nel fatto che si può accedere a ogni suo elemento mediante uno stesso nome e un *indice* scorrevole:

*nomeArray[n]*

posizione dell'elemento dentro l'array

Come già detto, il primo elemento ha indice 0, il successivo 1 e così via. Per esempio, la seguente istruzione:

`cout << numeri[4] << endl;`

visualizza a schermo il valore del quinto elemento dell'array *numeri*.

#### ATTENZIONE

C++ non verifica che gli indici dell'array stiano dentro la dimensione definita. Così, per esempio, si può tentare di accedere a *numeri[12]* e il compilatore non segnalerà alcun errore.

L'indicizzazione partendo da 0 ha come effetto che l'indice di un elemento dell'array indica il numero di «passi» da fare per raggiungerlo partendo da quello iniziale. Per esempio, *a[3]* sta a 3 passi dall'elemento *a[0]*. Il meccanismo intrinseco si capirà meglio quando tratteremo le relazioni tra array e puntatori (Capitolo 9).

#### Esempio 6.1

`int eta[5];`

Array *eta* contiene 5 elementi: il primo, *eta[0]* e l'ultimo, *eta[4]*.

`int pesi[25], lunghezze[100];` Definisce 2 array di interi.

`float salari[25];` Definisce un array di 25 elementi float.

`double temperature[50];` Definisce un array di 50 elementi double.

`char lettere[15];` Definisce un array di caratteri.

Si possono referenziare elementi anche utilizzando espressioni negli indici:

`vendite[totale + 5]`

`salario[eta[i] * 5]`

### Esempio 6.2

*Scrivere un programma che chieda in input le età degli undici calciatori e le visualizzi a schermo.*

```
int main()
{
    unsigned eta[10]; //vettore di 11 numeri naturali
    for (int i = 0; i < 11; i++)
    {
        cout << "Introduca età del calciatore: ";
        cin >> eta[i];
    }
    for (int i = 0; i < 11; i++)
        cout << "le età dei calciatori sono " << eta[i] << endl;
    return 0;
}
```

### 6.1.3 Allocazione in memoria degli array

Gli elementi degli array si allocano in blocchi contigui. Così, per esempio, gli array

```
int eta[5];
int codici[5];
```

possiamo pensarli allocati in memoria come rappresentato nella Figura 6.3, dove ogni elemento occupa 4 byte.

Gli array di caratteri funzionano alla stessa maniera, ma ogni carattere occupa solo un byte. Così, per esempio, un array chiamato nome si può rappresentare come in Figura 6.4.

### 6.1.4 Dimensione degli array (sizeof)

La funzione `sizeof()` restituisce il numero di byte necessari per contenere il dato o il tipo di dato passatole come argomento. Se la si usa per conoscere la

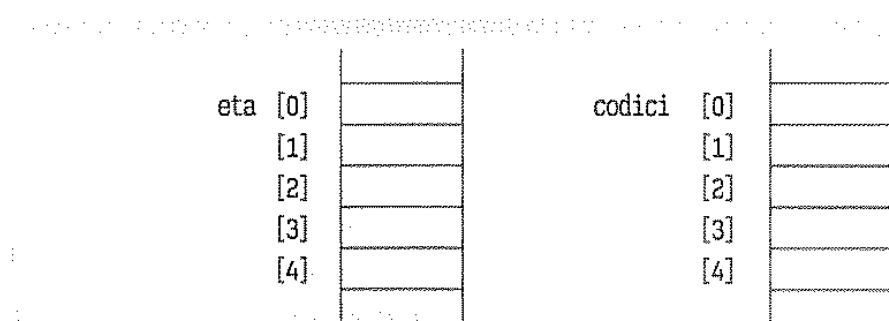


Figura 6.3

Allocazione in memoria di array.

		nome
char	nome[]	= "Mortimer"
char	nome[10]	= "Mackoy"
		M [0]
		o [1]
		r [2]
		t [3]
		i [4]
		m [5]
		e [6]
		r [7]

**Figura 6.4**

Un array di caratteri in memoria.

dimensione di un array, essa restituisce il numero di byte occupati da tutto l'array (ovvero la dimensione del tipo di dato moltiplicata per il numero di elementi dell'array). Per esempio, supponiamo che si definisca un array di 100 numeri interi denominato eta; se si vuole conoscere la dimensione dell'array, si può utilizzare un'istruzione simile a:

```
n = sizeof(eta);
```

dove *n* prenderà il valore 400. Se si vuole conoscere la dimensione di un elemento individuale dell'array si può scrivere:

```
n = sizeof(eta[6]);
```

*n* prenderà il valore 4 (numero di byte che contengono un intero).

### 6.1.5 Verifica dell'indice di un array

A differenza di altri linguaggi di programmazione, C/C++ non verifica che il valore dell'indice di un array si mantenga inferiore alla sua lunghezza e non coda nel famigerato errore di *"buffer overflow"*. Questo significa che è responsabilità del programmatore inserire controlli sul valore dell'indice per evitare che esso sfiori la dimensione del vettore.

#### Esempio 6.3

Esempio di come proteggere una funzione da errori di *buffer overflow*. La funzione calcola la somma dei valori degli elementi di un vettore (passatole come argomento) verificando che l'indice non superi la sua dimensione.

```
double somma(double a[], int n)
{
    if (n * sizeof(double) > sizeof(a)) return 0;
    double S = 0.0;
```

```

    for (int i = 0; i < n; i++) S += a[i];
    return S;
}

```

## 6.2 Inizializzazione di un array

Si può ovviamente *inizializzare* un array un elemento alla volta:

```

prezzi[0] = 10;
prezzi[1] = 20;
prezzi[3] = 30;
prezzi[4] = 40;
...

```

ma questo metodo non è pratico. Il metodo normalmente utilizzato è quello di inizializzarlo in una sola istruzione all'atto della sua definizione. Il metodo consiste nell'assegnargli una lista di valori separati da virgole, in corrispondenza di posizione e racchiusi tra parentesi graffe (Figura 6.5).

Quando si inizializzano gli array in questo modo la dimensione è opzionale dato che il compilatore conterà il numero di valori dentro le parentesi graffe e ne desumerà la dimensione, come nel seguente esempio.

### Esempio 6.4

Esempi di come s'inizializza un array.

```

int main ()
{
    char caratteri[] = ['A', 'l', 'd', 'o']; // vettore caratteri
    char stringa1[] = {"Aldo"}; // stringa di caratteri
    char stringa2[] = "Aldo"; // altra inizializzazione di stringa
    int numeri[6] = {10, 20, 30, 40, 50, 60};
    int n[] = {3, 4, 6}; // dimensione 3 riconosciuta automaticamente
}

```

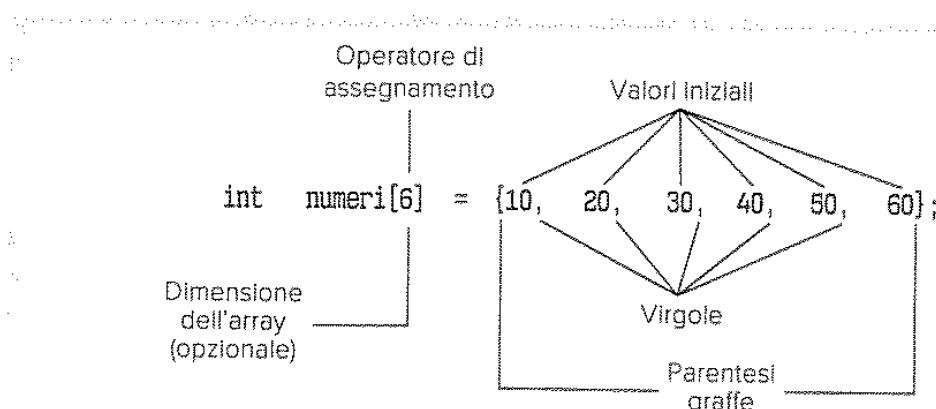


Figura 6.5

Sintassi di inizializzazione di un array.

```

const int G=31,F=28,M=31,A=30,M=31,G=30,L=31,A=31,S=30,O=31,N=30,D=31;
int mesi[12]={G, F, M, A, M, G, L, A, S, O, N, D};
cout << "dimensione vettore caratteri = " << sizeof(caratteri) << endl;
cout << "dimensione vettore stringa1 = " << sizeof(stringa1) << endl;
cout << "dimensione vettore stringa2 = " << sizeof(stringa2) << endl;
cout << "dimensione vettore numeri = " << sizeof(numeri) << endl;
cout << "dimensione vettore n = " << sizeof(n) << endl;
cout << "dimensione vettore mesi = " << sizeof(mesi) << endl;
}

```

### Esecuzione

```

dimensione vettore caratteri = 4
dimensione vettore stringa1 = 5
dimensione vettore stringa2 = 5
dimensione vettore numeri = 24
dimensione vettore n = 12
dimensione vettore mesi = 48

```

I primi due esempi mostrano la differenza fra un semplice *vettore di caratteri* (il primo) e una *stringa di caratteri* (il secondo), la quale è un vettore di caratteri con un carattere conclusivo in più, cioè il carattere ASCII NULL (quello con otto bit tutti a 0). Si vede che la stringa ha un elemento in più del semplice vettore di caratteri. Il terzo esempio mostra come meglio inizializzare una stringa. L'ultimo esempio mostra come si possano anche assegnare costanti simboliche come valori numerici agli elementi di un array.

Alternativamente, per inizializzare l'array si può usare un'istruzione ciclica (tipicamente il for come nell'Esempio 6.2).

Se non si inizializza un vettore, i suoi elementi assumono tutti valore 0 se il vettore è definito come variabile globale, assumono invece valori indeterminati se il vettore è definito come variabile locale.

### Esempio 6.5

```

int globaleint[10];
char globalecar[10];
int main()
{
    cout << "[ ";
    for (int j = 0; j <= 9; j++) cout << globaleint[j] << ' ';
    cout << "] \n";
    int localeint[10];
    cout << "[ ";
    for (int j = 0; j <= 9; j++) cout << localeint[j] << ' ';
    cout << "] \n";
    cout << "[ ";
    for (int j = 0; j <= 9; j++) cout << globalecar[j] << ' ';
    cout << "] \n";
}

```

```

char localecar[10];
cout << "[ ";
for (int j = 0; j <= 9; j++) cout << localecar[j] << ' ';
cout << "]\n";
}

```

### Esecuzione di prova

```

[ 0 0 0 0 0 0 0 0 0 ]
[ 2 0 1813677165 22011 946974664 32598 1813677088 22011 0 0 ]
[           ]
[   0 j G   ]

```

Si vede dall'esempio che i due vettori definiti come globali hanno tutti gli elementi inizializzati a 0 (il carattere 0 corrisponde all'ASCII NULL quindi non viene visualizzato nulla nel vettore di caratteri), mentre i vettori definiti come locali hanno valori indeterminati (nel caso del vettore di caratteri nell'esecuzione di prova solo due elementi sono stati inizializzati, casualmente, a valori stampabili).

### 6.3 Array di caratteri e stringhe di testo

Una stringa di testo è una sequenza di caratteri *terminata da un carattere nullo*. Le stringhe vengono ospitate su vettori, a partire dal primo elemento fino al "carattere nullo" (\0 - ASCII 0) che non deve essere necessariamente l'ultimo elemento del vettore. La Figura 6.6 mostra una stringa e un array di caratteri.

Quando s'inizializza un array con una stringa:

```
char Stringa[7] = "AbC3zG";
```

Visualizzazione della memoria (Windows)

Stringa1[0]	M	Stringa2[0]	M
[1]	o	[1]	o
[2]	r	[2]	r
[3]	t	[3]	t
[4]	i	[4]	i
[5]	m	[5]	m
[6]	e	[6]	e
[7]	r	[7]	r
[8]		[8]	\0

(a)
(b)

carattere nullo

Figura 6.6

(a) Array di caratteri; (b) stringa.

Il compilatore inserisce automaticamente il carattere nullo alla fine della stringa, in modo che la sequenza di caratteri allocata in memoria è:

Stringa	A	b	C	S	z	G	\0
---------	---	---	---	---	---	---	----

Tuttavia non si può assegnare una stringa a un array al di fuori della definizione:

```
Stringa = "AbC3zG"; \\ errore
```

Per assegnare una stringa a una variabile si può utilizzare la funzione di libreria standard `strcpy()` della libreria `<cstring>`. Per esempio, per assegnare la stringa "Abracadabra" all'array di caratteri `stringa`, si può scrivere:

```
strcpy(stringa, "Abracadabra"); // alloca Abracadabra in stringa
```

Anche `strcpy` aggiunge il carattere nullo alla fine della stringa. Si deve però essere sicuri che l'array di caratteri `stringa` abbia sufficienti elementi per contenere la stringa, altrimenti avremo un *buffer overflow* e si potrà andare a scrivere in celle di memoria esterne all'array.

## 6.4 Array multidimensionali

Gli array visti finora sono *unidimensionali* e hanno un solo indice. Gli array di array si dicono *bidimensionali*; hanno due dimensioni e, pertanto, due indici. Gli array di array di array sono tridimensionali e così via. Gli array bidimensionali sono noti anche con il nome di *tabelle* o *matrici* (Figura 6.7).

Ogni elemento sarà localizzato tramite le coordinate rappresentate dal suo numero di riga e il suo numero di colonna. La sintassi per la dichiarazione di un array di due dimensioni è:

```
<tipo di dato Elemento> <nome array> [<NumRighe>] [<NumColonne>]
```

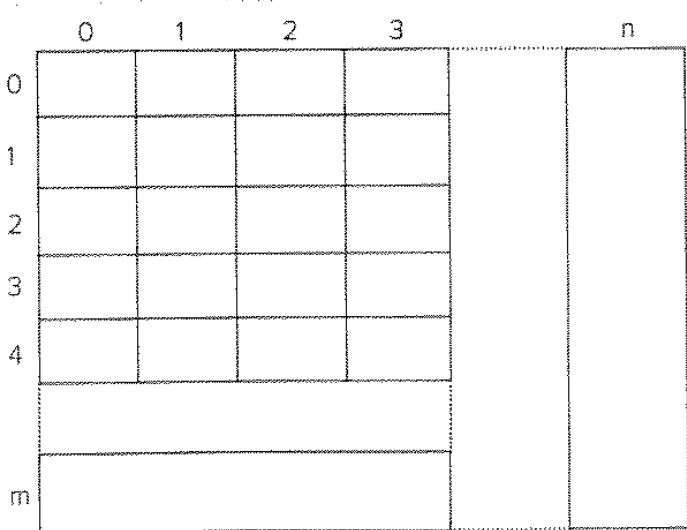


Figura 6.7

Struttura di un array di due dimensioni.

**Tabella 6.1** Un array bidimensionale

Elemento	Posizione relativa di memoria
tabella[0][0]	0
tabella[0][1]	4
tabella[1][0]	8
tabella[1][1]	12
tabella[2][0]	16
tabella[2][1]	20
tabella[3][0]	24
tabella[3][1]	28

per esempio:

```
char schermata[25][80];
int classifica[20][8];
```

Gli elementi degli array si allocano in memoria in modo che il primo indice rappresenta la riga e l'altro la colonna. La Tabella 6.1 rappresenta elementi e posizioni relative in memoria dell'array di interi:

```
int tabella[4][2];
```

#### 6.4.1 Inizializzazione di array multidimensionali

Anche gli array multidimensionali si possono inizializzare nella definizione. Per esempio, nel caso del primo array della Figura 6.8 l'inizializzazione consta di una sequenza di costanti separate da virgole e racchiuse tra parentesi graffe:

```
int tabella[2][3] = {51, 52, 53, 54, 55, 56};
```

oppure:

```
int tabella[2][3] = {[51, 52, 53], [54, 55, 56]};
```

tabella[2][3] 0 1 2 Colonne

Righe	0	1	2
0	51	52	53
1	54	55	56

tabella[3][4] 0 1 2 3 Colonne

Righe	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

**Figura 6.8**

Tabelle di due dimensioni.

### 6.4.2 Accesso agli elementi di array bidimensionali

Per accedere agli elementi di array bidimensionali si debbono specificare l'indice di riga e quello di colonna. Il formato generale per l'assegnamento di valori agli elementi è:

<nome array>[indice riga][indice colonna]=valore elemento;  
per leggere gli elementi:  
    <variabile> = <nome array> [indice riga] [indice colonna];

Alcuni esempi di inserzioni possono essere:

```
Tabella[2][3] = 4.5;
cin >> Tabella[2][3];
```

esempi di estrazione:

```
Vendite = Tabella[1][1];
cout << Tabella[1][1];
```

### 6.4.3 Accesso a elementi tramite cicli

Si può accedere agli elementi di array bidimensionali tramite cicli annidati:

```
for (int IndiceRiga = 0; IndiceRiga < NumRiga; ++IndiceRiga)
    for (int IndiceCol = 0; IndiceCol < NumCol; ++IndiceCol)
        processare elemento[IndiceRiga][IndiceCol]
```

#### Esempio 6.6

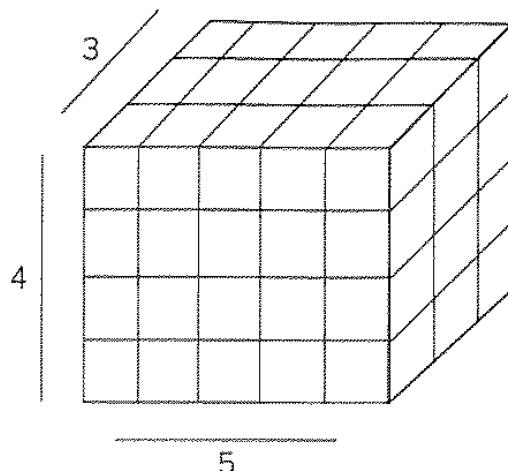
```
float dischi[2][4]; // definisce la tabella
// inserisce dati dall'input in una tabella
for (int riga = 0; riga < 2; riga++)
    for (int colonna = 0; colonna < 4; colonna++)
        cin >> dischi[riga][colonna];
// inserisce dati dalla tabella nel flusso di output
for (int riga = 0; riga < 2; riga++)
    for (int colonna = 0; colonna < 4; colonna++)
        cout << "Disco: " << dischi[riga][colonna] << "\n";
```

### 6.4.4 Array di più di due dimensioni

C++ dà la possibilità di inserire dati in array pluridimensionali, anche se raramente si richiede più di due o tre dimensioni. Il modo più facile per rappresentare un array di tre dimensioni è di immaginare un parallelepipedo (Figura 6.9).

Un esempio di definizione di array tridimensionale è:

```
const int PAGINE=500, RIGHE=45, COLONNE=80;
char libro[PAGINE][RIGHE][COLONNE];
```



**Figura 6.9** Un array di tre dimensioni ( $4 \times 5 \times 3$ ).

Ovviamente anche qui per accedere agli elementi dell'array, sia in lettura sia in scrittura, si utilizzeranno cicli annidati:

```
for (int pagina = 0; pagina < PAGINE; ++pagina)
    for (int riga = 0; riga < RIGHE; ++riga)
        for (int colonna = 0; colonna < COLONNE; ++colonna)
            processare libro[pagina][riga][colonna]>
```

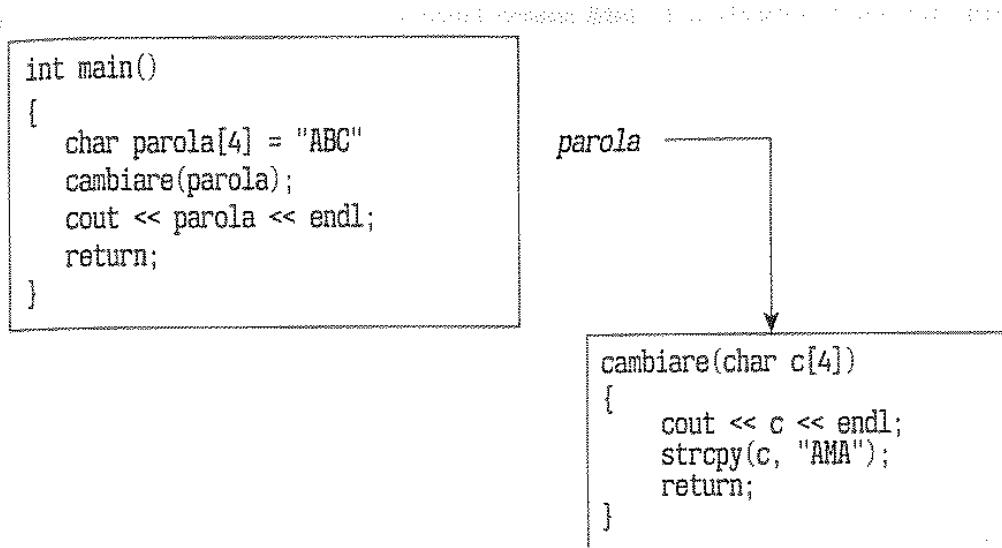
## 6.5 Passaggio di vettori come parametri

*Gli array si passano per riferimento.* Ciò significa che quando si passa un array come parametro a una funzione, C++ tratta la chiamata come se vi fosse l'operatore di indirizzo & davanti al nome del vettore. La Figura 6.10 aiuta a capire il meccanismo.

Supponiamo di voler definire una funzione per calcolare la somma degli elementi di un vettore di double. La funzione avrà bisogno di due parametri, uno per accettare in input il vettore (passato per riferimento) e uno per accettare in input la sua dimensione.

```
double SommaVettore(double vettore[], int n)
{
    double Sommatoria = 0;
    while (n > 0) Sommatoria += vettore[n];
    return Sommatoria;
}
```

Si lasciano le parentesi quadre in bianco perché ciò che viene in realtà passato non è il vettore ma solo il suo riferimento, ovvero il suo nome (sarà più chiaro quando vedremo la relazione fra vettori e puntatori). Il secondo parametro è fondamentale perché bisogna passare alla funzione anche la dimensione del vettore; la funzione non può conoscere quanti elementi sono



**Figura 6.10** Passaggio di un array a una funzione. Il vettore viene passato per valore.

Passaggio di un array a una funzione.  
Il vettore viene passato per valore. La funzione non può modificare l'array originale.

nell'array, essa sa dove inizia l'array ma non sa dove termina. Supponendo di aver definito un vettore:

`double voti[100];`

questa chiamata visualizzerebbe a schermo la somma degli elementi dell'array:

`cout << "Somma dei voti è: " << SommaVettore(voti, 100);`

mentre questa visualizzerebbe la somma dei primi 80:

`cout << "Somma dei primi 80 voti è: " << SommaVettore(dati, 80);`

Questa infine sarebbe un *buffer overflow*:

`cout << "Somma dei voti è: " << SommaVettore(dati, 200);`

### Esempio 6.7

Passaggio di un vettore a funzioni.

```

#define MAXLUNG 100
void leggeVettore(double[], int&);
void stampaVettore (double [], int);

int main()
{
    double a[MAXLUNG];
    int n;
    leggeVettore(a, n);
    cout << "L'array ha " << n << " elementi\nSono:\n";
    stampaVettore(a, n);
}

```

```

void leggeVettore(double a[], int& numero_immessi)
{
    cout << "Per favore, immetta dei numeri non nulli; per terminare immetta 0.\n";
    for (numero_immessi = 0; numero_immessi < MAXLUNG; numero_immessi++)
    {
        cout << numero_immessi << ":\t";
        cin >> a[numero_immessi];
        if (a[numero_immessi] == 0) break;
    }
}

void stampaVettore(double a[], int n)
{
    for (int i = 0; i < n; i++)
        cout << i << ":\t" << a[i] << endl;
}

```

### Esecuzione di prova

Per favore, immetta dei numeri non nulli; per terminare immetta 0.

```

0: 11.11
1: 22.22
2: 33.33
3: 0

```

L'array ha 3 elementi

Sono:

```

0: 11.11
1: 22.22
2: 33.33

```

Il secondo parametro della funzione `void leggeVettore(double[], int&)` è un riferimento a intero. Questo perché la `main()` le passa la sua variabile `n` affinché la funzione la possa valorizzare correttamente in base al numero degli elementi immessi dall'utente. Il vettore ha dimensione `MAXLUNG` ma contiene solo `n` numeri, cioè quelli immessi fino a quando l'utente inserisce 0. Comunque sia, il `for` d'immissione termina dopo `MAXLUNG` iterazioni, quindi non si verifica il *buffer overflow*.

### Esempio 6.8

Passaggio di una matrice a funzioni.

```

#define MAXRIGHE 10
#define MAXCOLONNE 10
void leggeMatrice(double[][] [MAXCOLONNE], int&, int&);
void stampaMatrice (double [][] [MAXCOLONNE], int, int);

int main()
{
    double matrice[MAXRIGHE][MAXCOLONNE];
    int righe, colonne;
    leggeMatrice(matrice, righe, colonne);
}

```

```

    stampaMatrice (matrice, righe, colonne);
}

void leggeMatrice(double a[][MAXCOLONNE], int& r, int& c)
{
    cout << "Quante sono le righe? ";
    cin >> r;
    cout << "Quante sono le colonne? ";
    cin >> c;
    if (c > MAXCOLONNE) {
        cout << "Purtroppo le colonne sono troppe!";
        return;
    }
    for (int i = 0; i < r; i++)
    {
        cout << "Immetta " << c << " numeri: ";
        for (int j = 0; j < c; j++) cin >> a[i][j];
    }
}
}

void stampaMatrice(double a[][MAXCOLONNE], int r, int c)
{
    cout << "La matrice immessa è:\n";
    cout << "      ";
    for (int i = 0; i < c; i++) cout << "C" << i << "  ";
    cout << endl;
    for (int i = 0; i < r; i++)
    {
        cout << "R" << i << ": ";
        for (int j = 0; j < c; j++) cout << a[i][j] << ' ';
        cout << endl;
    }
}

```

### Esecuzione di prova

```

Quante sono le righe? 3
Quante sono le colonne? 3
Immetta 3 numeri: 11.11 22.22 33.33
Immetta 3 numeri: 44.44 55.55 66.66
Immetta 3 numeri: 77.77 88.88 99.99
La matrice immessa è:
      C0   C1   C2
R0: 11.11 22.22 33.33
R1: 44.44 55.55 66.66
R2: 77.77 88.88 99.99

```

Importante! La funzione che accetta una matrice in input deve specificare il numero delle colonne perché questo è necessario a dimensionare la lunghezza della riga (che si ottiene moltiplicando la dimensione del tipo degli elementi

per il numero delle colonne). Questo vale per tutti i vettori multidimensionali; la funzione che li accetta in input deve definire il parametro formale lasciando non specificata solo la prima dimensione e specificando tutte le altre. Quando tratteremo i puntatori torneremo su questo punto per spiegare meglio.

## 6.6 Strutture

Gli array sono strutture dati tutti dello stesso tipo, e questo è un limite quando bisogna descrivere oggetti caratterizzati da attributi di tipo diverso. Per esempio, per descrivere un oggetto "cliente" bisogna raccogliere dati di tipo diverso, come il nome, l'età, l'indirizzo, il numero del conto corrente ecc. La soluzione a questo problema è di utilizzare un tipo di dato *struttura*.

I componenti individuali di una struttura si chiamano *campi*, ognuno caratterizzato da un *nome* e capace di contenere dati di tipo diverso.

Una **struttura** è una collezione di elementi denominati **campi**, ognuno dei quali può contenere un dato di tipo diverso.

Una struttura può contenere qualunque numero di campi. Supponiamo di voler immagazzinare i dati di una collezione di compact disc. I campi della struttura CD potrebbero essere cinque:

- titolo;
- artista;
- numero di canzoni;
- prezzo;
- data di acquisto.

Dopodiché bisogna capire di che tipo sarà ciascun campo:

Nome campo	Tipo di dato
Titolo	Array di caratteri di dimensione 30
Artista	Array di caratteri di dimensione 25
Numero di canzoni	Intero
Prezzo	Virgola mobile
Data di acquisto	Array di caratteri di dimensione 8

La Figura 6.11 contiene la struttura CD, mostrando graficamente i tipi di dato dentro la struttura. Si osservi che ogni campo è un tipo di dato differente.

Titolo	Hebron Gate
Artista	Groundation
Numero di canzoni	9
Prezzo	11.99
Data di acquisto	8-10-07

Figura 6.11

Rappresentazione grafica di una struttura CD.

### 6.6.1 Dichiarazione di una struttura

Una struttura è un tipo di dato definito dal programmatore. Prima di definire una variabile di quel tipo bisogna definire il tipo stesso! Il formato della dichiarazione è:

```
struct <nome della struttura>
{
    <tipo di dato campo1> <nome campo1>;
    <tipo di dato campo2> <nome campo2>;
    ...
    <tipo di dato campoN> <nome campoN>;
};
```

La dichiarazione della struttura CD è

```
struct CD
{
    char titolo[30];
    char artista[25];
    int num_canzoni;
    float prezzo;
    char data_acquisto[8];
}
```

### 6.6.2 Definizione di variabili di struttura

Dopo aver definito il tipo di dato, si può definire variabili di quel tipo. Le variabili di tipo struct si possono definire in due modi:

1. elencandole dopo la parentesi graffa di chiusura della dichiarazione della struttura;
2. scrivendo il nome della struttura seguita dalle variabili di quel tipo:

#### 1. struct CD

```
{
    char titolo[30];
    char artista[25];
    int num_canzoni;
    float prezzo;
    char data_compra[8];
}
```

cd1, cd2, cd3;

#### 2. CD cd1, cd2, cd3;

Questa seconda definizione non è consentita in C, dove è obbligatorio l'uso della parola riservata struct nella definizione:

```
struct CD cd1, cd2, cd3;
```

Consideriamo un programma che archivi libri processando i seguenti dati: titolo del libro, nome dell'autore, editore e anno di pubblicazione. Un tipo libro potrebbe essere:

```
struct libro {
    char titolo[60];
    char autore[30];
    char editore[30];
    int anno;
};
```

La definizione di variabili di tipo libro si potrebbe fare così:

1. libro libro1, libro2, libro3;

oppure subito dopo la definizione della struttura:

2. struct libro
 [
 char titolo[60];
 char autore[30];
 char editoriale[30];
 int anno;
 ] libro1, libro2, libro3;

### 6.6.3 Inizializzazione di una dichiarazione di strutture

Una variabile di tipo struttura si può inizializzare in qualunque punto del programma, anche nella definizione. Per esempio:

```
struct data
{
    int mese;
    int giorno;
    int anno;
} data_di_nascita = {1,6,1982};

struct libro
{
    char titolo[60];
    char autore[30];
    char editoriale[30];
    int anno;
} libro1 = { "C++", "Luis Joyanes", "McGraw-Hill", 1994 };
```

I valori si debbono dare nell'ordine in cui sono definiti i corrispondenti campi nella struttura. Nell'esempio precedente, il campo `mese` di `data_di_nascita` prende il valore iniziale 1, il campo `giorno` prende il valore 6 e il campo `anno` prende il valore 1982. Se vi sono più valori di inizializzazione che campi della struttura il compilatore segnala un errore. Al contrario, se i valori sono di meno vengono inizializzati solo i primi campi a cui corrisponde un valore, gli altri prendono il valore di default corrispondente al loro tipo.

#### 6.6.4 Le strutture possono essere assegnate

Un altro vantaggio notevole delle strutture rispetto ai vettori è che esse possono essere assegnate. Per esempio, si può assegnare alle variabili libro4, libro5 e libro6 gli stessi valori rispettivamente di libro1, libro2 e libro3:

```
libro libro4 = libro1;
libro libro5 = libro2;
libro libro6 = libro3;
```

#### 6.6.5 Dimensione di una struttura

Il seguente programma illustra l'uso dell'operatore `sizeof` per determinare la dimensione di una struttura:

```
struct persona
{
    char nome[30];
    int eta;
    float altezza;
    float peso;
};

int main()
{
    persona tizio;
    cout << "sizeof(persona): " << sizeof(tizio);
}
```

Quando si esegue il programma si produce l'output:

```
sizeof(persona): 42
```

Il risultato si ottiene determinando il numero di byte che occupa la struttura.

Persona	Campi dato	Dimensione (byte)
nome[30]	char(1)	30
Eta	int(4)	4
Altezza	float(4)	4
Peso	float(4)	4
<i>Totale</i>		42

#### 6.7 Accesso ai singoli campi delle strutture

Per accedere in lettura o in scrittura a un singolo campo della struttura si utilizza l'operatore punto (`.`). Per esempio, per assegnare un valore a un campo di una variabile di tipo struct si scrive:

```
<variabile_struct>.<campo> = valore;
operatore punto
```

Per esempio:

```
cd1.prezzo = 30.75;
cd1.num_canzoni = 7;
```

L'operatore punto consente l'accesso diretto al campo menzionato, il cui nome diventa a tutti gli effetti quello di una variabile, quindi a esso si potranno assegnare solo valori del suo tipo.

#### Esempio 6.9

```
struct datiStudente
{
    int matricola;
    char corso[30];
};

int main()
{ datiStudente studente;
    studente.matricola = 20212021;
} studente.corso = "Fondamenti di Informatica";
```

Ovviamente si può valorizzare il campo di una variabile di tipo struct anche leggendo il valore da un flusso, per esempio dal flusso standard input cin:

```
cin >> miaMacchina.PotenzaCV;
```

Se invece si vuole mandare in output l'informazione contenuta in un campo, basta utilizzare l'operatore di inserimento << applicato al flusso dello standard output cout:

```
cout << "La potenza è di " << miaMacchina.PotenzaCV << " CV.";
```

## 6.8 Strutture annidate

Un campo di una struttura può a sua volta essere di tipo struct. Se strutture diverse possiedono parti identiche, definire queste ultime come sottostrutture fa risparmiare tempo e riduce le possibilità di errore. Consideriamo le seguenti due definizioni di strutture:

```
struct impiegato
{
    char nome[30];           //Nome dell'impiegato
    char indirizzo[25];      //Indirizzo dell'impiegato
    char citta[20];
    char provincia[20];
    long int cod_postale;
    double salario;          //Salario annuale
};
```

```
struct cliente
{
    char nome[30];      //Nome del cliente
    char indirizzo[25]; //Indirizzo del cliente
    char citta[20];
    char provincia[20];
    long int cod_postale;
    double saldo;        //Saldo
};
```

Queste strutture contengono campi identici che si potrebbero raccogliere in una struttura, info:

```
struct info
{
    char nome[30];
    char indirizzo[25];
    char citta[20];
    char provincia[20];
    long int cod_postale;
};
```

Questa struttura si può utilizzare come un campo delle altre strutture, cioè, *annidarsi*.

```
struct impiegato
{
    struct info anagrafica;
    double salario;
};

struct cliente
{
    info anagrafica; // si può non specificare struct
    double saldo;
};
```

Ogni campo della sottostruttura si raggiungerà mediante un doppio uso dell'operatore punto (.). Per esempio, supponiamo di avere definita una variabile x di tipo impiegato.

```
impiegato x;
```

Se dobbiamo assegnargli il nome leggendolo dalla tastiera scriviamo:

```
cout << "Introduca il nome dell'impiegato: ";
cin >> x.anagrafica.nome;
```

#### Esempio 6.10

Si vuole definire una struttura *impiegato* che contenga una struttura *persona* che a sua volta contenga una struttura *data\_di\_nascita*

```

struct data_di_nascita
{
    unsigned giorno;
    unsigned mese;
    unsigned anno;
};

struct persona
{
    char nome[20];
    unsigned eta;
    float altezza;
    float peso;
    data_di_nascita data;
};

struct impiegato
{
    persona pers;
    unsigned salario;
    unsigned ore_settimanali;
};

int main()
{
    impiegato x; // definisce una variabile di tipo impiegato
    cout << "introdurre il nome: "; cin >> x.pers.nome;
    cout << "introdurre l'età: "; cin >> x.pers.eta;
    cout << "introdurre l'altezza: "; cin >> x.pers.altezza;
    cout << "introdurre il peso: "; cin >> x.pers.peso;
    cout << "introdurre la data di nascita: ";
    cin >> x.pers.data.giorno;
    cin >> x.pers.data.mese;
    cin >> x.pers.data.anno;
    cout << "introduca il salario: ";
    cin >> x.salario;
    cout << "introduca numero di ore: ";
    cin >> x.ore_settimanali;
    cout << endl;
    cout << "nome: " << x.pers.nome << endl;
    cout << "età: " << x.pers.eta << endl;
    cout << "giorno di nascita: " << x.pers.data.giorno;
    return 0;
}

```

### Esecuzione

introdurre il nome: Paolo  
 introdurre l'età: 33

```

introdurre l'altezza: 178
introdurre il peso: 78
introdurre la data di nascita: 22 6 1961
introduca il salario: 1000
introduca numero di ore: 40
nome: Paolo
età: 33
giorno di nascita: 22

```

L'accesso a campi dato di strutture annidate richiede l'uso di operatori punto in cascata:

Le strutture si possono annidare a qualunque livello. È anche possibile inizializzare strutture annidate nella definizione. Il seguente esempio inizializza una variabile Luigi di tipo Persona.

```
Persona Luigi { "Luigi", 25, 1.940, 40, [ 12', 1, 70]};
```

## 6.9 Array di strutture

Si può creare un array di strutture, che sono le fondamenta delle "basi di dati" (*database*), collezioni di record da gestire in maniera efficiente mediante opportune funzioni.

Un array di strutture si definisce come qualunque array, per esempio:

```
libro libri[100];
```

riserva spazio in memoria per ospitare un vettore di 100 elementi di tipo libro. Per accedere ai campi di ognuno degli elementi struttura si utilizzano l'indice dell'array e l'operatore punto:

```

libri[0].anno = 1994;
strcpy(libri[0].titolo, "C++");
strcpy(libri[0].autore, "Luis Joyanes");
strcpy(libri[0].editore, "McGraw-Hill");

```

Si può anche inizializzare un array di strutture alla definizione, racchiudendo la lista di inizializzatori tra parentesi graffe, [ ]. Per esempio:

```

info_libro libri[4] = {
    "C++ alla sua portata", "Luis Joyanes", "McGraw-Hill", 1994,
    "The Annotated C++. Reference Manual", "Stroustrup", "Addison-Wesley", 1992,
    "The Design and Evolution of C++", "Stroustrup", "Addison-Wesley", 1994,
    "Object Orientation", "Hares, Smart", "Wiley", 1994};

```

### Array come campi

Come abbiamo già visto negli esempi, i campi delle strutture possono essere vettori.

**Esempio 6.11**

Una libreria vuole catalogare il suo inventario di libri. Con il seguente programma crea un vettore di 100 strutture, dove ogni struttura contiene diversi tipi di variabili, vettori inclusi. La funzione `getline()` sarà introdotta nel prossimo capitolo.

```
struct inventario
{
    char titolo[25];
    char data_pub[20];
    char autore[30];
    int num;
    int ordinazione;
    float prezzo_vendita;
};

int main()
{
    struct inventario libri[100];
    int id = 0;
    char risp;
    do {
        cout << "Totale libri " << id++ << "\n";
        cout << "Qual è il titolo? ";
        cin.getline(libri[id].titolo,80);
        cout << "Qual è la data di pubblicazione? ";
        cin.getline(libri[id].data_pub,80);
        cout << "Chi è l'autore? ";
        cin.getline(libri[id].autore,80);
        cout << "Quante copie esistono? ";
        cin >> libri[id].num;
        cout << "Quanti esemplari esistono ordinati? ";
        cin >> libri[id].ordinazione;
        cout << "Qual è il prezzo di vendita? ";
        cin >> libri[id].prezzo_vendita;
        cout << "\n Altri libri? (S/N)";
        cin >> risp;
    } while (risp == 'S');
    return 0;
}
```

## 6.10 Utilizzazione di strutture come parametri

C++ permette di passare strutture come parametri attuali alle funzioni, sia per valore sia per riferimento (ovvero passando non la struttura ma il suo indirizzo in memoria).

Il seguente programma passa la struttura a una funzione per valore e a un'altra per riferimento in maniera, quest'ultima, da consentirle di modificare i valori dei campi.

### Esempio 6.12

```
struct persona {
    char nome[20];
    char via[40];
    char citta[25];
    char provincia[25];
    char codicePostale[7];
};

void visualizza_info(persona individuo)
{
    cout << endl << individuo.nome << endl
        << individuo.via << endl
        << individuo.citta << " " << individuo.provincia
        << " " << individuo.codicePostale << endl << endl;
}

void cambio_residenza(persona& individuo)
{
    cout << "Qual è il nuovo indirizzo? ";
    cout << "Via: "; cin.getline(individuo.via,80);
    cout << "Città: "; cin.getline(individuo.citta,80);
    cout << "Provincia: "; cin.getline(individuo.provincia,80);
    cout << "CAP: "; cin.getline(individuo.codicePostale,80);
}

int main()
{
    persona x = {"Luigi Clifford",
                  "3 maggio", "Tresnuraghes",
                  "Oristano", "09079"};
    visualizza_info(x);
    cambio_residenza(x);
    visualizza_info(x);
    return 0;
}
```

### Esecuzione

Luigi Clifford  
 3 maggio  
 Tresnuraghes Oristano 09079

Qual è il nuovo indirizzo? Via: via Martiri della Resistenza 23  
 Città: Porto San Giorgio

Provincia: Fermo

CAP: 20123

Luigi Clifford  
via Martiri della Resistenza 23  
Porto San Giorgio Fermo 20123

### 6.11 Funzioni membri di strutture

In C++ le strutture possono anche contenere funzioni, come le classi che, vedremo in seguito, costituiscono il fulcro della "programmazione orientata agli oggetti" (*Object Oriented Programming, OOP*).

#### Esempio 6.13

Progettare una struttura *Punto* (coordinate in 3 dimensioni x, y, z) che contenga due membri funzione *sommare()* e *sottrarre()* che sommano e sottraggano, rispettivamente, due oggetti di tipo *Punto*.

```
struct Punto
{
    double x, y, z; // campi dato
    void sommare(Punto& p1, Punto& p2) // funzione membro
    {
        x = p1.x + p2.x;
        y = p1.y + p2.y;
        z = p1.z + p2.z;
    }
    void sottrarre(Punto& p1, Punto& p2) // funzione membro
    {
        x = p1.x - p2.x;
        y = p1.y - p2.y;
        z = p1.z - p2.z;
    }
};

int main()
{
    Punto p1, p2, p3;

    p1.x = 1.0; p1.y = 2.0; p1.z = 3.0;
    p2.x = 8.0; p2.y = 9.0; p2.z = 10.0;
    p3.x = 0.0; p3.y = 0.0; p3.z = 0.0;

    cout << "prima: " << endl;
    cout << "p3.x: " << p3.x << endl;
    cout << "p3.y: " << p3.y << endl;
    cout << "p3.z: " << p3.z << endl;
    p3.sommare(p1, p2); // lancia sommare di p3 passando p1 e p2

    cout << "dopo p3.sommare(p1, p2): " << endl;
```

```

cout << "p3.x: " << p3.x << endl;
cout << "p3.y: " << p3.y << endl;
cout << "p3.z: " << p3.z << endl;
p3.sottrarre(p1, p2); // lancia sottrarre di p3 passando p1 e p2

cout << "dopo p3.sottrarre(p1, p2): " << endl;
cout << "p3.x: " << p3.x << endl;
cout << "p3.y: " << p3.y << endl;
cout << "p3.z: " << p3.z << endl;
}

```

### Esecuzione

```

prima:
p3.x: 0
p3.y: 0
p3.z: 0
dopo p3.sommare(p1, p2):
p3.x: 9
p3.y: 11
p3.z: 13
dopo p3.sottrarre(p1, p2):
p3.x: -7
p3.y: -7
p3.z: -7

```

La definizione del tipo Punto include due funzioni membro sommare() e sottrarre(). A queste funzioni si passano due argomenti di tipo Punto ed esse manipolano questi due punti aggiungendo o sottraendo i loro rispettivi campi dato (x, y, z). Se una struttura si dichiara con visibilità globale, i suoi campi e le sue funzioni sono accessibili da qualunque funzione del programma. In altre parole, sono *pubblici*.

## 6.12 Unioni

Le unioni somigliano alle strutture, ma invece di allocare i campi in maniera contigua esse li sovrappongono nella stessa posizione di memoria.

La sintassi di un'unione è:

```

union nome {
    tipo1 campo1;
    tipo2 campo2;
    ...
};

```

Un esempio:

```

union ProvaUnione {
    float x;
    double y;
};

```

La quantità di memoria riservata per un'unione è data dal campo più grande, perché i campi condividono la stessa zona di memoria. Per esempio, la quantità totale di memoria utilizzata da ProvaUnione è di 8 byte, dato che l'elemento double è il campo più grande dell'unione.

Le unioni servono per risparmiare memoria se non si ha bisogno di valorizzare tutti i campi nello stesso momento. Si pensi a una situazione in cui bisogna leggere diverse stringhe dall'input, come:

```
char riga_ordini[80];
char messaggio_errore[80];
char aiuto[80];
```

Queste tre variabili occupano 240 byte di memoria. Tuttavia, se il programma non ha bisogno di utilizzarle simultaneamente, perché non permettergli di condividere la memoria utilizzando un'unione? Combinate, per esempio, nel tipo union frasi, queste variabili occupano un totale di 80 byte.

```
union frasi {
    char riga_ordini[80];
    char messaggio_errore[80];
    char aiuto[80];
} stringhe;
```

Per riferirsi ai campi di un'unione, si utilizza l'operatore punto (.).

```
stringhe.aiuto
stringhe.messaggio_errore
```

### SINTESI

Questo capitolo è stato dedicato alle due strutture dati fondamentali della programmazione strutturata, cioè il vettore o "array" e le struct (strutture, record).

Il vettore è un tipo di dato strutturato che si utilizza per raccogliere elementi dello stesso tipo, anche di tipo vettore stesso (vettori multidimensionali). Gli array si definiscono specificando il tipo di dato dell'elemento, il nome dell'array e, fra parentesi quadre, il numero degli elementi del vettore. Abbiamo visto come si accede agli elementi dell'array specificandone il numero d'indice e abbiamo discusso le caratteristiche di quei particolari vettori di caratteri che sono le stringhe (sequenze di caratteri terminanti col carattere nullo).

Il tipo `struct` permette di encapsulare dati di tipo diverso. Abbiamo visto come il programmatore deve dichiarare questo tipo di dato e come si definiscono poi variabili di quel tipo. Abbiamo visto come poi si accede ai singoli campi di una struttura e che, per default, i campi sono accessibili pubblicamente. Abbiamo anticipato (sarà tema centrale nel capitolo dedicato alle "classi") che una struttura potrebbe anche encapsulare funzioni che operano sui dati della medesima. Il tipo di dato `union` è poi simile a `struct`, ma può contenere solo uno dei suoi campi alla volta e la sua dimensione è, pertanto, la dimensione del suo campo più grande.

## CONTENUTI DI APPRENDIMENTO

- Array
- Array multidimensionali
- Array di caratteri: stringa di testo

- Struttura
- Strutture annidate

## ESERCIZI

**Esercizio 1** Scrivere un programma che inizializzi solo i primi due elementi di un vettore di 4 e visualizzi il valore degli elementi non assegnati.

**Esercizio 2** Scrivere un programma che raccolga le temperature di una città in una settimana e visualizzi a schermo la temperatura maggiore e minore raggiunte.

**Esercizio 3** Scrivere un programma che legga da tastiera una tabella di 4 righe per 5 colonne di elementi interi e dopo la lettura la visualizzi sullo schermo.

**Esercizio 4** Qual è l'output del codice seguente?

```
char a[4] = {'a', 'b', 'c', 'd'} ;
cout a[0] << " " << a[1] << " " << a[2] << endl ;
a[1] = a[2] ;
cout << a[0] << " " << a[1] << " " << a[2] << endl;
```

**Esercizio 5** Qual è l'output del seguente codice?

```
char simbolo[3] = {'a', 'b', 'c'} ;
for (int indice=0 ; indice<3 ; indice++)
cout << simbolo[indice] ;
```

**Esercizio 6** Scrivere un programma che riempia un vettore con i primi 80 numeri primi e li visualizzi.

**Esercizio 7** Scrivere un programma che legga le dimensioni di una matrice, legga e visualizzi la matrice e, poi, trovi gli elementi maggiore e minore della matrice e le loro posizioni.

**Esercizio 8** Se  $x$  rappresenta la media dei numeri  $x_1, x_2, \dots, x_n$ , allora la varianza è la media dei quadrati delle deviazioni dei numeri dalla media:

$$\text{Varianza} = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

e la deviazione standard è la radice quadrata della varianza. Scrivere un programma che legga un vettore di numeri reali, li conti e, poi, ne calcoli e stampi la media, la varianza e la deviazione standard. Utilizzare perciò funzioni.

**Esercizio 9** Scrivere un programma che legga una collezione di stringhe di caratteri di lunghezza arbitraria. Per ogni stringa letta, il programma dovrà:

- stampare la lunghezza della stringa;
- contare il numero di occorrenze di parole di quattro lettere;
- sostituire ogni parola di quattro lettere con una stringa di quattro asterischi e stampare la nuova stringa.

**Esercizio 10** Scrivere una funzione che inverta l'ordine di  $n$  numeri interi. Il primo si mette in ultima posizione, il secondo in penultima ecc.

**Esercizio 11** Calcolare la media aritmetica di un vettore di numeri reali.

**Esercizio 12** Scrivere un programma che legga una frase, sostituisca tutte le sequenze di due o più bianchi per un solo bianco e visualizzi la frase restante.

**Esercizio 13** Scrivere un programma che sposti una parola letta dalla tastiera da sinistra fino a destra dello schermo.

**Esercizio 14** Scrivere una funzione conversione che riceva come parametro una stringa rappresentando una data in formato "gg/mm/aa", come 17/11/91, e la restituisca in formato di testo: 17 novembre 1991.

**Esercizio 15** Progettare un programma che determini la media del numero di ore

Soluzioni degli esercizi sul sito Web [www.mheducation.it](http://www.mheducation.it)

lavorate durante tutti i giorni della settimana.

**Esercizio 1** Un vettore di  $n$  elementi è detto simmetrico se l'elemento che occupa la posizione  $i$ -esima coincide con quello che occupa la posizione  $(n - i)$ -esima. Per esempio, il vettore che immagazzina i valori 2, 4, 5, 4, 2 è simmetrico. Scrivere una funzione che decida se il vettore di  $n$  dati che riceve come parametro è simmetrico.

**Esercizio 2** Scrivere una funzione che riceva come parametro una matrice quadrata di ordine  $n$ , e ne generi la trasposta.

**Esercizio 3** Scrivere un programma che legga un numero naturale dispari  $n$  minore o uguale a 11, calcoli e visualizzi il quadrato magico di ordine  $n$ . Un quadrato di ordine  $n \times n$  si dice che è magico se contiene i valori 1, 2, 3, ...,  $n^2$ , e comple la condizione che la somma dei valori immagazzinati in ogni riga e colonna coincida.

**Esercizio 4** Scrivere una funzione che trovi l'elemento maggiore e minore di una matrice, nonché le posizioni che occupa e li visualizzi su schermo.

**Esercizio 5** Scrivere una funzione che riceva come parametro una matrice quadrata di ordine  $n$  e decida se è simmetrica. Una matrice quadrata di ordine  $n$  è simmetrica se  $a[i][j] = a[j][i]$  per tutti i valori degli indici  $i, j$ .

**Esercizio 6** Codificare un programma C++ che permetta di visualizzare il triangolo di Pascal:

1					
1					1
1				2	1
1		3	3	1	
1	4	6	4	1	
1	5	10	10	5	1

Nel triangolo di Pascal ogni numero è la somma dei due numeri situati su di esso. Questo problema si deve risolvere utilizzando prima un array bidimensionale e, poi, un array di una sola dimensione.

**Esercizio 7** Scrivere una funzione che riceva un numero scritto in una certa base  $b$  come stringa di caratteri e lo trasformi nello stesso numero scritto in un'altra certa base  $b_1$  come stringa di caratteri. Le basi  $b$  e  $b_1$  sono maggiori o uguali di 2 e minori o uguali di 16. Può supporre che il numero entri in una variabile numerica intero lungo, e usare la base 10 come base intermedia.

**Esercizio 8** Si consideri la seguente definizione:

```
struct TipoScarpa
{
    char modello;
    double prezzo;
};
```

Quale sarà l'output prodotto dal seguente codice?

```
TipoScarpa scarpa1, scarpa2;
scarpa1.modello = 'A';
scarpa1.prezzo = 34,50;
cout << scarpa1.modello << " euro " << scarpa1.
prezzo << endl;
scarpa2 = scarpa1;
cout << scarpa2.modello << " euro " << scarpa2.
prezzo << endl;
```

**Esercizio 9** Data la seguente struttura studente e una variabile uno\_studente di quel tipo, scrivere un'istruzione che visualizzi il nome dello studente nel formato cognome, nome.

```
struct studente
{
    string nome;
    string cognome;
    int eta;
    int voto;
};
```

```
studente uno_studente;
```

Scrivere una funzione che visualizzi tutti i dati della variabile di tipo studente.

**Esercizio 10** Trovare gli errori del seguente codice:

```
void scrive(struct data f);
int main()
{
    struct data
    {
        int giorno;
```

```

int mese;
int anno;
char mese[];
} ff;
ff = {1,1,2000,"GENNAIO"};
scrive(ff);
return 1;
}

```

**Esercizio 1** Scrivere funzioni che permettano di fare le operazioni di somma, sottrazione, moltiplicazione e quoziente di numeri complessi in forma binomiale,  $a+bi$ . Il tipo complesso deve definirsi come una struttura. Aggiungere funzioni per leggere e scrivere numeri complessi, e un programma che permetta interattivamente di realizzare operazioni con numeri complessi.

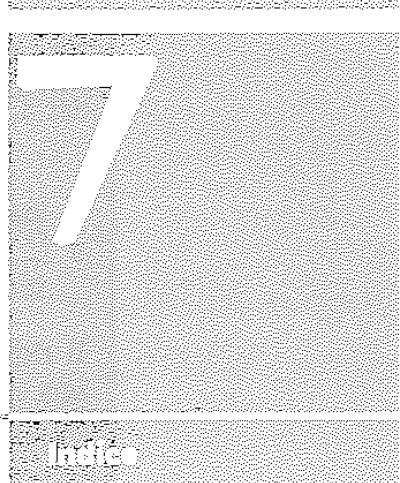
**Esercizio 2** Un punto nel piano si può rappresentare tramite una struttura con due campi. Scrivere un programma che realizzi le seguenti operazioni con punti nel piano.

Dati due punti calcolare la distanza tra di loro. Dati due punti determinare il punto medio della riga che li unisce.

**Esercizio 3** I protocolli IPv4 di indirizzamento di rete Internet definiscono l'indirizzo di ogni nodo come un intero su 32 bit. Per la visualizzazione agli utenti si separano di solito di quattro in quattro byte separandoli per punti. Scrivere una funzione che visualizzi un intero come un indirizzo IP (quattro byte dell'intero separati per punti).

**Esercizio 4** Una libreria vuole avere l'inventario dei libri. Perciò decide di creare una base di dati in cui si immagazzinano le seguenti informazioni per libro: indice progressivo, titolo; data di pubblicazione; autore; numero di copie; prezzo di vendita. Scrivere funzioni che permettano di leggere i dati del database e di visualizzarli a schermo.

# Puntatori e Riferimenti



<b>7.1</b>	Riferimenti	<b>7.8</b>	Aritmetica dei puntatori
<b>7.2</b>	Puntatori	<b>7.9</b>	Puntatori "costanti" e puntatori "a costanti"
<b>7.3</b>	Puntatori null	<b>7.10</b>	Puntatori come argomenti di funzioni
<b>7.4</b>	Puntatore a puntatore	<b>7.11</b>	Puntatori a funzioni
<b>7.5</b>	Puntatori e array	<b>7.12</b>	Puntatori a strutture
<b>7.6</b>	Array di puntatori		
<b>7.7</b>	Puntatori a stringhe		

## INTRODUZIONE

Si dice che i puntatori in C (e in C++) siano molto difficili da capire e da utilizzare, ma non è così. Essi sono una delle tecniche che rendono potente il linguaggio C ma non sono affatto difficili da apprendere. Il *puntatore* è un tipo di dato predeterminato a rappresentare indirizzi di memoria. Una variabile di tipo puntatore contiene quindi l'indirizzo di memoria di una variabile o di una funzione dette, rispettivamente, variabile e funzione puntate. Il dato in sé è un intero positivo (su 4 o 8 byte a seconda che le architetture hardware siano a 32 o 64 bit), ma si intende che esso rappresenta specificamente un indirizzo di memoria.

In questo capitolo si approfondiranno differenti aspetti dei puntatori, in particolare:

- l'assegnamento dinamico della memoria;
- l'aritmetica dei puntatori;
- gli array di puntatori;
- i puntatori a puntatori, funzioni e strutture.

Il C++ ha poi recuperato dal Pascal i riferimenti (in C non esistono), che sono nomi aggiuntivi (cioè degli "alias") per una variabile. Implementati mediante puntatori nascosti, argomenti di funzione di questo tipo consentono il passaggio di argomenti per riferimento (tecnica preclusa al C).

### 7.1 Riferimenti

Quando si definisce una variabile, se ne determinano tre attributi fondamentali: il suo *nome*, il suo *tipo* e il suo *indirizzo* in memoria.

**Esempio 7.1**

```
int n;      // associa al nome n, il tipo int e l'indirizzo della
           // posizione di memoria dove scrivere il valore di n
```

0x4ffd34  
 n   
 int

Il riquadro rappresenta le celle di memoria occupate dalla variabile; alla sua sinistra c'è il nome della variabile, l'indirizzo è sopra e sotto c'è il tipo. Se il valore della variabile è noto, lo si può scrivere nel riquadro.

0x4ffd34  
 n   
 int

Al valore di una variabile si accede tramite il suo nome. Per esempio, si può stampare il valore di n con l'istruzione

```
cout << n;
```

All'indirizzo della variabile si accede mediante l'*operatore di indirizzo* &. Per esempio, si può stampare l'indirizzo di n con l'istruzione

```
cout << &n;
```

L'operatore di indirizzo applicato al nome di una variabile restituisce il suo indirizzo.

**Esempio 7.2**

Ottenere il valore e l'indirizzo di una variabile.

```
int main()
{
    int n = 75;
    cout << "n = " << n << endl;      // visualizza il valore di n
    cout << "&n = " << &n << endl;      // visualizza indirizzo di n
}
```

**Esecuzione**

```
n = 75
&n = 0x7ffc29332824
```

**Nota**

0x7ffc29332824 è un indirizzo espresso in codice esadecimale, come indicato dal prefisso 0x.

Un *riferimento* a una variabile è un ulteriore nome per essa, un "alias". Il C++ introduce un nuovo tipo di dato: il "riferimento al tipo". Si dichiara utilizzando la parola chiave `reference`.

zando l'operatore & suffisso al tipo di dato riferito. Per esempio, `int&` è il tipo "riferimento al tipo `int`". Questa dichiarazione di tipo serve poi per definire una variabile di quel tipo, per esempio:

```
int& r = n;
```

definisce una variabile `r` di tipo "riferimento a `int`" e la inizializza anche al valore `n`, il che significa che `r` è un puro sinonimo di `n`.

### Ricorda

Il tipo "riferimento al tipo" viene utilizzato come un normale tipo di dato in una definizione di variabile, ma quest'ultima deve essere inizializzata contestualmente alla definizione, cioè deve essere seguita dall'operatore di assegnamento e dal nome di una variabile già definita dello stesso tipo del tipo riferito.

### Esempio 7.3

Utilizzo di riferimenti.

```
int main()
{
    int n = 75;
    int& r = n;      // r è un riferimento per n
    cout << "n = " << n << ", r = " << r << endl;
}
```

### Esecuzione

`n = 75, r = 75`

### Nota

I due identificatori `n` e `r` non indicano due variabili con lo stesso valore ma riferiscono la stessa variabile (si veda l'esempio successivo).

### Esempio 7.4

Le variabili `n` e `r` hanno lo stesso indirizzo di memoria

```
int main()
{
    int n = 75;
    int& r = n;
    cout << "&n = " << &n << ", &r =" << &r << endl;
}
```

### Esecuzione

`&n = 0x7ffc6dc4f75c, &r = 0x7ffc6dc4f75c`

### QUESTIONI

Il carattere & ha tre usi in C++:

1. quando si utilizza come prefisso al nome di una variabile, ne restituisce l'indirizzo;
2. quando si utilizza come suffisso a un tipo nella definizione di un riferimento, dichiara quest'ultimo essere sinonimo della variabile utilizzata per la sua inizializzazione;
3. quando si utilizza come suffisso a un tipo nella dichiarazione dei parametri formali di una funzione, dichiara questi ultimi essere riferimenti delle corrispondenti variabili passate alla funzione.

## 7.2 Puntatori

Ogni volta che si definisce una variabile il compilatore riserva un'area di memoria per scriverne il contenuto. La quantità di spazio riservato dipende dal tipo della variabile. Lo spazio è allocato in maniera contigua a partire da una cella il cui indirizzo viene preso come *indirizzo di memoria* della variabile. Quando si nomina (o si riferisce) una variabile, il compilatore accede automaticamente al suo indirizzo di memoria risalendovi dal suo nome (o dal suo riferimento).

Questa potenzialità d'accesso è resa ancora più efficace dalla tecnologia dei **puntatori**. Il C e il C++ utilizzano il tipo di dato: il "puntatore al tipo". Si dichiara utilizzando l'operatore \* suffisso al tipo di dato puntato. Per esempio, `int*` è il tipo "puntatore al tipo `int`". Questa dichiarazione di tipo serve poi per definire una variabile di quel tipo, per esempio:

```
int* p = &n;
```

definisce una variabile `p` di tipo "puntatore a `int`" e la inizializza dandole come valore l'indirizzo di memoria della variabile `n`, il che significa che "`p` punta `n`". Le variabili di tipo puntatore non devono necessariamente essere inizializzate all'atto della loro definizione, cioè possiamo avere:

```
int* p;  
...  
p = &n;
```

Un *puntatore* è quindi una variabile che contiene l'indirizzo di memoria di un'altra variabile. Si chiama così appunto perché *punta* un'altra variabile. È importante distinguere bene l'indirizzo di una variabile dal suo valore. Nel caso precedente, il valore intero della variabile `n` è contenuto dentro la variabile stessa, mentre il suo indirizzo è contenuto nella variabile `p` di tipo "puntatore a intero" (Figura 7.1). Riassumendo:

- un *puntatore* è una *variabile*;
- un puntatore contiene un *indirizzo di memoria*;
- all'indirizzo di memoria contenuto nel puntatore c'è una variabile (oppure, come vedremo, la prima istruzione di una funzione), quindi si dice che un puntatore punta una variabile (o una funzione).

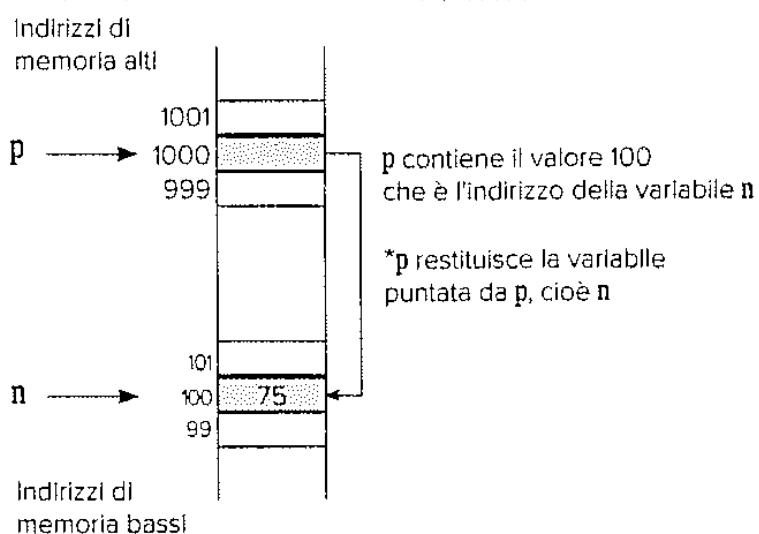


Figura 7.1

Relazioni fra variabile puntatore e variabile puntata.

0x4ffd30                            0x4ffd34

0x4ffd34	75
----------	----

p                                    n  
int\*                                    int

### Esempio 7.5

```
int main()
{
    int n = 78;
    int* p = &n; // p contiene l'indirizzo di n
    cout << "n = " << n << ", &n = " << &n << ", p " << p << endl;
    cout << "&p = " << &p << endl;
}
```

### Esecuzione

```
n = 78, &n = 0x7ffe47515c0c, p 0x7ffe47515c0c
&p = 0x7ffe47515c10
```

Altri esempi di variabili di tipo puntatore:

```
long* ptr1; // Puntatore a un tipo di dato intero lungo (long int)
char* ptr2; // Puntatore a un tipo di dato char
float* f; // Puntatore a un tipo di dato float
```

Dopo aver definito e inizializzato una variabile puntatore, il passo successivo è quello di utilizzarlo per riferire la variabile puntata, sia in scrittura sia in lettura. Arrivare alla variabile puntata partendo da un suo puntatore significa *dereferenziare* il puntatore. L'operazione si fa con il cosiddetto operatore di

indirezione \* prefisso alla variabile puntatore. Facendo riferimento al precedente esempio:

`*p`

restituisce il nome della variabile puntata da p, cioè n. A questo punto il nome della variabile può essere usata sia a destra dell'operatore di assegnamento (in lettura) sia a sinistra (in scrittura). Per esempio:

`int q = *p;`

significa che la variabile q prende lo stesso valore che c'è dentro la variabile n, mentre:

`*p = 75;`

significa che la variabile n prende il valore 75.

Analogamente:

`cin >> *p;`

legge dallo standard input un intero e lo mette dentro la variabile n, mentre:

`cout << *p;`

scrive sullo standard output il valore letto dentro la variabile n.

Il seguente programma riepiloga i concetti di creazione, inizializzazione e dereferenziamento di una variabile puntatore.

```
char c;           // variabile carattere
int main()
{
    char *pc;      // puntatore a una variabile carattere
    pc = &c;
    for (c = 'A'; c <= 'Z'; c++) cout << *pc;
    return 0;
}
```

### Esecuzione

ABCDEFGHIJKLMNPQRSTUVWXYZ

La variabile pc è un puntatore a una variabile carattere. La riga `pc = &c` assegna a pc l'indirizzo della variabile c. Il ciclo for mette in c le lettere dell'alfabeto e l'istruzione `cout << *pc;` visualizza il contenuto della variabile puntata da pc. c e \*pc riferiscono la stessa posizione in memoria.

Abbiamo visto che il tipo puntatore è riferito a uno specifico tipo "puntato". Così, per esempio, se si dichiara un tipo puntatore a float, a una variabile di quel tipo non si può assegnare l'indirizzo di una variabile di tipo char o int. Per esempio, questo segmento di codice non funzionerà

```
float* fp;
char c;
fp = &c;      // non è valido
```

appunto perché C++ non permette l'assegnamento dell'indirizzo della variabile c di tipo char a fp che è una variabile di tipo "puntatore a float".

Comunque in C/C++ si può anche dichiarare un tipo "puntatore a qualsiasi tipo di dato". Il metodo è di dichiarare il puntatore come un puntatore a void. Per esempio:

```
void* ptr; // dichiara un puntatore a qualunque tipo di dato
```

ptr potrà indirizzare qualunque tipo di variabile e qualunque posizione di memoria.

### 7.3 Puntatori null

Normalmente, un puntatore inizializzato adeguatamente punta a qualche posizione specifica della memoria. Un puntatore non inizializzato, come qualsiasi variabile, ha un valore aleatorio dato dallo stato dei bit delle celle di memoria da lui occupate al momento della sua definizione. Per questo è importante assegnare sempre un valore a un puntatore, eventualmente il valore 0 o NULL.

Un puntatore a cui viene assegnato il valore 0 o NULL non punta alcun oggetto, cioè non indirizza alcun dato valido in memoria, ed è noto come *puntatore nullo*. NULL è una costante simbolica definita in vari header file, compreso <iostream>. Inizializzare un puntatore a 0 o NULL è equivalente, ma in C++ si preferisce generalmente 0. Per esempio:

```
char* p = 0;
char* p = NULL;
```

Si può testare il valore del puntatore:

```
if (p == 0) ...
```

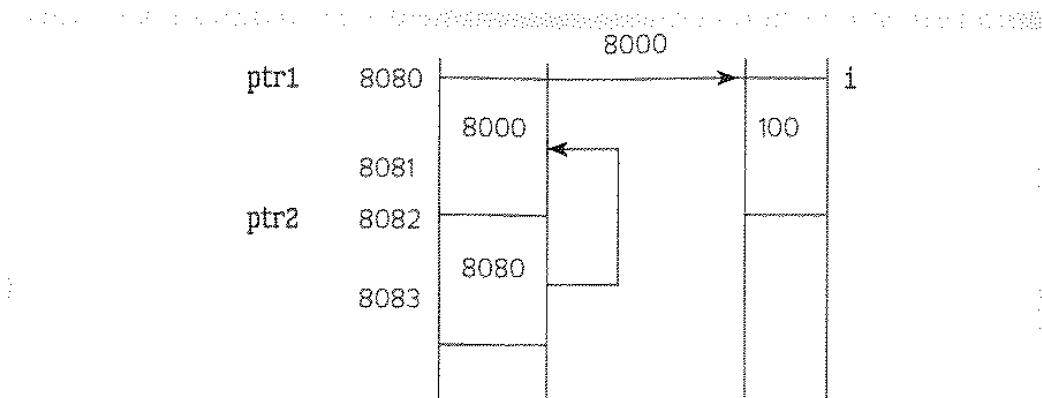
oppure

```
if (p != NULL) ...
```

Interpretandola letteralmente, l'istruzione di assegnamento p=0 assegna al puntatore p l'indirizzo di memoria 0x0 che è sicuramente fuori dal segmento di memoria riservato al programma dal sistema operativo. Se si tentasse quindi di dereferenziare p si commetterebbe un errore a run-time. I puntatori nulli si utilizzano spesso per indicare la terminazione di qualcosa, per esempio di liste dinamiche, ma sono frequenti cause di errori. Prima di dereferenziare un puntatore che scorre lungo una lista si deve sempre controllare che esso non sia nullo.

### 7.4 Puntatore a puntatore

Un tipo puntatore può puntare qualunque tipo, anche un altro puntatore. Per dichiarare un tipo "puntatore a puntatore" si usano naturalmente due asterischi. Per esempio, il seguente ptr2 è un puntatore a puntatore di interi.

**Figura 7.2****Un puntatore a puntatore di interi.**

```
int i = 100;
int* ptr1 = &i;
int** ptr2 = &ptr1;
```

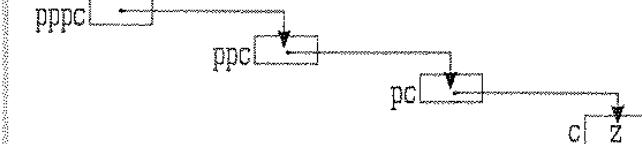
`ptr1` e `ptr2` sono puntatori diversi. `ptr1` è puntatore a interi e punta la variabile `i` di tipo int. `ptr2` è puntatore a puntatori di interi e punta la variabile `ptr1` (Figura 7.2).

Per dereferenziare il puntatore di puntatori si possono usare i due asterischi. Per esempio, le seguenti istruzioni sono equivalenti:

```
i = 95;           // Assegna 95 a i
*ptr1 = 95;     // Assegna 95 a i
**ptr2 = 95;    // Assegna 95 a i
```

#### Esempio 7.6

```
char c = 'z';
char* pc = &c;
char** ppc = &pc;
char*** pppc = &ppc;
****pppc = 'm';      // cambia il valore di c a 'm'
```



## 7.5 Puntatori e array

In C/C++ gli array sono implementati mediante puntatori (Figura 7.3). Il nome di un vettore è esattamente un puntatore al primo elemento dell'array. Prendiamo per esempio il seguente vettore:

```
int gradi[5] = {10, 20, 30, 40, 50};
```

L'istruzione `cout << gradi[0]` visualizzerà ovviamente 10, ma lo farà anche l'istruzione `cout << *gradi`, perché il nome `gradi` è un puntatore al primo elemento del

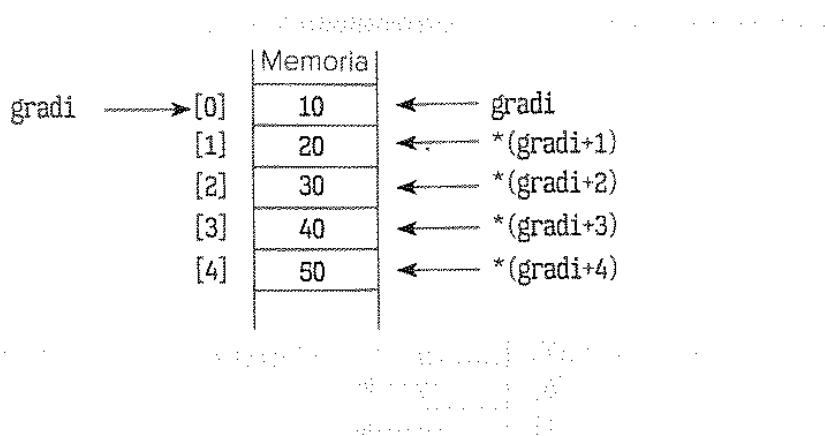


Figura 7.3  
Un array.

vettore. Quindi dereferenziandolo si prende il primo elemento del vettore, ma non basta; come sarà chiarito nel Paragrafo 7.8, se incrementiamo il puntatore di un valore pari a un indice del vettore e poi lo dereferenziamo andiamo a prendere l'elemento del vettore corrispondente a quell'indice, cioè:

gradi + 0	punta a	gradi[0]
gradi + 1	punta a	gradi[1]
gradi + 2	punta a	gradi[2]
gradi + 3	punta a	gradi[3]
gradi + 4	punta a	gradi[4]

Pertanto, per leggere o scrivere un elemento di un array si possono utilizzare indifferentemente entrambe le notazioni, quella vettoriale e quella basata sui puntatori.

Il nome di un array è però una *costante puntatore*, non una variabile puntatore. Non si può cambiarne il valore. L'osservazione è importante perché, ovviamente, si può cambiare i valori delle variabili di tipo puntatore per farle puntare valori differenti in memoria. Per esempio, il seguente programma inizializza due variabili in virgola mobile e manda un puntatore a puntarle entrambe in successione:

```
int main()
{
    float v1 = 756.423;
    float v2 = 900.545;
    float *p_v;
    p_v = &v1;
    cout << "Il primo valore è " << *p_v << endl;
    p_v = &v2;
    cout << "Il secondo valore è " << *p_v << endl;
    return 0;
}
```

## Esecuzione

Il primo valore è 756.423  
Il secondo valore è 900.545

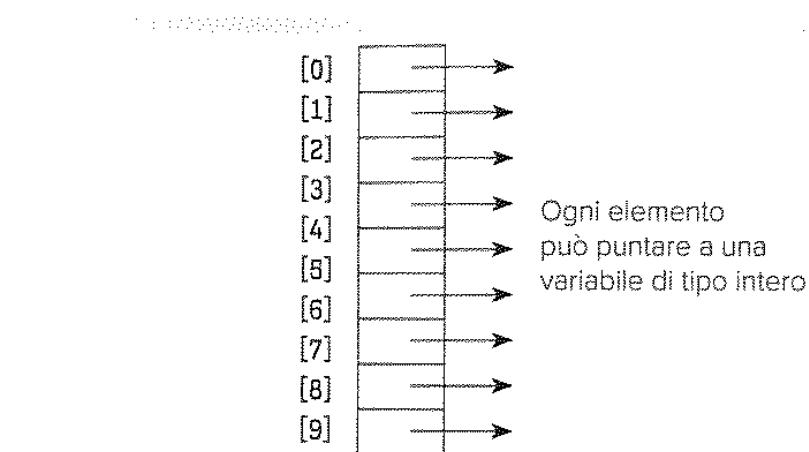


Figura 7.4

Un array di 10 puntatori a interi:

## 7.6 Array di puntatori

Se bisogna definire parecchi puntatori a uno stesso tipo si può utilizzare un *array di puntatori*. Un array di puntatori è un vettore i cui elementi sono puntatori dello stesso tipo. Per esempio:

```
int* ptr[10]; // riserva un array di 10 puntatori a interi
```

definisce un vettore di dieci variabili puntatore a interi (Figura 7.4). Ogni elemento contiene un indirizzo che *punta* ad altre variabili di tipo intero in memoria e può essere riassegnato. Così, per esempio,

```
ptr[6] = &eta // ptr[6] punta all'indirizzo di età
```

Un altro esempio di array di puntatori, in questo caso di caratteri, è:

```
char* punti[25]; // array di 25 puntatori caratteri
```

Queste due definizioni sono quindi equivalenti:

```
char stringa[] = "Ciao vecchio mondo";
char* stringa = "Ciao vecchio mondo";
```

## 7.7 Puntatori a stringhe

Si consideri la seguente dichiarazione di un array di caratteri:

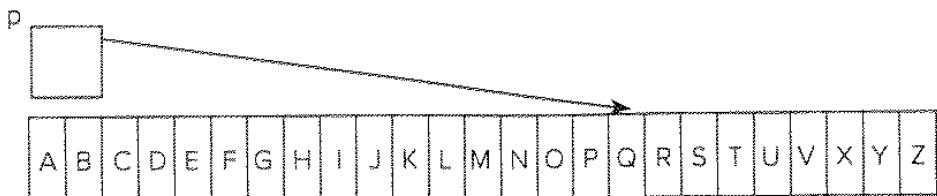
```
char alfabeto[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Dichiaramo ora p un puntatore a char

```
char* p;
```

Facciamo puntare p al primo carattere di alfabeto scrivendo:

```
p = alfabeto; // o anche p = &alfabeto[0]
```



**Figura 7.5**  
Un puntatore ad alfabeto[15].

Ora, l'istruzione:

```
cout << *p << endl;
```

stampa *A*, dato che *p* punta al primo elemento della stringa. Possiamo anche assegnare:

```
p = alfabeto + 15; // o anche p = &alfabeto[15];
```

facendo puntare *p* al sedicesimo carattere (la lettera *P*). Ma non si può assegnare:

```
p = &alfabeto;
```

perché *p* e *alfabeto* sono puntatori al tipo *char*, e l'istruzione vorrebbe assegnare a *p* l'indirizzo di un puntatore a *char* e non l'indirizzo di una variabile di tipo *char*. Si produrrebbe un errore di compilazione perché *p* è di tipo puntatore a *char* e non puntatore a puntatore a *char* (Figura 7.5).

Si può ovviamente definire un array di puntatori a stringhe:

```
char* nomi_mesi[12] = { "Gennaio", "Febbraio", "Marzo",
                        "Aprile", "Maggio", "Giugno",
                        "Luglio", "Agosto", "Settembre",
                        "Ottobre", "Novembre",
                        "Dicembre" };
```

Il seguente programma implementa una funzione per contare il numero di caratteri di una stringa. Nel primo programma la stringa si descrive utilizzando un array e nel secondo, si descrive utilizzando un puntatore.

#### Esempio 7.7

```
int lunghezza_stringa1(char str[])
{
    int posizione = 0;
    while (str[posizione] != '\0') posizione++;
    return posizione;
}
int lunghezza_stringa2(register char* str)
{
```

```

    int contatore = 0;
    while (*str++) contatore++;
    return(contatore);
}

int main()
{
    char str[] = "Università Politecnica delle Marche";
    cout << "La lunghezza di " << str << " è "
        << lunghezza_stringa1(str) << " caratteri " << endl;
    cout << "La lunghezza di " << str << " è "
        << lunghezza_stringa2(str) << " caratteri " << endl;
}

```

### Esecuzione

La lunghezza di Università Politecnica delle Marche è 36 caratteri  
 La lunghezza di Università Politecnica delle Marche è 36 caratteri

## 7.6 Aritmetica dei puntatori

Se un puntatore a un tipo T viene incrementato (o decrementato) di 1, il suo valore viene in realtà incrementato (o decrementato) di un numero pari alla dimensione del tipo T espressa in byte. In generale, incrementare o decrementare di un intero *n* un puntatore significa incrementarlo o decrementarlo di un numero di byte pari a *n* moltiplicato per la dimensione del tipo puntato. Per esempio, date le definizioni:

```

int gradi[5] = {10, 20, 30, 40, 50};
int* p = gradi;

```

*p* punta al primo intero 10 in *gradi*; ma dopo l'istruzione

```

p++;

```

*p* punterà al secondo intero 20. Poiché ogni elemento di *gradi* occupa 4 byte, il valore di *p* è stato incrementato di 4 (e non di 1).

Si può quindi utilizzare questa tecnica per attraversare ogni elemento di un vettore senza usare una variabile di indice. Per esempio:

```

p = &gradi[0];
for (int i = 0; i < strlen(gradi); i++)
    cout << *p++ << endl;

```

Se il vettore da attraversare è una stringa si può semplificare il ciclo considerando il fatto che essa termina con il carattere nullo. Per esempio, utilizzando l'istruzione *while* per realizzare il ciclo si può omettere la variabile di controllo:

```

char messaggio[] = "Ciao vecchio mondo";
p = messaggio;
while (*p) cout << *p++;

```

Il while cicla finché \*p ha un valore diverso da zero; si stampa il carattere e si incrementa p per puntare al successivo carattere. Quando si raggiunge il byte nullo alla fine della stringa il ciclo termina. Nell'esempio sopra p++ incrementa in effetti di 1 ma solo perché ogni carattere occupa un solo byte.

L'esempio che segue mostra un puntatore che attraversa una stringa di caratteri e converte qualunque carattere minuscolo in maiuscolo.

### Esempio 7.8

```
int main()
{
    char *p;
    char StringaTesto[80];
    cout << "Introdurre una stringa da convertire:\n";
    cin.getline(StringaTesto, sizeof(StringaTesto));
    p = StringaTesto;
    while (*p)
        if ((*p >= 'a') && (*p <= 'z')) *p++ = *p-32;
        else p++;
    cout << "La stringa convertita è:" << endl;
    cout << StringaTesto << endl;
}
```

### Esecuzione

```
Introdurre una stringa da convertire:
a dicembre 2020 il C torna a essere il linguaggio + diffuso al mondo
La stringa convertita è:
A DICEMBRE 2020 IL C TORNA A ESSERE IL LINGUAGGIO + DIFFUSO AL MONDO
```

Se il carattere letto cade tra 'a' e 'z', cioè è una lettera ASCII minuscola, l'assegnamento

```
*p++ = *p-32;
```

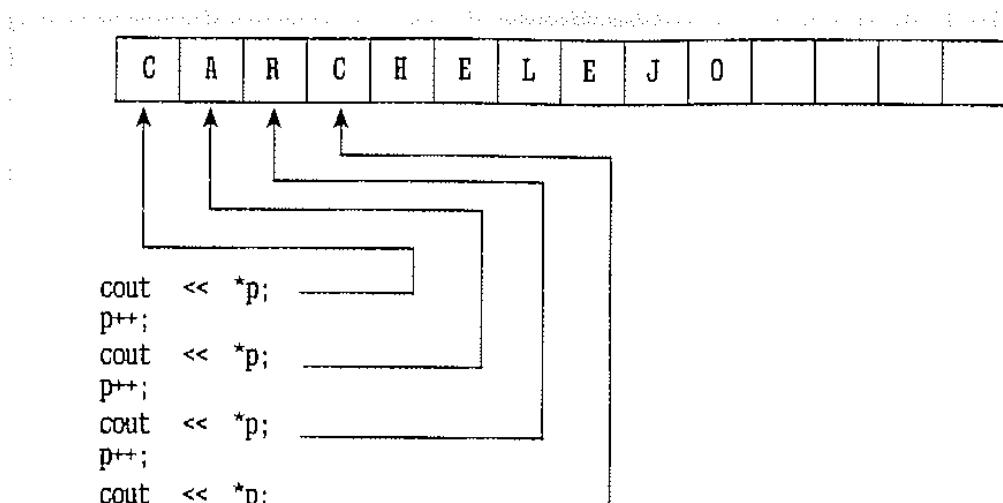
sottrae 32 dal codice ASCII convertendo la lettera in maiuscolo e poi incrementa il puntatore all'elemento successivo del vettore (Figura 7.6).

I puntatori possono anche essere sottratti. Per esempio:

```
char messaggio[] = "Ciao vecchio mondo";
p = messaggio;
q = p+1;
while (*q) cout << q - p << *p++ << endl;
```

però non possono essere sommati, moltiplicati o divisi.

- Non si possono sommare due puntatori.
- Non si possono moltiplicare due puntatori.
- Non si possono dividere due puntatori.



**Figura 7.6** L'esempio di visualizzazione della funzione del punto e della virgola `*p++` si utilizza per accedere incrementalmente alla stringa.

## 7.9 Puntatori “costanti” e puntatori “a costanti”

Abbiamo già visto che il nome di un array è un puntatore costante: il suo valore non può cambiare ma può essere cambiato il valore della variabile puntata. Pensiamo, al contrario, a un puntatore a una costante: il suo valore può cambiare (perché è una variabile) ma non può cambiare il valore della costante puntata (perché appunto è una costante).

Vediamo come definire puntatori “costanti” e puntatori “a costanti”.

### Puntatori costanti

Per creare un puntatore costante si deve utilizzare il seguente schema:

```
<tipo_dato>* const <nome_puntatore> = <indirizzo_variabile>;
```

Per esempio, si considerino le seguenti:

```
int x;
int e;
int* const p1 = &x; // p1 è una costante puntatore a intero
```

p1 è una costante puntatore che punta a x. p1 è costante, ma \*p1 è una variabile, pertanto si può cambiare il valore di \*p1, ma non p1. Per esempio, il seguente assegnamento è corretto, dato che si cambia il contenuto di memoria dove p1 punta, ma non il puntatore in sé:

```
*p1 = e;
```

D'altra parte, il seguente assegnamento non è corretto, dato che si tenta di cambiare il valore del puntatore:

```
p1 = &e;
```

Per creare un puntatore costante a una stringa:

```
char* const nome = "Luigi";
nome non si può modificare, *nome sì. Pertanto,
strcpy(*nome = "C");
è corretto, dato che si modifica il dato puntato da nome, ma non è corretto questo assegnamento:
nome = &Altra_Stringa;
perché tenta di modificare proprio il valore del puntatore (che però è un valore costante).
```

### Puntatori a costanti

Lo schema per definire un puntatore a una costante è:

```
const <tipo_dato>*<nome_puntatore> = <indirizzo_const_o_stringa>;
```

Alcuni esempi sono:

```
const int x = 25;
const int e = 50;
const int *p1 = &x; // manda il puntatore a puntare x
```

dove p1 è un puntatore alla costante x. x è costante, non lo è il puntatore, quindi si può scrivere:

```
p1 = &e; // manda il puntatore a puntare e (invece che x)
```



**Nota**  
La definizione di un puntatore costante ha la parola riservata const davanti al nome del puntatore, mentre il puntatore a una costante ha la parola riservata const davanti al tipo di dato.

Qualunque tentativo di modificare il contenuto della posizione di memoria puntata da p1, per esempio:

```
*p1 = 15;
```

provocherà un errore in fase di compilazione. La creazione di un *puntatore a una costante stringa* si può quindi fare così:

```
const char *cognome = "Mortimer";
```

### Puntatori costanti a costanti

L'ultimo caso da considerare è quello di puntatori costanti a costanti, cosa che si può fare con lo schema seguente:

```
const <tipo_dato>*<const nome_puntatore> = <indirizzo_const_o_stringa>;
```

Un esempio può essere:

```
const int x = 25;
const int* const p1 = &x;
```

che indica: p1 è un puntatore costante che punta alla costante intera x. Qualunque intento di modificare p1 oppure \*p1 produrrà un errore di compilazione.

- Se è noto che un puntatore punterà sempre la stessa posizione e non dovrà mai essere ricollocato, lo si definisca come "puntatore costante".
- Se è noto che il dato puntato dal puntatore non dovrà mai essere modificato, si definisca il puntatore come "puntatore a costante".

### Esempio 7.9

Il seguente esempio mostra le differenze fra puntatore a costante e puntatore costante.

```
// Questo pezzo di codice dichiara quattro variabili
// un puntatore p; un puntatore costante cp; un puntatore pc a una
// costante e un puntatore costante cpc a una costante
```

```
int *p;           // puntatore a un int
++(*p);          // incremento int *p
++p;             // incrementa un puntatore p
int * const cp; // puntatore costante a un int
++(*cp);         // incrementa int *cp
++cp;            // non valido: il puntatore cp è costante
const int * pc; // puntatore a una costante int
++(*pc);         // non valido: int * pc è costante
++pc;            // incrementa puntatore pc
const int * const cpc; // puntatore costante a costante int
++(*cpc);        // non valido: int *cpc è costante
++cpc;           // non valido: puntatore cpc è costante
```

**Lo spazio non è significativo nella dichiarazione di puntatori. Le dichiarazioni seguenti sono equivalenti:**

```
int* p;
int * p;
int *P;
```

## 7.10 Puntatori come argomenti di funzioni

Spesso, si vuole che una funzione restituisca più di un valore oppure modifichi variabili passate come argomenti. Si possono ottenere questi risultati passando puntatori, per esempio nomi di vettori, come argomenti attuali.

Si consideri la seguente definizione della procedura Incrementare5, che incrementa un intero di 5:

```
void Incrementare5(int* i)
{
    *i = *i + 5;
}
```

La chiamata a questa funzione passa l'indirizzo della variabile da incrementare, per esempio:

```
int j = 10;
Incrementare5(&j);
```

È possibile combinare passaggio per riferimento e per valore. Per esempio, la funzione `funz1` definita come

```
int funz1(int* s, int t)
{
    *s = 6;
    t = 25;
}
```

potrebbe essere chiamata così:

```
int i= 5, j= 7;
funz1(&i, j); //chiamata a funz1
```

Quando si ritorna dalla funzione `funz1` dopo la sua esecuzione, `i` sarà uguale a 6 e `j` sarà sempre 7, dato che si passò per valore.

Passare il nome di un array a una funzione equivale a passare il primo elemento del vettore, quindi facendo scorrere il puntatore se ne possono modificare gli elementi.

Il passaggio per riferimento che il C++ ha reintrodotto è però più comodo del passaggio per indirizzo tipico del C perché è più facile da scrivere (e da leggere). Per esempio, supponiamo di raccogliere in una struttura la massima e la minima temperatura di una giornata:

```
struct temperatura {
    float massima;
    float minima;
};
```

e di dover scrivere una procedura che modifichi i campi di una variabile di questo tipo passatole come argomento.

#### Metodo C++

```
void scrivitemp(temperatura& t)
{
    float attuale;
    leggitempattuale(attuale);
    if (attuale > t.alta) t.alta = attuale;
    else if (attuale < t.bassa) t.bassa = attuale;
}
```

Poiché il parametro `t` è un riferimento, la funzione lavora direttamente sulla variabile passata, per esempio la `temp` qui sotto:

```
temperatura temp;
scrivitemp(temp);
```

### Metodo C

In C gli argomenti si passano solo per valore; pertanto, per modificare una variabile bisognerà passarne l'indirizzo come argomento a un parametro puntatore. La stessa funzione scrivitemp scritta in C sarebbe:

```
void scrivitemp(struct temperatura *t)
{
    float attuale;
    leggitemppattuale(attuale);
    if (attuale > (*t).alta) (*t).alta = attuale;
    else if (attuale < (*t).bassa) (*t).bassa = attuale;
}
```

*Il passaggio per indirizzo (tramite puntatore) è più comune in C che in C++ perché in C++ esiste il passaggio per riferimento, che è più comodo.*

### 7.11 Puntatori a funzioni

Finora sono stati analizzati puntatori a dati. È possibile anche creare puntatori che puntino a funzioni. Invece di indirizzare dati, i puntatori di funzioni puntano la prima istruzione di un codice eseguibile. Infatti non solo i dati ma anche le istruzioni stanno in memoria a certi indirizzi. In C/C++ si possono assegnare gli indirizzi iniziali di funzioni a puntatori. Tali funzioni si possono chiamare in un modo indiretto, cioè tramite un puntatore il cui valore è uguale all'indirizzo iniziale della funzione in questione.

La sintassi generale per *definire* un puntatore a funzione è:

*Tipo\_di\_ritorno (\*PuntatoreFunzione) (<argomenti\_formali>)*

Questo formato indica al compilatore che *PuntatoreFunzione* è un puntatore a una funzione che restituisce il tipo *Tipo\_di\_ritorno* e una lista di parametri (Figura 7.7).

Un puntatore a una funzione è semplicemente un puntatore il cui valore è l'indirizzo del nome della funzione. Dato che il nome di una funzione è un puntatore, come il nome di un vettore, un puntatore a una funzione è un puntatore a un puntatore costante.

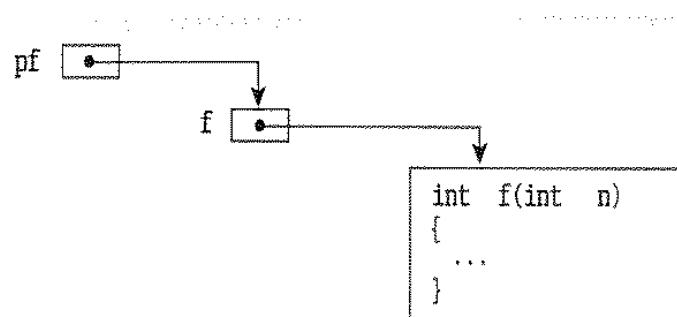


Figura 7.7

Puntatore a funzione.

**Esempio 7.10**

```
double (*fp) (int n);
float (*p) (int i, int j);
void (*sort) (int* ArrayInt, unsigned n);
unsigned (*search)(int CercareChiave, int* ArrayInt, unsigned n);
```

Il primo identificatore, fp, punta a una funzione che restituisce un tipo double e ha un unico parametro di tipo int. Il secondo puntatore, p, punta a una funzione che restituisce un tipo float e accetta due parametri di tipo int. Il terzo puntatore, sort, è un puntatore a una procedura e prende due parametri: un puntatore a int e un tipo unsigned. Da ultimo, search è un puntatore a una funzione che restituisce un tipo unsigned e ha tre parametri: un int, un puntatore a int e un unsigned.

La sintassi generale per *inizializzare* un puntatore a funzione è:

*PuntatoreFunzione* = *NomeDiUnaFunzione*

mentre quella per *chiamare* la funzione puntata è:

*(PuntatoreFunzione)* (<argomenti attuali>)

La funzione assegnata deve avere il tipo del ritorno e dei parametri giusti secondo il tipo del puntatore a funzione; in caso contrario, si produrrà un errore di compilazione.

**Esempio 7.11**

Supponiamo di avere un puntatore a funzione p come:

float (\*p) (int i, int j);

e una funzione esempio:

```
float esempio(int i, int j)
{
    return 3.14159 * i * i + j;
```

si può assegnare al puntatore l'indirizzo della funzione scrivendo:

p = esempio;

e dopo questo assegnamento si può scrivere la seguente chiamata alla funzione:

(\*p) (12, 45);

L'effetto è lo stesso che

esempio(12, 45);

Si può anche omettere l'asterisco e/o financo le parentesi nella chiamata (\*p) (12, 45), e scrivere in maniera banale:

p (12, 45);

I puntatori a funzione permettono anche di passare una funzione come argomento a un'altra funzione specificandone semplicemente il nome. Per esempio, vediamo come scrivere una funzione generale che calcoli la somma di qualche valore, cioè

$$f(1) + f(2) + \dots + f(n)$$

per qualunque funzione *f* che ritorni il tipo *double* e abbia un unico argomento di tipo *int*. Il seguente programma mostra una funzione *funzsomma* che ha due argomenti: *n*, il numero di termini della somma, e *f*, la funzione utilizzata. Così, la funzione *funzsomma* si potrà chiamare, per esempio, due volte calcolando la somma di:

$$\text{inversi}(k) = 1.0 / k \quad (k = 1, 2, 3, 4, 5)$$

$$\text{quadrati}(k) = k^2 \quad (k = 1, 2, 3)$$

#### Esempio 7.12

```
double inversi(int k)
{
    return 1.0 / k;
}

double quadrati(int k)
{
    return (double)k * k;
}

double funzsomma(int n, double (*f)(int k))
{
    double s = 0;
    for (int i = 1; i <= n; i++) s += f(i);
    return s;
}

int main()
{
    cout << "Somma di cinque inversi:" 
        << funzsomma(5, inversi) << endl;
    cout << "Somma di tre quadrati:" 
        << funzsomma(3, quadrati) << endl;
    return 0;
}
```

Il programma calcola le somme di

a)  $1 + \frac{1.0}{2} + \frac{1.0}{3} + \frac{1.0}{4} + \frac{1.0}{5}$

b)  $1.0 + 4.0 + 9.0$

e il suo output sarà:

Somma di cinque inversi: 2.283333

Somma di tre quadrati: 14

Alcune delle funzioni di libreria hanno come argomento un puntatore a funzione. Si deve passare loro il nome di una funzione (cioè un puntatore). Per esempio, `qsort()` utilizza il *quicksort* per ordinare un array di qualunque tipo di dato (si veda il Capitolo 16 sull'ordinamento di vettori), ma è necessario fornire una funzione che specifichi la relazione d'ordine desiderata per quel tipo di array. Nel programma che segue la funzione, `confrontare()` confronta due elementi (passatigli per indirizzo mediante puntatori a costante) e restituisce un numero negativo se il primo è minore del secondo, restituisce zero se sono uguali e un numero positivo se il primo è maggiore del secondo.

#### Esempio 7.13

```
int confrontare(const void* arg1, const void* arg2)
{
    return *(int*)arg1 - *(int*)arg2;
}

int main()
{
    int vettore[10] = {14, 17, 5, 21, 12, 40, 60, 55, 35, 15};
    qsort((void*)vettore, (size_t)10, sizeof(int), confrontare);
    cout << "Vettore ordinato: \n";
    for (int i = 0; i < 10; i++) cout << " " << vettore[i];
    return 0;
}
```

L'output del programma è:

Vettore ordinato: 5 12 14 15 17 21 35 40 55 60



I parametri della funzione `qsort()` sono:

- (`void*`) `vettore` array da ordinare;
- (`size_t`) `10` numero di elementi dell'array;
- `sizeof(int)` dimensione in byte di ciascun elemento dell'array;
- `confrontare` nome della funzione che confronta due elementi dell'array.

Certe applicazioni richiedono di selezionare fra numerose funzioni in base alla valutazione a run-time di certe condizioni. Un metodo è quello di utilizzare un'istruzione `switch` con parecchi case. Un'altra soluzione è di utilizzare un array di puntatori a funzione.

La sintassi generale di un array di puntatori a funzione è:

`tipoRitorno (*PuntatoreFunz[LunghezzaArray]) (<argomenti>);`

#### Esempio 7.14

```
double (*fp[2])(int n);
void (*ordinare[MAX_ORD])(int* ArrayInt, unsigned n);
```

`fp` punta a un array di funzioni: ognuna restituisce un valore `double` e ha un unico parametro di tipo `int`. `ordinare` punta un array di funzioni, ognuna delle quali restituisce un tipo `void` e prende due parametri: un puntatore a `int` e un `unsigned`.

### Esempio 7.14

- `funz` è un oggetto;
- `funz[]` è un array;
- `(*funz[])` è un array di puntatori;
- `(*funz[])()` è un array di puntatori a funzione.

Si può assegnare l'indirizzo delle funzioni all'array, fornendo le funzioni che già sono state dichiarate, per esempio:

```
int f1(int i) {...};
int f2(int i) {...};
int (*fp[2])() = {f1, f2};
```

Quello che segue è un programma che può sommare, sottrarre, moltiplicare o dividere numeri. Si scrive un'espressione semplice in input e il programma visualizza la risposta.

### Esempio 7.15

```
float somm(float x, float y)
{
    return x + y;
}

float sott(float x, float y)
{
    return x - y;
}

float molt(float x, float y)
{
    return x * y;
}

float divi(float x, float y)
{
    return x / y;
}

float (*f)(float x, float y);

int main()
{
    char segno, operatori[] = {'+', '-', '*', '/'};
    float(*funz[])(float, float) = {somm, sott, molt, divi};
```

```

float x, y, z;
cout << "Un semplice calcolatore\nEspressione: ";
cin >> x >> segno >> y;
for (int i = 0; i < 4; i++)
{
    if (segno == operatori[i])
    {
        f = funz[i];
        z = f(x, y);
        cout << x << " " << segno << " " << y << " = " << z;
    }
}

```

**Esecuzione**

Un semplice calcolatore

Espressione: 28 / 4

28 / 4 = 7

**7.12 Puntatori a strutture**

Un puntatore può anche puntare una struttura, come si usa nella rappresentazione di liste, alberi e grafi in generale.

Dati un tipo struct e una variabile di questo tipo, si può definire come al solito un puntatore a questo tipo e inizializzarlo in maniera che punti quella variabile. Per esempio:

```

struct persona
{
    char nome[30];
    int eta;
    int altezza;
    int peso;
};

persona impiegato = {"Mortimer, Peppe", 47, 182, 85};
persona *p;
p = &impiegato;

```

Come sappiamo dal capitolo precedente, per riferirsi a un campo di una variabile di tipo struttura si deve indicare il nome della variabile seguita dal punto e dal nome del campo, per esempio `impiegato.nome`. Se la variabile `impiegato` è indicata dereferenziando il puntatore `p` che ne contiene l'indirizzo si dovrebbe scrivere

`*p.nome`

ma l'operatore punto (`.`) ha precedenza sull'operatore asterisco (`*`) per cui bisogna mettere le parentesi tonde

`(*p).nome`

Questa notazione diventa un po' scomoda, e siccome è assai frequente differenziare un puntatore a una struttura, si è introdotta una notazione equivalente ma più immediata e intuitiva mediante l'operatore freccia a destra ( $\rightarrow$ ), che sostituisce le parentesi tonde, l'asterisco e il punto: l'espressione  $(^p).nome$  viene semplicemente resa come

$p \rightarrow nome$

#### Esempio 7.16

```
struct persona
{
    char nome[30];
    int eta;
    int altezza;
    int peso;
};

void mostra_persona(persona* ptr)
{
    cout << "Nome: " << ptr -> nome
        << "\tEta: " << ptr -> eta
        << "\tAltezza " << ptr -> altezza
        << "\tPeso: " << ptr -> peso << "\n";
}

int main()
{
    persona impiegati[] = {"Mortimer, Peppe", 47, 182, 85},
        {"Mason, Perry", 39, 170, 75},
        {"Clifford, Luigi", 18, 176, 80} ;
    persona* p;
    p = impiegati;
    for (int i = 0; i < 3; i++, p++) mostra_persona(p);
}
```

#### Esecuzione

Nome: Mortimer, Peppe	Età: 47	Altezza 182	Peso: 85
Nome: Mason, Perry	Età: 39	Altezza 170	Peso: 75
Nome: Clifford, Luigi	Età: 18	Altezza 176	Peso: 80

I puntatori sono uno degli strumenti più potenti della programmazione in C/C++. Anche se il loro uso può sembrare complicato, è senza dubbio fondamentale apprenderne bene il funzionamento.

In questo capitolo abbiamo visto che un puntatore è una variabile che contiene l'indirizzo di una posizione in memoria (cioè l'indirizzo di un'altra variabile di un qualunque tipo), mentre un riferimento è

Per evitare questo problema si può dare un altro nome per una stessa variabile. Abbiamo visto come definire puntatori e riferimenti e come utilizzarli. In particolare:

- per definire una variabile puntatore si colloca un asterisco tra il tipo di dato puntato e il nome del puntatore, per esempio: `int* p;`
- per prelevare il valore della variabile puntata si utilizza l'operatore di indirizzazione (`*`) applicato come prefisso al nome del puntatore. Il valore di `p` è l'indirizzo della variabile puntata, mentre il valore di `*p` è quello della variabile puntata, perché `*p` è il "nome" della variabile puntata;
- per ottenere l'indirizzo di una variabile, le si applica come prefisso l'operatore di indirizzo (`&`);

- un puntatore a `void` è un puntatore a un tipo di dato generico e può essere utilizzato per puntare tipi di dato differenti in diversi punti del programma;
- per far sì che un puntatore non punti alcuna variabile gli si assegna come valore la costante `NULL` o l'intero 0;
- poiché il nome di un array è un puntatore al suo primo elemento, si può utilizzare il nome come puntatore per accedere a ogni elemento dell'array in modo sequenziale. Incrementando e decrementando di `n` il puntatore, il suo valore aumenta o diminuisce di `n * T`, dove `T` è la dimensione del tipo del vettore.

### Esercizi

- Aritmetica dei puntatori
- Array di puntatori
- Indirizzi
- Parola riservata `const`
- Parola riservata `null`

- Parola riservata `void`
- Puntatore
- Puntatori e array
- Riferimenti
- Tipi di puntatore

## Esercizi

**1** Se `p` e `q` sono puntatori a struct dire quali istruzioni sono valide ed eventualmente spiegarne l'effetto.

```
struct circuito_elettrico
{
    char corrente[];
    int volt;
};

circuito_elettrico *p, *q;
• p -> corrente = "CA";
• p -> volt = q -> volt;
• *p = *q;
• p = q;
```

- `p -> corrente = "AT";`
- `p -> corrente = q -> volt;`
- `p = 53;`
- `*q = p;`

**2** Sono uguali gli indirizzi di memoria di `r` e `n`? Perché?

```
int main()
{
    int n = 33;
    int &r = n;
    cout << "&n = " << &n << ", &r = " << &r << endl;
```

Soluzioni degli esercizi sul sito web [www.mheducation.it](http://www.mheducation.it)

**Esercizio 1** Qual è l'output del seguente programma?

```
int main()
{
    int n = 35;
    int *p = &n;
    int &r = *p;
    cout << "r = " << r << endl;
}
```

**Esercizio 2** Qual è l'output del seguente programma?

```
int main ()
{
    int n = 35;
    int *p = fn;
    cout << "*p = " << *p << endl;
}
```

**Esercizio 3** Qual è l'output del seguente programma che utilizza una funzione che restituisce un array?

```
float& componente(float *v, int k)
{
    return v[k-1];
}

int main()
{
    float v[5];
    for (int k = 1; k <= 5; k++)
        componente(v,k) = 1.0/k;
    for (int i = 0; i < 5; i++)
        cout << "v[" << i << "] = " << v[i] << endl;
}
```

**Esercizio 4** Trovare gli errori delle seguenti dichiarazioni di puntatori:

```
int x, *p, &x;
char* b= "Stringa lunga";
char* c= 'C';
float z;
void* r = &x;
```

**Esercizio 5** Qual è la differenza fra un puntatore a costante e un puntatore costante?

**Esercizio 6** Un vettore si può indicizzare con l'aritmetica di puntatori. Che tipo di puntatore dovrebbe essere definito per indicizzare una matrice?

**Esercizio 7** Il codice seguente accede agli elementi di una matrice. Accedere agli stessi elementi con aritmetica dei puntatori.

```
double mt[4][5];
for (int f = 0; f < 4; f++)
{
    for (int c = 0; c < 5; c++)
        cout << mt[f][c];
    cout << "\n";
}
```

**Esercizio 8** Scrivere una funzione con un argomento di tipo puntatore a double e un altro argomento di tipo int. Il primo argomento deve corrispondere a un array e il secondo al numero di elementi dell'array. La funzione deve essere di tipo puntatore a double per restituire l'indirizzo dell'elemento minore.

**Esercizio 9** Scrivere un programma che mostri come un puntatore si può utilizzare per scorrere un vettore.

**Esercizio 10** Scrivere una funzione che utilizzi puntatori per copiare un array di dati di tipo double.

**Esercizio 11** Scrivere una funzione che utilizzi puntatori per cercare l'indirizzo di un intero in un array. Se il numero è nel vettore la funzione deve restituire il suo indirizzo, altrimenti deve restituire NULL.

**Esercizio 12** Scrivere una funzione somma mediante puntatori a funzioni che calcoli la somma dei quadrati e dei cubi degli  $n$  primi numeri interi.

**Esercizio 13** Si vogliono immagazzinare le seguenti informazioni di una persona: nome, età, altezza e peso. Scrivere una funzione che legga i dati di una persona, ricevendo come parametro un puntatore e un'altra funzione che li visualizzi.

**Esercizio 14** Scrivere un programma che decida se una matrice di numeri reali è simmetrica. Utilizzare: 1, una funzione di tipo bool che riceva come input una matrice di reali, il suo numero di righe e quello delle colonne, e decida se la matrice è simmetrica; 2, un'altra funzione che generi la matrice di 10 righe e 10 colonne di numeri aleatori da 1 a 100; 3, un programma principale che chiama le due funzioni. **Nota:** passare le matrici come puntatori e utilizzare l'aritmetica dei puntatori.

**Esercizio 1** Scrivere un programma per generare una matrice di  $4 \times 5$  numeri reali, moltiplicare la prima colonna per un'altra qualunque della matrice e mostrare la somma dei prodotti. Il programma deve scomporsi in sottoprogrammi e utilizzare puntatori per accedere agli elementi della matrice.

**Esercizio 2** Scrivere funzioni per sommare a una riga di una matrice un'altra riga, scambiare due righe, e sommare a una riga una combinazione lineare di altre. **Nota:** utilizzare l'aritmetica del puntatori.

**Esercizio 3** Scrivere funzioni che realizzino le seguenti operazioni: moltiplicare una matrice per un numero; riempire di zeri una matrice. **Nota:** utilizzare l'aritmetica di puntatori.

**Esercizio 4** Scrivere un programma nel quale si leggano 20 righe di testo, ognuna con al massimo 80 caratteri. Mostrare su schermo il numero di vocali in ogni riga, il testo letto, e il numero totale di vocali lette.

**Esercizio 5** A una gara di nuoto si iscrivono 16 nuotatori. Concepire un vettore che contenga il nome di ciascuno, l'età e il tempo (minuti, secondi) ottenuto in vasca. Scrivere un programma che consenta l'immersione dei dati nel vettore e visualizzi la graduatoria finale.

**Esercizio 6** Scrivere un programma che permetta di calcolare l'area di diverse figure: un triangolo rettangolo, un triangolo isoscele, un quadrato, un trapezio e un cerchio. **Nota:** utilizzare un array di puntatori di funzioni (che calcolano l'area di ciascuna figura).

**Esercizio 7** Sviluppare un programma in C++ che usi una struttura per raccogliere informazioni su pazienti d'ospedale: nome, indirizzo, data di nascita, genere, giorno di ingresso, giorno di uscita e problema medico. Il programma deve avere una funzione per immettere i dati, raccoglierli in un array e visualizzare in output i pazienti ricoverati in un dato giorno.

**Esercizio 8** Scrivere una funzione che prenda in input una stringa di cifre decimali e restituisca il corrispondente numero intero.

**Esercizio 9** Scrivere un programma per simulare una piccola calcolatrice che permetta di leggere espressioni intere terminanti nel simbolo = e le valuti da sinistra a destra senza tenere in conto la priorità degli operatori. Per esempio,  $4*5-8=12$ . **Nota:** usare un array di funzioni per compiere le diverse operazioni (somma, sottrazione, prodotto, quoziente, resto).

# Allocazione dinamica della memoria

8

- |            |                                 |            |                                      |
|------------|---------------------------------|------------|--------------------------------------|
| <b>8.1</b> | Gestione dinamica della memoria | <b>8.4</b> | Esempi di new e delete               |
| <b>8.2</b> | L'operatore new                 | <b>8.5</b> | Gestione dell'overflow della memoria |
| <b>8.3</b> | L'operatore delete              | <b>8.6</b> | Tipi di memoria in C++               |

## INTRODUZIONE

Abbiamo visto nei precedenti capitoli che le variabili possono essere globali o locali. Le globali occupano posizioni fisse di memoria all'interno del *segmento dati* assegnato al programma dal sistema operativo, e tutte le funzioni possono utilizzarle. Le variabili locali a una funzione vengono invece memorizzate nello *stack* ed esistono *soltanto* mentre è in esecuzione la funzione. È possibile inoltre creare variabili static, che occupano anch'esse posizioni fisse di memoria nel segmento dati, ma sono utilizzabili solo nel modulo (ovvero, nel file sorgente) o nella funzione in cui sono definite.

Queste classi di variabili condividono una caratteristica, quella di

essere definite al momento della compilazione, e questo può essere anche un loro limite. Infatti non sempre è possibile sapere prima dell'esecuzione quanta memoria (cioè quante variabili) saranno necessarie al programma.

In C è possibile assegnare dinamicamente memoria durante l'esecuzione del programma mediante le funzioni `malloc()` e `free()`. In questo capitolo vedremo che il C++ offre un metodo migliore e più efficiente. Invece che chiamare una funzione per assegnare o liberare memoria, C++ invoca due operatori: `new()` e `delete()`, che assegnano e liberano la memoria della zona denominata *heap*.

## 8.1 Gestione dinamica della memoria

Con il termine *heap* si denota la parte della memoria principale che può essere allocata dinamicamente durante l'esecuzione del programma. Per poter essere riallocata, la memoria allocata a "run-time" può essere deallocated automaticamente mediante un *garbage collector* (come avviene, per esempio, in Java) oppure rimanere occupata fino a che il programmatore la libera esplicitamente mediante opportune istruzioni. In C la gestione della memo-

ria dinamica avviene tramite le funzioni `malloc`, `calloc` e `free`. In C++ esiste un'alternativa migliore, gli operatori `new` e `delete`. Con l'operatore `new` si definiscono variabili a run-time e con l'operatore `delete` si possono rilasciare le posizioni occupate dinamicamente perché possano eventualmente essere riutilizzate da altre variabili.

Per esempio, consideriamo un programma per la valutazione degli studenti di un corso. I voti riportati dagli studenti devono essere memorizzati in un array sufficientemente grande da poterli contenere tutti. L'istruzione

```
int corso [40];
```

riserva, per esempio, 40 interi. Questo perché supponiamo che gli studenti siano non più di 40, ma se il loro numero aumentasse, si dovrebbe cambiare la lunghezza dell'array e ricompilare il programma. In effetti, gli array sono la struttura dati più efficiente *quando si conosce la loro lunghezza al momento di scrivere il programma*, ma cosa succede se la dimensione del vettore può essere nota *solo* in fase di esecuzione?

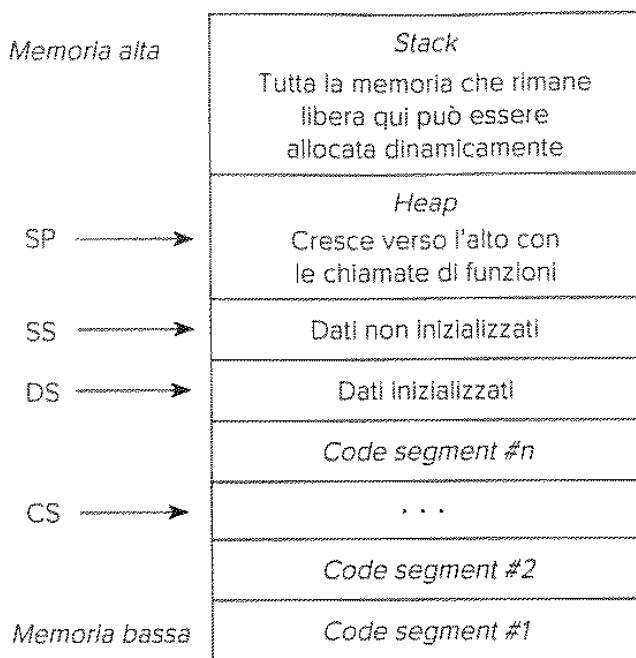
È abbastanza frequente non conoscere la quantità di memoria necessaria fino al momento dell'esecuzione del programma. Per esempio, se si vuole memorizzare una stringa di caratteri liberamente digitata dall'utente, non si può prevedere *a priori* la dimensione dell'array necessario per ospitarla, bisogna quindi preallocare un vettore molto grande con conseguente probabile spreco di memoria. Per risolvere questo problema si deve ricorrere ai *puntatori* e alle tecniche di *allocazione dinamica della memoria*.

Il sistema operativo assegna al programma tre specifici segmenti della memoria principale: il *code segment* che ospita il codice stesso; il *data segment* che ospita costanti e variabili globali; lo *stack* che, a run-time, conterrà le variabili locali alle funzioni, i parametri delle funzioni e le copie dei registri per restituire il controllo alle funzioni chiamanti al termine delle funzioni chiamate. Quindi la memoria allocata in questo segmento cresce e decresce continuamente durante l'esecuzione del programma. La memoria che rimane nello stack, denominata *heap*, viene appunto usata per allocare dinamicamente variabili, e anche questa parte dello stack, lo heap, cresce e decresce dinamicamente, semplicemente lo fa dalla parte opposta.

 Il fatto che questo programma non venga compilato

```
int dimensioneglobale;
char vettore[dimensioneglobale]; // ERRORE! SPECIFICARE LA DIMENSIONE
int main()
{
    int dimensionelocale;
    char vettore[dimensionelocale];// OK! ALLOCATO NELLO STACK A RUN-TIME
}
```

mostra che per allocare un vettore nel segmento dati bisogna conoscerne la dimensione già in fase di compilazione, mentre per allocarlo (a run-time) nello stack no.

**Figura 8.1**

Mappa di memoria di un programma.

La Figura 8.1 mostra la generica mappa di memoria di un programma. L'heap è dentro lo stack e cresce verso l'alto, mentre lo stack cresce verso il basso.

Gli operatori `new` e `delete` sono più evoluti e affidabili delle `malloc()` e `free()` del C perché allocano (o deallozano) memoria in funzione del tipo di dato da memorizzare ed effettuano ogni volta i relativi controlli. Inoltre `new` e `delete` si implementano come operatori e non come funzioni, e si possono utilizzare senza includere alcun header file. Un'altra loro caratteristica importante è che, essendo fortemente sovraccaricati, non richiedono *casting* di tipi e ciò li rende più facili da utilizzare che `malloc()` e `free()`.

## 8.2 L'operatore `new`

L'operatore `new` genera dinamicamente una variabile di un certo tipo assegnandole un blocco di memoria della dimensione di quel tipo. L'operatore restituisce poi l'indirizzo del blocco di memoria allocato, cioè della variabile, che verrà assegnata a un puntatore a quel tipo. La variabile dinamica sarà quindi accessibile dereferenziando il puntatore.

La sintassi dell'operatore `new` è:

`tipo* puntatore = new tipo`

dove `puntatore` sta per il nome del puntatore a cui si assegna l'indirizzo dell'oggetto generato. Per esempio:

```
char* p = new char;
```

Si può anche assegnare l'indirizzo di memoria di un blocco sufficientemente grande per contenere un array di elementi dello stesso *tipo* con questa sintassi:

```
tipo* puntatore = new tipo[dimensione]
```

Per esempio:

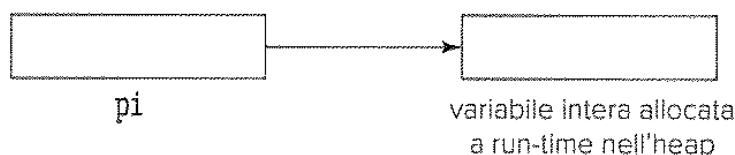
```
char* p = new char[100];
```

genera dinamicamente un vettore di cento char, l'indirizzo del primo dei quali viene assegnato al puntatore p. Si noti che l'operatore new può effettuare questo assegnamento proprio perché esso restituisce l'indirizzo del blocco di memoria allocata dinamicamente.

Ogni volta che si invoca l'operatore new, il compilatore verifica che il tipo puntato dal puntatore (a sinistra) corrisponda al tipo della variabile allocata (a destra). Se i tipi non coincidono, il compilatore produce un messaggio d'errore. Il puntatore potrebbe essere già stato definito, per esempio si può scrivere anche:

```
int* pi;  
...  
pi = new int;
```

L'effetto è sempre quello di creare una variabile intera senza nome, accessibile solo dereferenziando il puntatore pi.



Così per allocare dinamicamente un array di 100 interi si può anche scrivere:

```
int* BloccoMem;  
BloccoMem = new int[100];
```

Se nello heap esiste un blocco libero della dimensione richiesta (400 byte in questo caso) new restituisce l'indirizzo del primo byte del blocco, altrimenti new restituisce 0 o NULL.

La dimensione del blocco di memoria da allocare si può definire a runtime, per esempio richiedendolo dall'input:

```
int n;  
cin >> n;  
char* s = new char[n];
```

#### Esempio 8.1

```
#include <cstring> // per strlen() e strcpy()  
int main()  
{  
    char str[] = "Sierras di Cazorla, Segura e Mágina";
```

```

int lun = strlen(str);
char* ptr = new char[lun+1]; // un byte in più per '\0'
strcpy(ptr, str);
cout << "ptr = " << ptr;
delete ptr;
}

```

### Esecuzione

`ptr = Sierras di Cazorla, Segura e Magina`

Si può invocare l'operatore `new` per allocare un vettore, perfino se non si sa in anticipo quanta memoria richiedono i suoi elementi. Tutto quello che si deve fare è invocare l'operatore `new` utilizzando un puntatore all'array. Anche se in fase di compilazione non si può calcolare la quantità di memoria necessaria, `new` aspetta il momento dell'esecuzione, quando sarà nota la dimensione del tipo del vettore. Per esempio, questo segmento di codice assegna memoria per un array di dieci stringhe la cui lunghezza sarà nota solo al momento dell'esecuzione:

```
miaStringa* mioTesto = new miaStringa[10];
```

Si può inizializzare la variabile dinamica indicandone il valore fra parentesi tonde alla fine dell'istruzione che invoca l'operatore `new`. Per esempio:

```
int* pi = new int(1000);
```

equivale a:

```
int* pi = new int;
*pi = 1000;
```

Per allocare dinamicamente un array multidimensionale si indica ogni dimensione dell'array. Per esempio, per assegnare un puntatore a un array contenente i nomi dei dodici mesi si può allocare dinamicamente una matrice di caratteri di dimensione [12] per [10]:

```
char* p = new char[12][10];
```

Quando si utilizza `new` per assegnare un array multidimensionale, solo la dimensione più a sinistra può essere una variabile. Ciascuna delle altre dimensioni deve essere un valore costante, esattamente come quando si definisce un parametro formale come array dimensionale.

### 8.3 L'operatore `delete`

L'operatore `delete` libera la memoria allocata dinamicamente perché possa eventualmente essere riallocata mediante successive chiamate all'operatore `new`. La sintassi dell'operatore `delete` è:

```
delete puntatore          // non array
delete [] puntatore       // per array
```

Per esempio, lo spazio assegnato per le variabili dinamiche:

```
int* ad = new int;
char* adc = new char[100];
```

si può liberare con le istruzioni:

```
delete ad;
delete [] adc;
```

`delete` rende riutilizzabile la memoria puntata, ma non cancella il puntatore che può quindi essere riutilizzato, per esempio per puntare un'altra variabile successivamente allocata con `new`. Se una variabile allocata dinamicamente non serve più è bene liberare lo spazio che essa occupa nell'heap perché altrimenti questo potrebbe esaurirsi anzitempo; l'operazione viene detta *garbage collection* e in C/C++ è totalmente lasciata alla responsabilità del programmatore. È importante cioè che esista sempre una corrispondenza `new-delete`.

Non si può utilizzare `delete` per liberare memoria occupata da variabili ordinarie.

```
int* p = new int;
delete p;           // corretto
delete p;           // non corretto perché duplicato
int num = 10;
int* pn = &num;
delete pn;          // non corretto perché memoria non assegnata da new
```

 Non si possono creare due puntatori allo stesso blocco di memoria.

#### 8.4 Esempi di `new` e `delete`

Si può allocare dinamicamente qualunque tipo di dato. L'operatore `new` non è costretto a predeterminare la dimensione dei vettori. Utilizzando una variabile si può dimensionare l'array a tempo d'esecuzione. I programmi seguenti mostrano come assegnare una quantità variabile di memoria, a seconda del bisogno.

##### Esempio 8.2

```
#include <cstring>
int main()
{
    int lunghezza_stringa;
    char *p;
    cout << "Quanti caratteri si assegnano? ";
    cin >> lunghezza_stringa;
    p = new char[lunghezza_stringa];
    strcpy(p, "Carchel tambien esta en Sierra Magina");
    cout << p << endl;
```

```
    delete p;
}
```

### Esecuzione

Quanti caratteri si assegnano? 12  
 Carchel tambien esta en Sierra Magina

### Esempio 8.3

```
#include <cstring>
int main()
{
    char str[] = "Mi pueblo es Carchelejo en Jaen";
    int lung = strlen(str);
    char* p = new char[lung+1];
    strcpy (p, str);
    cout << "p = " << p << endl;
    delete [] p;
    return 0;
}
```

### Esecuzione

p = Mi pueblo es Carchelejo en Jaen

L'esempio seguente alloca dinamicamente memoria per una struttura, riempie i suoi campi, visualizza e successivamente libera la memoria allocata.

### Esempio 8.4

```
struct scheda {
    int numero;
    char nome[30];
};

int main ()
{
    scheda* punt_scheda = new scheda;
    cout << "Introduca il numero del cliente: " ;
    cin >> punt_scheda -> numero;
    cout << "Introduca il nome: ";
    cin >> punt_scheda -> nome;
    cout << "Numero: " << punt_scheda -> numero;
    cout << "\nNome: " << punt_scheda -> nome;
    delete punt_scheda;
}
```

### Esecuzione

Introduca il numero del cliente: 12

```
Introduca il nome: Aldo
Numero: 12
Nome: Aldo
```

Il seguente programma mostra come si possa utilizzare l'operatore `new` per allocare dinamicamente un array.

#### Esempio 8.5

```
int main()
{
    int* data = new int[3];
    data[0] = 15;
    data[1] = 8;
    data[2] = 1999;
    cout << "L'appuntamento è il giorno "
        << data[0] << "/"
        << data[1] << "/"
        << data[2];
    delete [] data;
    return 0;
}
```

#### Esecuzione

```
L'appuntamento è il giorno 15/ 8/ 1999
```

Il programma seguente chiede all'utente di digitare la dimensione dell'array allocato dinamicamente.

#### Esempio 8.6

```
int main()
{
    cout << "Quanti elementi ha il vettore? ";
    int lun;
    cin >> lun;
    int* mioArray = new int[lun];
    for (int n = 0; n < lun; n++) mioArray[n] = n + 1;
    for (int n = 0; n < lun; n++) cout << ' ' << mioArray[n];
    delete [] mioArray;
    return 0;
}
```

#### Esecuzione

```
Quanti elementi ha il vettore? 27
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

Il seguente programma definisce un tipo di struttura cane, poi utilizza l'operatore `new` per allocare un array di tre strutture di tipo cane e ne assegna l'in-

dirizzo al puntatore pcane. Il programma assegna valori ai campi di ogni elemento utilizzando la funzione di libreria strcpy() per copiare costanti stringa nei campi razza e colore della struttura cane. Da ultimo, il programma visualizza il contenuto dei tre elementi dell'array.

#### Esempio 8.7

```
#include <cstring>

struct cane {
    char razza[20];
    int eta;
    int altezza;
    char colore[15];
};

int main()
{
    cane* pcane = new cane[3];
    strcpy(pcane[0].razza, "Pastore tedesco");
    strcpy(pcane[0].colore, "Biondo");
    pcane[0].eta = 4;
    pcane[0].altezza = 120;
    strcpy(pcane[1].razza, "dalmata");
    strcpy(pcane[1].colore, "bianco e nero");
    pcane[1].eta = 6;
    pcane[1].altezza = 130;
    strcpy(pcane[2].razza, "doberman");
    strcpy(pcane[2].colore, "nero");
    pcane[2].eta = 4;
    pcane[2].altezza = 155;
    for (int i = 0; i < 3; i++)
    {
        cout << "\nRazza: " << pcane[i].razza << endl;
        cout << "Colore: " << pcane[i].colore << endl;
        cout << "Altezza: " << pcane[i].altezza << endl;
        cout << "Eta: " << pcane[i].eta << endl;
    }
}
```

#### Esecuzione

Razza: Pastore tedesco

Colore: Biondo

Altezza: 120

Eta: 4

Razza: dalmata

Colore: bianco e nero

Altezza: 130

Eta: 6

```
Razza: doberman
Colore: nero
Altezza: 158
Eta: 4
```

## 8.5 Gestione dell'overflow della memoria

Per far fronte alla mancanza di spazio nello heap si può utilizzare la funzione C++ `set_new_handler()` che serve per gestire gli errori. Questa funzione è definita nell'header file `<new>`, e ha come argomento un puntatore a funzione. Quando la si invoca le si passa il nome di una funzione scritta per gestire l'errore.

Per ridurre la probabilità del riempimento dello heap si deve utilizzare accuratamente `delete` per liberare memoria assegnata dinamicamente non più utilizzata.

### Esempio 8.8

```
#include <new>           // set_new_handler

void overflow()
{
    cout << "Memoria insufficiente - fermare esecuzione" << endl;
    exit(1);
}

int main()
{
    set_new_handler(overflow);
    long dimensione;
    cout << "Dimensione dei blocchi da allocare? ";
    cin >> dimensione;
    for (int nblocco = 1; ; nblocco++)
    {
        int* pi = new int[dimensione];
        cout << "Allocazione blocco numero: " << pi << endl;
    }
}
```

### Esecuzione

```
Allocazione blocco numero: 0x7fed1235c010
...
Allocazione blocco numero: 0x7ff83f3bd010
Memoria insufficiente - fermare esecuzione
```

Il programma seguente assegna dinamicamente un array con dimensione richiesta dall'utente; accetta valori dallo standard input, li visualizza a schermo e ne calcola la media aritmetica.

**Esempio 8.9**

```

int main()
{
    int n, somma = 0;
    cout << "Introdurre numero di elementi: ";
    cin >> n;
    int* p = new int[n];
    for (int i=0; i<n; i++)
    {
        cout << "Introduca elemento " << i << ' ';
        cin >> p[i];
        somma += p[i];
    }
    cout << "elementi introdotti: ";
    for (int i = 0; i < n; i++) cout << p[i] << " , ";
    cout << endl;
    cout << "Totale: " << somma << endl;
    cout << "Media: " << (double) somma /n << endl;
    delete [] p;
    return 0;
}

```

**Esecuzione**

```

Introdurre numero di elementi: 4
Introduca elemento 0 34
Introduca elemento 1 56
Introduca elemento 2 78
Introduca elemento 3 65
elementi introdotti: 34 , 56 , 78 , 65 ,
Totale: 233
Media: 58.25

```

**8.6 Tipi di memoria in C++**

In conclusione di questo capitolo ribadiamo le differenze fra i tre tipi di memoria del C++: *automatica*, *statica* e *dinamica*.

**Memoria automatica**

Le variabili definite all'interno di una funzione si denominano *variabili automatiche*: si allocano automaticamente nello stack quando viene invocata la funzione in cui sono definite e si cancellano quando essa termina. Tali variabili sono ovviamente locali alla funzione in cui sono definite, ma, di più, sono locali al blocco che le contiene, cioè alla più piccola sezione di codice rinchiusa tra parentesi graffe in cui sono definite.

### Memoria statica

L'allocazione statica avviene nel *data segment* durante l'esecuzione di un programma completo. Ci sono due modi per allocare staticamente una variabile:

1. definirla al di fuori da qualsiasi funzione, `main()` compresa;
2. anteporre alla sua definizione la parola riservata `static`.

```
static double temperatura = 25.75;
```

### Memoria dinamica

Viene allocata a run-time nello heap dagli operatori `new` e `delete`. Non avendo C/C++ un *garbage collector*, come quelli di Java e Visual Basic, la memoria rimane allocata fino a che viene affrancata dall'esecuzione di un'opportuna istruzione di `delete`.

L'operatore `new` compie una chiamata al sistema operativo chiedendo una certa quantità di memoria. Il sistema risponde verificandone la disponibilità. Negli odierni elaboratori è raro aver problemi di disponibilità di memoria, ma l'esaurimento dello heap è sempre possibile.

#### CONOSCENZE

L'assegnamento dinamico della memoria permette di utilizzare quantità di memoria non note a priori. Abbiamo visto che in C++ si può allocare una variabile nell'heap quando se ne ha bisogno ed eliminarla quando non serve più mediante due operatori: `new` e `delete`. L'operatore `new` assegna un blocco di memoria della dimensione

del tipo specificato. Quando si termina di utilizzare un blocco di memoria definito dinamicamente con `new`, lo si può liberare con l'operatore `delete` perché possa essere riallocato ad altre variabili dinamiche. Per gestire l'eventualità dello svuotamento dello heap si può utilizzare la funzione di libreria `set_new_handler()`.

#### CONOSCENZE

- Array dinamico
- Array statico
- Overflow di memoria
- Stile C (malloc)
- Gestione dinamica
- Operatore `delete`
- Operatore `new`

## Esercizi

Un programma contiene il seguente codice per creare un array dinamico:

```
int* entrata = new int[10];
```

Se non esiste sufficiente memoria nell'heap, la chiamata fallirà. Scrivere un codice che verifichi questa eventualità e visualizzi un relativo messaggio d'errore.

Con riferimento all'esercizio precedente, scrivere il codice per riempire questo array dinamico con 10 numeri letti in input.

Con riferimento all'esercizio precedente, scrivere il codice per sommare gli elementi del vettore.

Data la seguente dichiarazione, definire un puntatore b alla struttura, riservare memoria dinamicamente per una struttura assegnando il suo indirizzo a b.

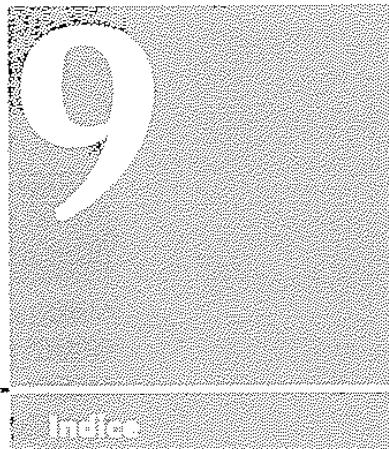
```
struct bottone  
{  
    char* cartellino;
```

```
    int codice;  
};
```

Una volta assegnata memoria al puntatore b dell'Esercizio 8.4 scrivere istruzioni per leggere i campi cartellino e codice.

Dichiarare una struttura per rappresentare un punto nello spazio tridimensionale con un nome. Dichiarare un puntatore alla struttura che abbia l'indirizzo di un array dinamico di  $n$  strutture punto. Assegnare memoria all'array e verificare che si è potuto assegnare la memoria richiesta.

# Stringhe



Indice

- 9.1 Concetto di stringa
- 9.2 Lettura di stringhe
- 9.3 Array e stringhe come parametri di funzioni

- 9.4 La libreria *cstring*
- 9.5 Conversione di stringhe a numeri

## Introduzione

In questo capitolo approfondiremo la conoscenza delle stringhe già introdotte nel capitolo sugli array. Il C++ fornisce due rappresentazioni delle stringhe: il convenzionale vettore di caratteri stile C e la classe *string* introdotta nel C++ standard. Ci sono situazioni in cui è utile o necessario comprendere e utilizzare la vecchia C-string, cioè un array di *char* in cui la sequenza di caratteri dalla posizione 0 fino al primo carattere ASCII nullo (cioè lo '\0')

rappresenta la stringa. Vedremo quindi:

- input e output di stringhe;
- uso delle funzioni di stringa della libreria standard;
- assegnamento di stringhe;
- operazioni varie di stringa (lunghezza, concatenazione, confronto e conversione);
- localizzazione di caratteri e sottostringhe;
- inversione dei caratteri di una stringa.

## 9.1 Concetto di stringa

Una *stringa letterale* è una sequenza di caratteri, per esempio "ABC". Una *C-string*, che qui chiameremo semplicemente *stringa*, è un array di *char* che contiene un carattere *nullo* ('\0'); la sequenza iniziale di caratteri dalla posizione 0 fino al carattere \0 rappresenta una stringa (Figura 9.1).

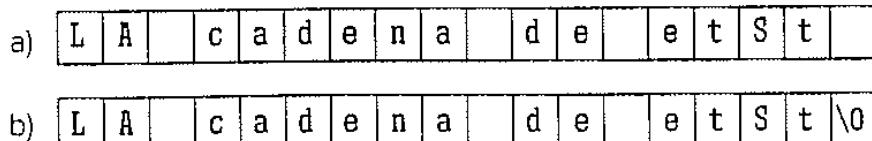


Figura 9.1

a) Array di caratteri; b) stringa.

Una stringa letterale di  $n$  caratteri è un vettore di char lungo esattamente  $n+1$ . Per esempio, la stringa letterale "ABC" è un vettore di 4 char: 'A', 'B', 'C' e '\0'. La stringa letterale è il nome del vettore, è quindi un puntatore a char il cui valore è l'indirizzo del suo primo carattere. Dereferenziando una stringa letterale si ottiene quindi il suo primo carattere (cioè il valore del primo elemento del vettore) ed è possibile anche utilizzare l'aritmetica dei puntatori:

*"ABC"	<i>è il valore</i>	'A'
*("ABC" + 1)	<i>è il valore</i>	'B'
*("ABC" + 2)	<i>è il valore</i>	'C'
*("ABC" + 3)	<i>è il valore</i>	'\0'

allo stesso modo, utilizzando l'indice dell'array si può scrivere:

"ABC"[0]	<i>è il valore</i>	'A'
"ABC"[1]	<i>è il valore</i>	'B'
"ABC"[2]	<i>è il valore</i>	'C'
"ABC"[3]	<i>è il valore</i>	'\0'

#### Esempio 9.1

1. `char str[] = "Clifford";`  
str ha nove caratteri: 'C', 'l', 'i', 'f', 'f', 'o', 'r', 'd' e '\0'
2. `cout << str;`  
il sistema invierà i caratteri di str a cout fino al carattere '\0'.
3. `cin >> buffer;`  
se buffer è un vettore di char abbastanza grande per contenere tutti, il sistema estrarrà caratteri da cin, dal primo non-“spazio” fino al primo “spazio” esclusi, per metterli nel buffer chiudendo automaticamente con il carattere '\0'.

#### 9.1.1 Dichiarazione di variabili stringhe

Le stringhe si allocano in array di char, o `unsigned char`:

```
char testo[80];
char comando[40];
unsigned char dati[10]; // attivazione del bit più significativo
```

Il tipo `unsigned char` può essere interessante nei casi in cui possono essere presenti caratteri speciali con l'ottavo bit settato a 1.

La dimensione del vettore deve essere tale da includere il carattere '\0'. Perciò, un array che dovrà contenere la stringa "ABCDEF" sarà definito come:

```
char UnaStringa[7];
```

#### 9.1.2 Inizializzazione di variabili stringhe

Essendo le stringhe usate assai frequentemente, per loro si utilizzano forme di inizializzazione semplificate:

```
char UnaStringa[7] = "ABCDEF";
char testo[81] = "Questa è una stringa";
char testodemo[255] = "Questa è una stringa opù lunga";
char stringatest[] = "Quale è la lunghezza di questa stringa?";
```

Le stringhe testo e testodemo possono contenere fino a 80 e 254 caratteri, rispettivamente, più il carattere nullo. Si noti la quarta stringa, stringatest, che non specifica la dimensione per farla definire automaticamente dal compilatore; siccome nella stringa letterale a destra dell'operatore di assegnamento vi sono 39 caratteri, il compilatore aggiunge il carattere '\0' e alloca 40 byte per stringatest.

Come ogni vettore, una stringa non può essere assegnata fuori dalla sua definizione. Per esempio, l'assegnamento:

```
UnaStringa = "ABC";
```

darà errore di compilazione, perché un identificatore di stringa, come qualunque identificatore di array, può prendere come valore un indirizzo, e non una stringa. Vedremo più avanti che per assegnare una variabile stringa sarà necessario utilizzare la funzione di libreria denominata strcpy().

### Esempio 9.2

Il seguente programma mostra che l'inizializzazione di una stringa aggiunge automaticamente il carattere '\0':

```
int main()
{
    char S[] = "ABCD";
    for (int i = 0; i < 5; i++)
        cout << "S[" << i << "] = '" << S[i] << "'\n";
}
```

### Esecuzione

```
S[0] = 'A'
S[1] = 'B'
S[2] = 'C'
S[3] = 'D'
S[4] = ''
```

## 9.2 Lettura di stringhe

L'operatore di estrazione >> applicato a un flusso come lo standard input cin, se deve mettere i caratteri letti in un vettore di caratteri si comporta in maniera speciale:

```
int main()
{
    char nome[30];
    cin >> nome; // legge caratteri e li mette dentro l'array
```

```

cout << '\n' << nome; // visualizza tutto il contenuto dell'array
}

```

Il programma definisce `nome` come un array di 30 char. Se l'utente batte da tastiera Peppe Mackoy, il programma visualizzerà su schermo soltanto Peppe. Cioè, la parola Mackoy non viene assegnata alla variabile stringa `nome`. Il motivo è che l'operatore `>>` termina la lettura quando trova uno spazio. Per leggere una sequenza di caratteri che comprende spazi bianchi si deve utilizzare la funzione di libreria `getline()` invece che l'operatore `>>`. Vedremo fra poco che la funzione `getline` permette a `cin` di leggere la stringa completa, includendo qualsiasi spazio in bianco. Nel prossimo esempio il ciclo `do while` dipende dal numero di parole introdotte fino a quando l'utente chiude il flusso di input.

### Esempio 9.3

```

int main()
{
    char parola[80];
    do {
        cin >> parola;
        if (*parola) cout << "\t\" " << parola << "\"\n";
    } while (*parola);
}

```

### Esecuzione di prova

Oggi è 16 Giugno 2007.

```

" Oggi"
" è"
" 16"
" Giugno"
" 2007."

```

Domani è Domenica.

```

" Domani"
" è"
" Domenica."

```

Ogni parola dal flusso di input `cin` viene immessa nel flusso di output `cout`. Il buffer di input non viene letto finché non viene battuto il carattere *Invio* (carriage return \r). Se `parola` contiene una stringa di lunghezza maggiore di 0, `*parola` è diverso da zero (e quindi è true), infatti solo in questo caso il primo elemento della stringa è il carattere '\0'. Premendo CTRL-C (nei sistemi UNIX/Linux) o CTRL-Z (in Windows) si invia il carattere di terminazione del flusso. Questa azione carica la stringa vuota in `parola` fermando appunto il ciclo.

Come capiremo meglio quando introdurremo la programmazione orientata agli oggetti, l'oggetto `cin` (flusso di input) include le funzioni di input: `cin.getline()`, `cin.get()`, `cin.ignore()`, `cin.putback()` e `cin.peek()`.

### 9.2.4 cin.getline()

cin è un oggetto della classe iostream, e getline() è una funzione membro della classe iostream; quindi cin può chiamare getline() per leggere una riga completa (includendo qualunque carattere di spaziatura) con il seguente formato:

```
cin.getline(var_str, max_lung_stringa, 'terminatore');
```

*var\_str* è l'identificatore della variabile stringa, *max\_lung\_stringa* è la lunghezza massima della stringa (che deve essere almeno due caratteri più lunga della stringa vera e propria per includere il carattere nullo '\0' e quello di fine linea '\n'), e '*terminatore*' è il carattere con il quale la lettura termina. Questo significa che la lettura termina dopo aver letto al massimo *max\_lung\_stringa* caratteri, ma può terminare prima non appena s'incontra il carattere '*terminatore*'. La funzione è sovraccaricata e si può lanciare anche con due argomenti (che in realtà è il caso più frequente).

```
cin.getline(var_str, max_lung_stringa);
```

con la quale s'intende che il carattere terminatore è '\n'.

Dopo aver depositato i caratteri letti nel vettore passato al primo argomento, la funzione getline() inserisce automaticamente anche il carattere nullo come terminatore di stringa.

### Esempio 9.4

```
int main()
{
    char Nome[32];
    char Via[32];
    char Citta[27];
    char Provincia[27];
    char CodicePostale[6];
    char Telefono[11];

    cin.getline(Nome, 31);
    cin.getline(Via, 31);
    cin.getline(Citta, 26);
    cin.getline(Provincia, 26);
    cin.getline(CodicePostale, 5);
    cin.getline(Telefono, 10);

    cout << Nome << endl;
    cout << Via << endl;
    cout << Citta << endl;
    cout << Provincia << endl;
    cout << CodicePostale << endl;
    cout << Telefono;
}
```

### Esecuzione di prova

```
Aldo Pietropaolo
Martiri Resistenza
Ancona
AN
6011
07362890
Aldo Pietropaolo
Martiri Resistenza
Ancona
AN
6011
07362890
```



- La chiamata `cin.getline(str, n, car)` legge tutti i primi `n` caratteri da `cin` fino alla prima occorrenza del carattere separatore `car` e li inserisce in `str`.
- Se il carattere specificato `car` è il *newline* '\n', la chiamata è equivalente a `cin.getline(str, n)`.

### 9.2.2 cin.get()

La funzione `get()` legge dal flusso carattere per carattere. La chiamata `cin.get(car)` scrive il successivo carattere proveniente dal flusso di input `cin` nella variabile `car` e restituisce 1, a meno che non venga letta la fine del flusso, caso in cui la `get()` restituisce 0.

#### Esempio 9.5

```
int main()
{
    char car;
    while (cin.get(car))
        if (car == 'a') cout << 'o';
        else cout << car;
    return 0;
}
```

### Esecuzione di prova

```
Garibaldi fu ferito
Goribaldi fu ferito
fu ferito ad una gamba
fu ferito od uno gombo
Garibaldi che comanda
Goribaldi che comondo
che comanda il battaglion!
che comondo il bottoglion!
^C
```



Il `while` cicla fintanto che la funzione `cin.get(car)` legge caratteri e li mette in `car`, cioè fintanto che l'utente termina il programma con CTRL-C (UNIX/Linux) o CTRL-Z (Windows).

### 9.2.3 cout.put()

La funzione opposta di `get()` è `put()` che scrive nel flusso carattere per carattere.

#### Esempio 9.6

Il seguente programma fa l'eco del flusso di input, mettendo però in maiuscolo l'iniziale di ogni parola ricorrendo alla funzione di libreria `toupper(car)`, che restituisce l'equivalente maiuscola di `car` se questa è una lettera minuscola, altrimenti restituisce la maiuscola stessa.

```
int main()
{
    char car, pre = '\0';
    while (cin.get(car)) {
        if (pre == ' ' || pre == '\n') cout.put(toupper(car));
        else cout.put(car);
        pre = car;
    }
    return 0;
}
```

#### Esecuzione di prova

```
Titolo di un importante Articolo scientifico
Titolo Di Un Importante Articolo Scientifico
ci riprovo un'altra Volta
Ci Riprovo Un'altra Volta
^C
```



In alternativa alla funzione di libreria `toupper()` si poteva anche scrivere: `car+='A'-'a'`. L'algoritmo si basa sul fatto che se il carattere precedente `pre` è uno spazio o il carattere nuova riga, allora il successivo sarà il primo carattere della parola seguente, quindi la lettera in `car` viene rimpiazzata dalla sua equivalente maiuscola. Si noti che nel secondo esempio la parola "altra" non è stata resa con l'iniziale in maiuscolo; si modifichi l'algoritmo in maniera da correggere questo problema.

### 9.2.4 cin.putback(), cin.ignore()

La funzione `putback()` rimette nel flusso di input l'ultimo carattere letto, mentre la funzione `ignore()` lo ignora.

**Esempio 9.7**

Il programma implementa una funzione che riconosce ed estrae gli interi dal flusso di input.

```
int seguenteIntero()
{
    char car;
    int n;
    while (cin.get(car))
        if (car >= '0' && car <= '9') {
            cin.putback(car);
            cin >> n;
            break;
        }
    return n;
}

int main()
{
    int m = seguenteIntero(), n = seguenteIntero();
    cin.ignore(80, '\n');
    cout << m << " + " << n << " = " << m + n << endl;
}
```

**Esecuzione di prova**

Quanto fa 406 più 3121?

406 + 3121 = 3527



La funzione `seguenteIntero()` esplora i caratteri passati in `cin` fino a che si trova la prima cifra, 4 nell'esempio, la rimette nel flusso e mette 4 in `n`, ripetendo per 0, mettendo 40 in `n`, e 6, quando passa finalmente allo spazio e `n` rimane settata a 406. `cin.ignore(80, '\n')` è una variante a due argomenti della `ignore()` (ce ne sono di analoghe anche per le altre funzioni che stiamo trattando, il consiglio è quello di approfondire leggendo i manuali online del linguaggio C++) che estrae continuamente caratteri dal flusso di input e li butta via fino a quando ne ha estratti 80 oppure incontra e butta via un *newline* '\n'.

**9.2.5 `cin.peek()`**

La funzione `peek()` può essere utilizzata al posto della combinazione `get()` e `putback()`. Attenzione, `peek()` non ammette parametri, quindi va chiamata così:

`car = cin.peek()`

Quest'istruzione copia il successivo carattere dal flusso di input `cin` nella variabile `car` (di tipo `int` o `char`) senza eliminare il carattere dal flusso di input.

**Esempio 9.8**

```

int seguenteIntero()
{
    char car;
    int n;
    while ((car = cin.peek()))
        if (car >= '0' && car <= '9')
        {
            cin >> n;
            break;
        }
        else cin.get(car);
    return n;
}

int main()
{
    int m = seguenteIntero(), n = seguenteIntero();
    cin.ignore(80, '\n');
    cout << m << " + " << n << " = " << m + n << endl;
}

```

**Esecuzione di prova**

Quanto farebbe 406 per 3121 se tu fossi un programma fatto bene?

$406 + 3121 = 3527$



L'istruzione `car = cin.peek()` copia il successivo carattere in `car`. Poi, se `car` è una cifra, viene letto l'intero completo in `n` e lo si restituisce, altrimenti si elimina il carattere da `cin` e si continua il ciclo. Se si raggiunge la fine del file, l'espressione `car = cin.peek()` restituisce 0, e il ciclo si arresta.

**9.3 Array e stringhe come parametri di funzioni**

Come tutti gli array, le stringhe possono essere passate alle funzioni fornendo il valore del puntatore al loro primo elemento. Per riferire la stringa la funzione dereferenzierà l'argomento formale. Per esempio, il seguente programma contiene una funzione che copia `num_car` caratteri dalla stringa sorgente alla stringa dest.

**Esempio 9.9**

```

int estrarre(char* dest, char* sorgente, char car)
{
    int i;
    for(i = 1; (*sorgente != '\0') && (*sorgente != car); i++)
        *dest++ = *sorgente++;
}

```

```

*dest = '\0';
return i; // restituisce numero di caratteri letti
}

int main(void)
{
    char c;
    char s1[40] = "SupercalifragilisticEspiralidoso";
    char s2[50];
    cout << "Fino a quale carattere vuole estrarre? ";
    cin >> c;
    cout << "Ho estratto " << estrarre(s2, s1, c) - 1
        << " caratteri, ovvero:\n";
    cout << s2 << endl;
return 0;
}

```

### Esecuzione di prova

\$ Fino a quale carattere vuole estrarre? c  
 Ho estratto 5 caratteri, ovvero:  
 Super

\$ Fino a quale carattere vuole estrarre? s  
 Ho estratto 16 caratteri, ovvero:  
 Supercalifragili

\$ Fino a quale carattere vuole estrarre? w  
 Ho estratto 32 caratteri, ovvero:  
 SupercalifragilisticEspiralidoso



Si osservi che i parametri formali sono dichiarati come puntatori di tipo char. L'istruzione:

`*dest++ = *sorgente++;`

dereferenzia entrambi i puntatori per accedere alle stringhe prese in input.

Potevamo ottenere lo stesso risultato con la notazione vettoriale. Il seguente programma contiene una funzione che calcola la lunghezza della stringa passata alla chiamata. Il parametro formale str è dichiarato come array di caratteri (ma ovviamente di dimensione indefinita perché è semplicemente un puntatore a caratteri).

### Esempio 3.10

```

int Lunghezza(const char str[])
{
    int contatore = 0;
    while (str[contatore] != '\0');

```

```

    return contatore;
}
int main()
{
    cout << Lunghezza("C++ è migliore del C") << endl;
    return 0;
}

```

**Esecuzione**

22

Il programma principale chiama la funzione `Lunghezza(const char str[])` passandole la stringa letterale "C++ è migliore del C" (cioè il valore del puntatore alla prima cella di memoria della stringa letterale, che sarebbe quella che contiene la 'C' iniziale). Il ciclo while dentro la funzione conta i caratteri non nulli e termina quando trova il byte nullo alla fine della stringa. Provare a vedere cosa fa il vostro compilatore se si rimuove lo specificatore `const` dal primo argomento della funzione.

**9.4 La libreria `cstring`**

La libreria `cstring` del C++ contiene funzioni di manipolazione di stringhe. La Tabella 9.1 ne riassume alcune.

Le funzioni di stringa raccolte nella Tabella 9.1 usano la parola riservata `const` per distinguere fra parametri di input e di output.

`strcpy(s1,s2)` copia i caratteri della stringa `s2` nella stringa `s1` aggiungendo il carattere nullo alla fine. La funzione suppone che la stringa destinazione abbia spazio sufficiente per contenere tutta la stringa sorgente. Per esempio:

```

int main()
{
    char s[100] = "Buongiorno Mr. Perry", t[100];
    strcpy(t, s);
    strcpy(t+15, "Dragonì");
    cout << s << endl << t << endl;
}

```

Il programma visualizza:

```
Buongiorno Mr. Perry
Buongiorno Mr. Dragonì
```

L'espressione `t+15` ottiene l'indirizzo della sottostringa di `s` che inizia a partire dall'indice 15.

**Conseguenze**

Si possono manipolare sottostringhe assegnando a puntatori l'indirizzo del loro primo carattere.

Tabella 9.1 Funzioni di &lt;cstring&gt;

Funzione	Intestazione della funzione e prototipo
<b>strcpy()</b>	char* strcpy(char* destinazione, const char* sorgente); Copia la stringa <i>sorgente</i> nella stringa <i>destinazione</i> . Restituisce <i>destinazione</i>
<b>strncpy()</b>	char* strncpy(char* destinazione, const char* sorgente, size_t num); Copia <i>num</i> caratteri da <i>sorgente</i> a <i>destinazione</i> . Restituisce <i>destinazione</i>
<b>strlen()</b>	size_t strlen (const char* s); Restituisce la lunghezza della stringa <i>s</i>
<b>strcat()</b>	char* strcat(char* destinazione, const char* sorgente); Aggiunge una copia della stringa <i>sorgente</i> alla fine di <i>destinazione</i> . Restituisce <i>destinazione</i>
<b>strncat()</b>	char* strncat(char* s1, const char* s2, size_t n); Aggiunge i primi <i>n</i> caratteri di <i>s2</i> a <i>s1</i> . Restituisce <i>s1</i> . Se <i>n</i> >= strlen( <i>s2</i> ), allora strncat( <i>s1</i> , <i>s2</i> , <i>n</i> ) ha lo stesso effetto che strcat( <i>s1</i> , <i>s2</i> )
<b>strcmp()</b>	int strcmp(const char* s1, const char* s2); Confronta le stringa <i>s1</i> e <i>s2</i> e restituisce: 0 se <i>s1</i> = <i>s2</i> <0 se <i>s1</i> < <i>s2</i> >0 se <i>s1</i> > <i>s2</i>
<b>strncmp()</b>	int strncmp(const char* s1, const char* s2, size_t n); Come strcmp() ma solo sui primi <i>n</i> caratteri
<b>strchr()</b>	const char* strchr(const char* str, int c); Restituisce un puntatore alla prima occorrenza del carattere <i>c</i> in <i>s</i> . Restituisce NULL se <i>c</i> non è in <i>s</i>
<b> strrchr()</b>	const char* strrchr(const char* s, int c); Restituisce un puntatore all'ultima occorrenza di <i>c</i> in <i>s</i> . Restituisce NULL se <i>c</i> non è in <i>s</i>
<b>strspn()</b>	size_t strspn(const char* s1, const char* s2); Restituisce la lunghezza della sottostringa più lunga di <i>s1</i> che comincia con <i>s1[0]</i> e contiene unicamente caratteri presenti in <i>s2</i>
<b>strstr()</b>	const char* strstr(const char* s1, const char* s2); Cerca la stringa <i>s2</i> in <i>s1</i> e restituisce un puntatore al carattere dove comincia <i>s2</i>
<b>strcspn()</b>	size_t strcspn(const char* s1, const char* s2); Restituisce la lunghezza della sottostringa più lunga di <i>s1</i> che comincia con <i>s1[0]</i> e non contiene alcuno dei caratteri presenti in <i>s2</i>
<b>strpbrk()</b>	const char* strpbrk(const char* s1, const char* s2); Restituisce l'indirizzo della prima occorrenza in <i>s1</i> di qualunque dei caratteri di <i>s2</i> . Restituisce NULL se nessuno dei caratteri di <i>s2</i> appare in <i>s1</i> .
<b>memcpy()</b>	void* memcpy(void* s1, const void* s2, size_t n); Rimpiazza i primi <i>n</i> byte di * <i>s1</i> con i primi <i>n</i> byte di * <i>s2</i> . Restituisce <i>s1</i>
<b>strnset()</b>	char* strnset(char* s, int ch, size_t n); Utilizza strcmp() su una stringa esistente per fissare <i>n</i> byte della stringa al carattere <i>ch</i>
<b>strstr()</b>	const char* strstr(const char* s1, const char* s2); Cerca la stringa <i>s2</i> in <i>s1</i> e restituisce un puntatore al carattere dove comincia <i>s2</i>
<b>strtok()</b>	char* strtok(char* s1, const char* s2); Divide la stringa <i>s1</i> in sottostringhe delimitate dai caratteri presenti nella stringa <i>s2</i> . Dopo la chiamata iniziale strtok( <i>s1</i> , <i>s2</i> ), ogni chiamata successiva a strtok(NULL, <i>s2</i> ) restituisce un puntatore alla successiva sottostringa in <i>s1</i> . Queste chiamate cambiano la stringa <i>s1</i> , rimpazzando ogni separatore con il carattere NULL ('<\0>')

```

char str1[41] = "Ciao mondo";
char str2[41];
char* p = str1;

p += 5; // p punta alla sottostringa "mondo"
strcpy(str2, p);
cout << str2 << "\n";

```

L'istruzione di output visualizza la stringa "mondo".

**strncpy(s1,s2,n)** copia *n* caratteri dalla stringa s2 alla stringa s1. Per esempio:

```

int main()
{
    char str1[] = "Aldo Franco";
    char str2[] = "Ciao mondo!";
    strncpy(str1, str2, 5);
    cout << str1 << endl << str2 << endl;
}

```

Il programma visualizza:

```

Ciao Franco
Ciao mondo!

```

**strlen()** restituisce il numero di caratteri del parametro stringa, escludendo il carattere nullo di terminazione della stringa. Il tipo di risultato *size\_t* rappresenta un tipo intero generale. Per esempio:

```

int main()
{
    char s[] = "ABCDEFG";
    cout << "strlen(" << s << ") = " << strlen(s) << endl;
    cout << "strlen(\"\\\") = " << strlen("") << endl;
    char buffer[80];
    cout << "Introduca stringa:"; cin >> buffer;
    cout << "strlen(" << buffer << ") = " << strlen(buffer) << endl;
}

```

Il programma visualizza:

```

strlen (ABCDEFG) = 7

```

```

strlen ("") = 0

```

Introduca stringa: Ascoli

```

strlen(Ascoli) = 6

```

**strcat(s1,s2)** appende il contenuto della stringa s2 alla stringa s1, restituendo un puntatore alla stringa s1. Per esempio:

```

int main()
{
    char stringa[30];

```

```

strcpy(stringa, "Ascoli");
strcat(stringa, " Piceno");
cout << stringa << endl;
}

```

Il programma visualizza:

Ascoli Piceno

**strncat(s1,s2,n)** appende alla stringa **s1** un quantitativo massimo **n** di caratteri della stringa **s2** e restituisce il puntatore a **s1**. Per esempio:

```

int main()
{
    char stringa[30];
    strcpy(stringa, "Ascoli");
    strncat(stringa, " Piceno", 3);
    cout << stringa << endl;
}

```

Il programma visualizza:

Ascoli Pi

Le funzioni di confronto lessicografico **strcmp** e **strncmp** confrontano i caratteri di due stringhe utilizzando il valore ASCII di ogni carattere.

**strcmp(s1,s2)** restituisce un valore minore di zero se **s1** è lessicograficamente minore di **s2**, restituisce un valore maggiore di zero nel caso opposto, e infine restituisce 0 se le due stringhe sono identiche. Ricordiamo che le lettere maiuscole sono lessicograficamente minori delle minuscole. Per esempio:

```

int main()
{
    cout << strcmp("Microsoft C++", "Microsoft Visual C++") << endl;
    cout << strcmp("Microsoft Visual Basic", "Microsoft Visual C++") << endl;
    cout << strcmp("Microsoft Visual C++", "Microsoft Visual C++") << endl;
}

```

Il programma visualizza:

-1  
-1  
0

**strcmp(s1,s2,n)** opera lo stesso confronto alfanumerico sui primi **num** caratteri delle due stringhe.

**<string>** contiene anche funzioni per cercare caratteri e sottostringhe.

*Funzioni di ricerca di caratteri*

strchr	strrchr	strspn
strcspn	strupr	

*Funzioni di ricerca di stringhe*

strstr	strtok
--------	--------

`strchr(stringa,car)` localizza la prima occorrenza del carattere car in stringa. Per esempio:

```
int main()
{
    char str[81] = "GNU C++ Compiler";
    cout << strchr(str, 'C') << endl;
}
```

Il programma visualizza:

```
C++ Compiler
```

`strrchr(stringa,car)` localizza l'ultima occorrenza del carattere car in stringa. Per esempio:

```
int main()
{
    char str[81] = "GNU C++ Compiler";
    cout << strrchr(str, 'C') << endl;
}
```

Il programma visualizza:

```
Compiler
```

`strspn(s1,s2)` restituisce la lunghezza della sottostringa *iniziale* di s1 che contiene esclusivamente caratteri appartenenti anche alla stringa s2. Per esempio:

```
int main()
{
    char stringa1[] = "1233456";
    char stringa2[] = "abc123";
    cout << strspn(stringa1, stringa2); // visualizza 4
}
```

Il programma visualizza:

```
4
```

`strcspn(s1,s2)` restituisce l'indice del primo carattere di s1 che si trova anche in s2. Per esempio:

```
int main()
{
    char stringa1[] = "SupercalifragilisticEspiralidoso";
    char stringa2[] = "caratteri da escludere";
    cout << strcspn(stringa1, stringa2);
}
```

Il programma visualizza:

```
1
```

**strpbrk(s1, s2)** restituisce un puntatore alla prima occorrenza di qualsiasi carattere di s2 in s1. Se le due stringhe non hanno caratteri in comune restituisce NULL. Per esempio:

```
int main()
{
    char stringa1[] = "SupercalifragilisticEspiralidoso";
    char stringa2[] = "caratteri candidati a essere il primo";
    cout << strcspn(stringa1, stringa2);
}
```

Il programma visualizza:

percalifragilisticEspiralidoso

**strstr e strtok** permettono di localizzare una sottostringa in una stringa oppure di separare una stringa in sottostringhe.

**strstr(s1, s2)** restituisce un puntatore al primo carattere della stringa s1 da cui inizia la sottostringa s2. Se in s1 non c'è la sottostringa s2, la funzione restituisce NULL. Per esempio:

```
int main()
{
    char stringa1[] = "SupercalifragilisticEspiralidoso";
    char stringa2[] = "fragil";
    cout << strstr(stringa1, stringa2);
}
```

Il programma visualizza:

fragilisticEspiralidoso

**strtok(s1,s2)** permette di estrarre da s1 sottostringhe limitate dai caratteri "separatori" contenuti in s2. Per esempio:

```
int main()
{
    char s1[] = "(Supercali+Fragilistic)=Espiralidoso";
    char s2[] = "+=()";
    cout << s1;
    cout << "\nviene separata da " << s2 << " in\n";
    char *ptr = strtok(s1, s2);
    while (ptr)
    [
        cout << ptr << endl;
        ptr = strtok(NULL, s2);
    ]
    return 0;
}
```

Il programma visualizza:

(Supercali+Fragilistic)=Espiralidoso  
viene separata da +=() in  
Supercali  
Fragilistic  
Espiralidoso

## 9.5 Conversione di stringhe a numeri

Molto spesso bisogna convertire stringhe di cifre nei corrispettivi formati numerici. C++ fornisce per questi scopi le funzioni `atoi`, `atof` e `atol`.

`int atoi(const char* str)` prova a convertire la stringa di cifre str al corrispondente valore int.

`double atof(const char* str)` prova a convertire la stringa di cifre str al corrispondente valore double.

`long atol(const char* str)` prova a convertire la stringa di cifre str al corrispondente valore long.

### Esempio 9.11

```
int main()
{
    char s1[] = "31415926535";
    char s2[] = "3.1415926535";
    char s3[] = "-3.1415926535";
    cout << "STRINGA s\t" << "atoi(s)\t" << "atof(s) \t" << "atol(s)\n";
    cout << "-----\n";
    cout << s1 << '\t' << atoi(s1) << '\t' << atof(s1) << '\t' << atol(s1) << endl;
    cout << s2 << '\t' << atoi(s2) << "\t" << atof(s2) << "\t" << atol(s2) << endl;
    cout << s3 << '\t' << atoi(s3) << "\t\t" << atof(s3) << "\t" << atol(s3) << endl;
}
```

Il programma visualizza:

STRINGA s	atoi(s)	atof(s)	atol(s)
31415926535	1351155463	3.14159e+10	31415926535
3.1415926535	3	3.14159	3
-3.1415926535	-3	-3.14159	-3

### Conclusione

In questo capitolo abbiamo visto quelle che in C++ si definiscono C-string (stringhe "alla C"), ovvero sequenze di caratteri terminate dal carattere nullo ('\0'). Queste vengono allocate nella parte iniziale di un vettore di caratteri. Gli opera-

tori di I/O del C++ sono ben sovraccaricati per trattare in maniera comoda queste stringhe. Inoltre C++ definisce nella libreria `cstring` varie funzioni di gestione delle C-string.

**Parole chiave**

- Assegnamento
- Stringa
- Stringa vuota
- Carattere nullo (`NULL`, `'\0'`)
- Confronto
- Conversione
- Libreria `<cstring>`
- Inversione
- Funzioni di stringa

## Esercizi

**E1** Scrivere il codice di una funzione che si comporti come `strlen`.

**E2** Cos'è che non va nell'istruzione:

```
cin >> s ;
```

pensata per leggere l'input "Ciao, Mondo!" e metterlo nella stringa `s`?

**E3** Qual è la differenza tra le due seguenti istruzioni, se `s1` e `s2` sono del tipo `char*`?

```
s1 = s2;
strcpy(s1, s2) ;
```

**E4** Cosa fa il seguente codice?

```
char *s1 = "ABCDE" ;
char *s2 = "ABC" ;
if(strcmp(s1, s2)<0) cout << s1 << " < " << s2 << endl ;
else cout << s1 << " >= " << s2 << endl ;
```

**E5** Quali delle seguenti dichiarazioni sono equivalenti?

```
char var_str0[10] = "Ciao";
char var_str1[10] = { 'C','i','a','o'};
char var_str2[10]= { 'C','i','a','o','\0'};
char var_str3[5]= "Ciao";
char var_str4[]= "Ciao";
```

**E6** Cosa c'è di scorretto (se c'è) nel seguente codice?

```
char var_str[]= "Ciao";
strcat(var_str, " e addio");
cout << var_str<< endl;
```

**E7** Che differenze e analogie vi sono tra le variabili `c1`, `c2`, `c3`? La dichiarazione è:

```
char** c1;
char* c2[10];
char* c3[10][21];
```

**E8** Scrivere un programma che legga due stringhe di caratteri, le visualizzi insieme alla loro lunghezza, le concatene e visualizzzi la concatenazione e la sua lunghezza.

**E9** Scrivere un programma che legga una stringa di caratteri in Input e la Inverta.

**E10** Scrivere un programma che faccia l'eco dell'input, riga per riga.

**E11** Contare il numero di occorrenze della lettera 'e' nel flusso di input.

**E12** Scrivere un programma che faccia l'eco del flusso di Input e poi metta in maiuscolo la prima lettera di ogni parola.

**E13** Scrivere una funzione che estragga gli interi dal flusso di input.

**E14** Leggere una sequenza di stringhe, immagazzinarle in un array e, poi, visualizzarle.

**E15** Scrivere un programma che faccia uso di una funzione `invertire()` per invertire una stringa letta dalla tastiera.

**E16** Leggere una riga di input. Scartare tutti i simboli eccetto le cifre. Convertire la stringa di cifre in un intero e fissare il valore dell'intero alla variabile `n`.

**E17** Definire un array di stringhe di caratteri per poter leggere un testo costituito da un massimo di 80 caratteri per riga. Scrivere una funzione per leggere il testo,

e un'altra per scriverlo; le funzioni debbono avere due argomenti, uno il testo e il secondo il numero di righe.

Scrivere una funzione che prenda in input una stringa e restituiscala il numero di vocali, di consonanti e di cifre della stringa.

Scrivere un programma che scopra se due stringhe introdotte dalla tastiera siano anagrammi. Si considera che due stringhe sono anagrammi se contengono esattamente gli stessi caratteri nello stesso o in differente ordine. Bisogna ignorare i bianchi e considerare che le maiuscole e le minuscole sono uguali.

Scrivere un programma per le righe di un testo sapendo che il massimo numero di caratteri per riga è 80. Contare il numero di parole che ha ogni riga, nonché il numero totale di parole lette.

Dato un testo formato al massimo da 30 righe, del quale si vuole sapere il numero di occorrenze di una parola chiave, scrivere un programma che legga

la parola chiave e ne determini il numero di occorrenze nel testo.

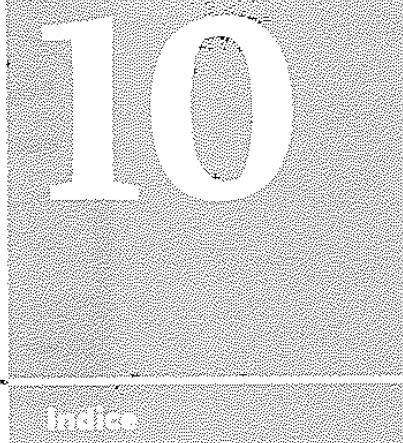
È dato un testo di 40 righe. Le righe hanno un numero di caratteri variabile. Scrivere un programma per mettere il testo in una matrice di righe, aggiustando la lunghezza di ogni riga al numero di caratteri. Il programma deve leggere il testo, immagazzinarlo nella struttura matriciale e scrivere su schermo le righe in ordine crescente di lunghezza.

Scrivere un programma che legga una stringa chiave e un testo di al massimo 50 righe. Il programma deve eliminare le righe che contengono la chiave.

Scrivere una funzione che riceva come parametri un numero grande come stringa di caratteri, e lo moltiplichi per una cifra, che riceva come parametro di tipo carattere.

Scrivere una funzione che moltiplicherà due numeri grandi, ricevuti come stringhe di caratteri.

# Flussi e file: libreria standard di I/O



10.1	Flussi ( <i>stream</i> )	10.6	Indicatori di formato
10.2	La libreria <i>iostream</i>	10.7	I/O da file
10.3	La classe <i>istream</i>	10.8	I/O binario
10.4	La classe <i>ostream</i>	10.9	Accesso diretto
10.5	Formattazione dell'output		

## Introduzione

Lo scambio dati fra il mondo e la CPU si chiama "I/O" (Input/Output). L'input dei dati viene anche detto **lettura** e può avvenire sia dall'esterno del sistema di elaborazione (tastiera, mouse ecc.) sia dal suo interno (memoria di massa). Analogamente l'output dei dati viene detto **scrittura** e può anch'esso avvenire sia verso l'esterno del sistema (scheda video, scheda audio, stampante ecc.) sia

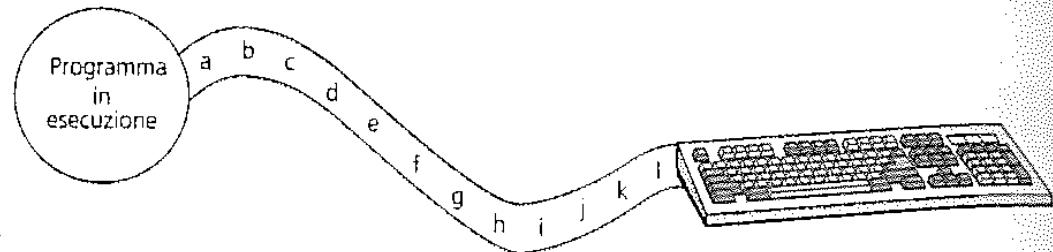
verso l'interno (dischi fissi, memorie USB ecc.).

Le funzionalità di I/O del sistema C++ sono gestite dalla libreria *iostream* (per l'I/O "esterno") e *fstream* (per l'I/O su file) le quali lavorano con i **flussi (stream)**; si tratta di un sistema flessibile ed efficiente basato sulle classi e sul sovraccaricamento di funzioni e operatori, che rende possibile gestire alla stessa maniera l'I/O di qualunque tipo di dato.

### 10.1 Flussi (*stream*)

Uno *stream* è un'astrazione che rappresenta un flusso di dati che scorre tra un *produttore* e un *consumatore*. Lo "standard input", per esempio, è visto come una corrente di byte che fluisce dalla tastiera verso un programma in esecuzione sulla CPU (Figura 10.1), e lo "standard output" come un flusso che scorre dalla CPU verso un dispositivo di uscita. Si dice "*estrarre da un flusso*" per riferirsi alla ricezione dati da un dispositivo di input, e "*inserire in un flusso*" per indicare la trasmissione di dati a un dispositivo di output.

Interpretando il flusso di byte come flusso di testo, cioè come sequenza di caratteri, può capitare che non si stabilisca una relazione biunivoca tra i byte introdotti o estratti dal flusso e i caratteri effettivamente scritti o letti. Questo perché in un flusso di testo possono occorrere certe conversioni di caratteri richieste dall'ambiente del sistema. Per esempio, il byte corrispon-

**Figura 10.1**

Simulazione di un flusso.

dente al carattere *newline* ('/\n' in C++) può essere interpretato come coppia di caratteri «*carriage-return/newline*» ("/\r\n" in C++).

Un *flusso binario* è la sequenza di byte pura e semplice, cioè non interpretata. In questo caso, la quantità di byte scritti/letti coincide con quella dei byte inseriti/estratti dal flusso.

La libreria che gestisce l'I/O in C++ è `iostream` ed è scritta secondo il paradigma della programmazione orientata agli oggetti che semplifica molto la gestione di una materia così complessa. Quindi per presentare i concetti faremo ampio uso del vocabolario tipico dell'Object Oriented Programming, usando spesso i termini "classe" e "oggetto" senza averli ancora introdotti. Rimedieremo presto, nel prossimo capitolo, ma per ora sarà sufficiente anticipare che una "classe" è un insieme di "oggetti" i quali sono strutture dati con annesse funzioni per operare su quei dati. Definita una class, quasi come si definisse un tipo struct, un oggetto altro non è che una variabile di quel tipo, quindi l'oggetto sta alla sua classe come una variabile sta al suo tipo.

L'input standard è modellato come l'oggetto `cin` della classe `istream` e l'output standard come l'oggetto `cout` della classe `ostream`, entrambe contenute nella libreria `iostream`. Normalmente l'input standard è associato (dal sistema operativo) alla tastiera e l'output standard allo schermo.

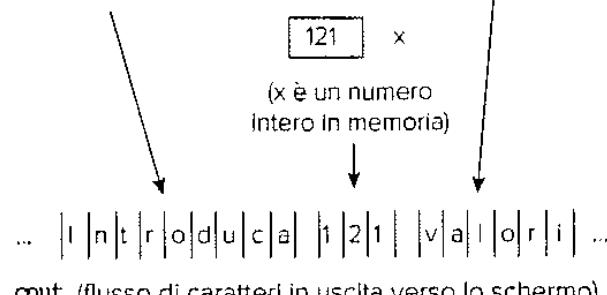
Sui flussi lavorano gli operatori "inseritore", `<<`, ed "estrattore", `>>`, che sono magistralmente sovraccaricati per gestire automaticamente variabili di qualunque tipo predefinito. La Figura 10.2 mostra questa conversione per i flussi `cin` e `cout`. L'inseritore `<<` converte i dati che appaiono nel suo operando/i di destra in una sequenza di caratteri che "inserisce" nel flusso di output `cout`. Al contrario, l'estrattore `>>` tira dal flusso di input una sequenza di caratteri convertendola nel formato appropriato per l'operando/i che si trova alla sua destra.

## 10.2 La libreria `iostream`

La gerarchia delle classi di I/O (Figura 10.3) è distribuita su quattro header file: `<iostream>` dichiara le classi `istream`, `ostream` e `iostream` per le operazioni di I/O con i flussi di input e output standard, nonché gli oggetti `cout`, `cin`, `cerr` e `clog` che si utilizzano nella maggior parte dei programmi C++ (Tabella 10.1).

**Conversione in uscita (da rappresentazione interna a caratteri)**

```
cout << "Introduca " << setw(3) << x << " valori interi ";
```



**Conversione in ingresso**

```
cin >> giorno >> mese >> anno;
```

cin (flusso di caratteri letti dalla tastiera)



**Figura 10.2**

Conversione di dati a/da flussi di caratteri.

**Tabella 10.1 Oggetti di <iostream>**

Oggetto di flusso	Funzione
cin	Oggetto della classe istream collegato all'input standard
cout	Oggetto della classe ostream collegato all'output standard
cerr	Oggetto della classe ostream collegato all'errore standard, per output senza buffer
clog	Oggetto della classe ostream collegato all'errore standard, per output tramite buffer

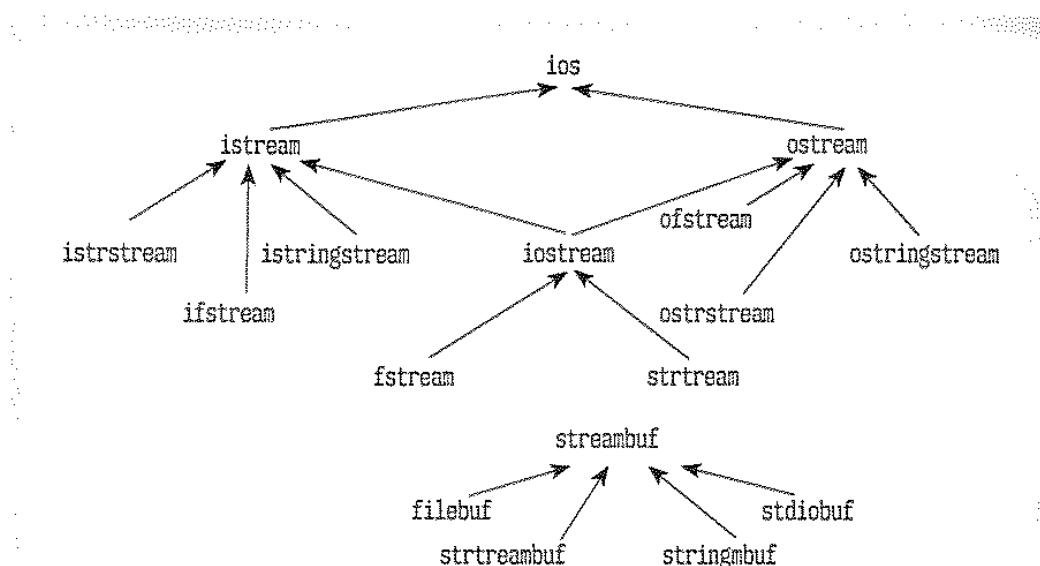
<fstream> dichiara le classi ifstream, ofstream e fstream per operazioni di I/O su file (memoria di massa).

<sstream> dichiara le classi istringstream e ostringstream.

<strstream> dichiara le classi istrstream, ostrstream e strstream per formattare dati con buffer di caratteri.

### 10.2.1 Gerarchia di classi ios

La classe istream (*input stream*) fornisce le operazioni di lettura, mentre la classe ostream (*output stream*) implementa le operazioni di scrittura. La classe

**Figura 10.3**

Libreria di classi di I/O.

`iostream` (*input-output stream*) deriva sia da `istream` sia da `ostream`, e fornisce operazioni bidirezionali di input/output (Figura 10.3).

Le classi `istrstream` e `ostrstream` si utilizzano per gestire le *C-string* e i flussi, mentre le classi `istringstream` e `ostringstream` si utilizzano per gestire gli oggetti della classe `string` (che vedremo poi).

### 10.2.2 Flussi standard

Il flusso `cin`, definito nella classe `istream`, è collegato all'input standard; se non è stato rediretto dal sistema operativo, l'input standard è la tastiera, rappresentata dal file `stdin`. Il flusso `cout`, definito dalla classe `ostream`, è collegato all'output standard, cioè lo *schermo*, rappresentato dal file `stdout` (se non è stato rediretto). Il flusso `cerr`, definito dalla classe `ostream`, è collegato al flusso "standard error" (anche questo normalmente diretto verso lo schermo); questo flusso non usa un buffer di memoria (manda i dati direttamente allo schermo). Il flusso `clog` è anch'esso collegato allo "standard error" ma si gestisce tramite un buffer (invia i dati al dispositivo collegato allo standard error solo dopo averli appoggiati, ed eventualmente editati, su una zona di memoria).

Qualunque oggetto della classe `ios` o di qualunque delle sue classi derivate è un *oggetto flusso*.

### 10.2.3 Input/output su file

L'I/O su file è gestito dalle seguenti tre classi nell'header file `fstream`:

- `ifstream`, derivata da `istream` e utilizzata per gestire la lettura da un file. Quando si crea un oggetto `ifstream` e se ne specificano i parametri, si apre un file in lettura.

- `ofstream`, derivata da `ostream` e utilizzata per gestire la scrittura su di un file. Quando si crea un oggetto `ofstream` e se ne specificano i parametri, si apre un file in scrittura su disco. Se un oggetto di `ofstream` è già dichiarato, si può utilizzare la funzione `open()` per aprire il file o la funzione `close()` per chiuderlo.
- `fstream`, derivata da `iostream` e utilizzata per gestire sia la lettura sia la scrittura su file. Gli oggetti `fstream` si utilizzano quando si vuole effettuare simultaneamente operazioni di lettura e scrittura nello stesso file.

#### 10.2.4 Input/output su buffer di memoria

Nell'header file `strstream` vi sono due classi specifiche per l'I/O su buffer di memoria:

- `istrstream`, derivata da `istream` e utilizzata per leggere caratteri da una zona di memoria, che serve da flusso di input.
- `ostrstream`, derivata da `ostream` e utilizzata per scrivere caratteri in una zona di memoria, che serve da flusso di output.

### 10.3 La classe `istream`

La classe `istream` permette di definire un flusso di input e contiene metodi per accettare dati in modo *formattato* e *non formattato*. Per poter effettuare operazioni di input ad alto livello l'estrattore `>>` è sovraccaricato per tutti i tipi di dato fondamentali del C++: `char`, `unsigned char`, `signed char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`, stringhe e puntatori.

L'estrattore `>>` si applica a un oggetto `istream` e a un oggetto variabile.

`oggetto_istream >> oggetto_variabile`

e cerca di estrarre da `oggetto_istream` una sequenza di caratteri corrispondenti a un valore del tipo di `oggetto_variabile`. Se non trova caratteri ammissibili si blocca. Per esempio, supponiamo che `cin` sia inizialmente vuoto e si eseguano le istruzioni:

```
int eta;
cin >> eta;
```

Se l'utente batte sulla tastiera 25, immette i caratteri 2 e 5 nell'oggetto `cin`, l'estrattore li legge, li associa al valore intero "venticinque" lo rappresenta in "complemento a due" (ovvero come si rappresentano in binario i numeri interi) e lo scrive nell'operando di destra `eta`.

Cosa succede se l'utente introduce valori non appropriati? Per esempio, invece di scrivere 25 digita t5? Il flusso `cin` conterrà i caratteri t e 5 e l'estrattore cerca di estrarre un carattere cifra ma trova il carattere alfabetico t! La classe `istream` ha un attributo che è il suo *stato* rappresentato tramite *flags*. Le tre condizioni base sono: `good` (lo stato è buono), `bad` (c'è qualcosa di scorretto nel flusso), `fail` (l'ultima operazione del flusso non ha avuto successo). La classe `istream` contiene una variabile `flag` booleana per ognuno di questi stati,

**Tabella 10.2** Indicatori di stato

Chiamata a funzione:	Riporta:
<code>cin.good()</code>	true se è solo se è tutto corretto in cin
<code>cin.bad()</code>	c'è qualcosa di sbagliato in cin
<code>cin.fail()</code>	non s'è potuta completare l'ultima operazione

con l'indicatore `good` inizializzato a `true` e gli indicatori `bad` e `fail` inizializzati a `false`. Quando si trova la lettera `t` si mette l'indicatore `good` del flusso a `false` e gli indicatori `bad` e `fail` a `true`. Per ognuno di loro la classe `istream` fornisce una funzione membro booleana che informa del valore di quell'indicatore (Tabella 10.2).

L'estrattore può operare in cascata, con un formato del tipo:

```
cin >> variabile1 >> variabile2 >> ... >> variabileN;
```

Quando si immettono più dati in input, si deve digitare almeno uno spazio bianco fra loro. Esso legge i caratteri uno a uno ignorando spazi bianchi, tabulazioni e caratteri di fine riga. La lettura di un carattere alla volta richiede una variabile per ogni carattere, per questo è meglio leggere intere stringhe, ma il problema è che l'estrattore non può lavorare con stringhe che contengono più di una parola, che contengano cioè spazi bianchi. Per risolvere questo problema il modo migliore è utilizzare due dei metodi dell'oggetto `cin`, ovvero `get()` e `getline()`. Abbiamo già incontrato queste due funzioni di libreria; adesso specifichiamo, ma sarà chiaro quando introdurremo l'OOP, che si tratta di funzioni membri di un certo oggetto, e che per mandare in esecuzione una funzione di un certo oggetto si fa come se quella funzione fosse membro di una struttura, cioè si scrive `oggetto.funzione_membro()`. Quindi nel nostro caso scriveremo `cin.get()` o `cin.getline()`.

La `cin.get()` legge un carattere o una stringa dal flusso associato all'oggetto `cin`. Essa ha diversi formati, con parametri o senza.

Senza parametro `get()` restituisce il valore del carattere trovato o `EOF` («end of file») se si raggiunge la fine del file, e questo viene interpretato come il valore booleano `false`.

#### Esempio 10.1

```
int main()
{
    char c;
    while (c = cin.get())
    {
        cout << c ;
    }
    cout << "\n Terminato!\n";
    cout << "EOF = " << EOF << endl;
    return 0;
}
```

## Esecuzione di prova

```
Ciao
Ciao
Mondo
Mondo
^D // oppure Ctrl+z
Terminato!
EOF = -1
```

Ogni chiamata alla funzione `cin.get()` legge un carattere in più da `cin` e lo mette nella variabile `c`. Poi, l'istruzione all'interno del ciclo inserisce nel flusso di output il carattere contenuto nella variabile `c`. Questi caratteri si accumulano in realtà in un buffer che viene svuotato quando si batte il tasto **<Invio>**. Quando si incontra EOF (*Ctrl+z* o *Ctrl+D*), si esce dal ciclo.

Con un parametro, il prototipo della funzione `get()` è:

```
istream& get(char&);
```

In questo formato la funzione legge il carattere dal flusso di input mettendolo nel parametro passatole per riferimento. Questa versione restituisce *null* (ovvero false) quando incontra la fine del file; la si può utilizzare per controllare un ciclo di input:

```
while (cin.get(car))
```

L'esempio precedente diventa:

```
int main()
{
    char c;
    while (c = cin.get())
    {
        cout << c ;
    }
    cout << "\n Terminato!\n";
    cout << "EOF = " << EOF << endl;
    return 0;
}
```

## Esempio 10.2

Un'altra applicazione di `cin.get(char&)`.

```
int main()
{
    char a, b, c;
    cout << "Introduca tre lettere:" ;
    cin.get(a).get(b).get(c);
    cout << "a: " << a << "\nb: " << b << "\nc: " << c << endl;
    return 0;
}
```

### Esecuzione di prova

Introduca tre lettere: Ugo

a: U

b: g

c: o

La chiamata composta `cin.get(a).get(b).get(c)` è possibile perché, come si vede nel prototipo, questo formato della funzione restituisce il riferimento all'oggetto flusso di cui è membro. Quindi la chiamata `cin.get(a)` restituisce `cin` che può successivamente comporsi con `.get(b)` e così via.

Un terzo formato della funzione `get()` (simile alla funzione `getline()` che vedremo poi) accetta due o tre argomenti. Il suo prototipo è

```
istream& get(char* buffer, int n, char sep = '\n');
```

Questo formato legge fino a  $n-1$  caratteri dal `buffer`, oppure (ma il terzo argomento è facoltativo) fino a incontrare il carattere separatore `sep`. Vale a dire che il primo parametro è un puntatore a un array di caratteri, il secondo è il numero massimo di caratteri da leggere più uno e il terzo è il carattere di terminazione. Per esempio:

```
char stringa[80];
cin.get(stringa, 80); // Legge caratteri fino a trovare un
                      // carattere newline oppure averne letti 79
```

### Esempio 10.3

```
int main()
{
    char buffer[80];
    cin.get(buffer, 8);
    cout << "[" << buffer << "]\n";
    cin.get(buffer, sizeof(buffer));
    cout << "[" << buffer << "]\n";
}
```

### Esecuzione di prova

Non so cosa dire

[Non so ]

[cosa dire]

L'altra funzione per la lettura, `getline()`, è molto simile alla `get()` con due o tre argomenti, ma `getline()` inserisce nella stringa il carattere di terminazione. Il formato è:

```
cin.getline(str, long max_str+2, car_separatore);
```

il numero massimo di caratteri che vengono letti deve essere maggiore della stringa reale almeno di due caratteri, per accettare il carattere separatore e poi '`\0`'. Se non si specifica alcun carattere separatore, si prende per default il '`\n`'.

La differenza tra `get()` e `getline()` è solo che quest'ultima immagazzina il carattere separatore o delimitatore nella stringa prima di aggiungere il carattere nullo.

Con `getline()` si presentano però problemi quando si tenta di utilizzare una variabile stringa, dopo aver letto da `cin` una variabile carattere o numerica. Per esempio, consideriamo il seguente programma:

```
int main()
{
    char Nome[30];
    int Eta;
    cout << "Introduca età: ";
    cin >> Eta;
    cout << Eta;
    cout << "Introduca il nome: ";
    cin.getline(Nome, 30);
    cout << Nome;
}
```

Se si introducono in `Eta` e `Nome` i valori 969 e Matusalemme:

```
Introduca età: 969
Introduca il nome: Matusalemme
```

I valori che prendono le variabili citate sono:

```
Eta 969
Nome '\n'\0'
```

Il problema è che per introdurre 969, si deve premere il tasto <INVI>. Quest'azione inserisce il carattere '\n' che rimane nel buffer di memoria. La successiva istruzione `cin.getline(Nome, 30)` legge il buffer della tastiera e ci trova il carattere '\n', e poiché questo è il carattere di separazione per default, ferma la lettura e inserisce il carattere '\0' nel vettore impedendo di introdurre il nome. Ci sono due modi per risolvere il problema:

1. specificare un carattere di separazione differente nella funzione `getline()`; l'utente deve introdurre questo carattere che sarà inserito come ultimo carattere prima di '\0';
2. pulire il buffer della tastiera appoggiando il carattere '\n' residuo su di una variabile ausiliaria:

```
int main()
{
    char Ausiliare[2];
    char Nome[30];
    int Eta;
    cout << "Introduca età:";
    cin >> Eta; // Lettura di dati numerici
    cout << Eta;
```

```

    cin.getline(Ausiliare, 2);      // Pulire buffer di tastiera
    cout << "Introduca il nome: ";
    cin.getline(Nome, 30);         // Leggere dati di stringa
    cout << Nome;
}

```

Un'altra alternativa per la lettura da un flusso è la funzione `read()`

```

istream& read(char* buf, int num);
istream& read(unsigned char* buf, int num);
istream& read(signed char* buf, int num);

```

La funzione legge `num` byte dal flusso associato (nel nostro caso la chiamata sarebbe del tipo `cin.read()`) e li colloca nel buffer al quale punta `buf`. Il parametro è un puntatore a un array di caratteri e `num` specifica il massimo numero di caratteri da leggere. La funzione estrae caratteri finché non si raggiunge la fine del file.

#### 10.4 La classe `ostream`

La classe `ostream` permette all'utente di definire un flusso di output e metodi per l'uscita *formattata* e *non formattata*. Da essa derivano le classi `ofstream`, `ostrstream`, `constream`, `iostream` e `ostream_withassign`.

La nozione astratta di flusso nasconde questi dettagli di basso livello al programmatore e associa il «flusso» di caratteri a un arbitrario dispositivo di output.

Si definiscono due oggetti `ostream` perché qualunque programma che include l'header file `iostream` abbia due flussi di output (associati a qualunque dispositivo l'utente stia utilizzando per output: una finestra, una console ecc.).

1. `cout` per visualizzare output normale;
2. `cerr` per visualizzare messaggi di errore o diagnostica.

L'inseritore `<<` è sovraccaricato per tutti i tipi di dato fondamentali: `char`, `unsigned char`, `signed char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double`, `long double`, `void*` e `char*`. Esso si applica a un oggetto `ostream` con la seguente sintassi:

*oggetto\_ostream* `<<` *espressione*

esso valuta *espressione* e inserisce la sequenza di caratteri corrispondenti nell'oggetto `ostream`. Per esempio, l'istruzione:

```
cout << "Ciao mondo, C++";
```

non effettuerà alcuna conversione mandando i caratteri della stringa uno a uno in `cout` (cioè probabilmente sullo schermo o sulla console), ma nelle istruzioni:

```
const double PI = 3.1416;
cout << PI;
```

la funzione << converte il valore double 3.1416, nella corrispondente sequenza di caratteri 3, ., 1, 4, 1 e 6 e li inserisce uno a uno in cout. La variabile PI è un'espressione, dunque potrebbe anche essere valida l'istruzione:

```
int i, j;
cout << i+j;
```

L'inseritore può operare in cascata. Per esempio:

```
cout << 1 << 2 << 3 << 4;
```

genererà un output del tipo

1234

Si possono mischiare stringhe e valori numerici. Così, per esempio, se la variabile Somma di tipo int contiene il valore 450:

```
cout << "Totale = " << Somma << endl;
```

visualizzerà

Totale = 450

Il simbolo endl, come già il lettore sa, fa avanzare il flusso di output alla riga seguente.

Dal punto di vista pratico, ogni operatore di inserimento invia un dato (una costante, un'espressione o una variabile) al flusso di output, e si possono concatenare dati di tipi differenti in un'unica espressione cout.

```
cout << Voltaggio << Corrente << Resistenza;
```

L'istruzione precedente scriverà i valori immagazzinati in memoria nell'ordine in cui sono scritti senza alcuno spazio tra loro (volendo si dovrà inserire "caratteri spazio" dentro l'istruzione cout).

La classe ostream fornisce i metodi put(), per inserire un unico carattere nel flusso di output, e write() per inserirvi una stringa. I prototipi sono:

```
ostream& put(char& c);
ostream& write(const char* buf, int num);
ostream& write(const unsigned char* buf, int num);
ostream& write(const signed char* buf, int num);
```

Entrambe le funzioni restituiscono un oggetto cout. I formati delle chiamate sono:

```
dispositivo.put (carattere);
dispositivo.write (stringa, num);
```

dove num è un valore int utilizzato per specificare il numero di caratteri della stringa da visualizzare. Il dispositivo può essere qualunque flusso di output. Vediamo fra un attimo come scrivere, per esempio, un carattere nella stampante.

Le istruzioni seguenti visualizzano due caratteri ('Z' e 'l') nel flusso di output:

```
cout.put('Z');
char lettera = 'l';
cout.put(lettera);
```

Con la funzione membro write visualizziamo stringhe. Per esempio, le istruzioni:

```
cout.write("Libreria", 3);
cout.write("Festa Nazionale", 15) << "\n";
cout.put(65) << "ntonio Molina \n";
cout.put('C');
cout.write("iao", 4) << " mondo C++ \n";
```

visualizzano:

```
LibFesta Nazionale
Antonio Molina
Ciao mondo C++
```

Si osservi che la prima funzione put() ha come argomento una costante intera che viene ovviamente interpretata come codice ASCII visualizzando la lettera A.

La funzione write() visualizza tanti caratteri quanti specificati nel secondo argomento. Se la quantità è maggiore del numero di caratteri della stringa, la funzione visualizza qualunque cosa risieda in memoria dopo la stringa.

L'invio dell'output di un programma alla stampante è facile con la funzione ofstream. Il formato di ofstream è

`ofstream dispositivo (nome_dispositivo)`

e il suo uso richiede l'header file fstream.

#### Esempio 10.4

Il seguente programma chiede nome e cognome dell'utente e stampa il nome completo sulla stampante.

```
#include <fstream>
int main()
{
    char nome[20];
    char cognome[30];
    cout << "Quale è il suo nome?";
    cin >> nome;
    cout << "Quale è il suo cognome?";
    cin >> cognome;
    // Invia nome e cognome alla stampante
    ofstream stampante ("PRN");
```

**Tabella 10.3** Conversione per output di diversi tipi di dato

Tipo	Tipo di conversione di output
char	I caratteri stampabili si visualizzano con la larghezza di una colonna. I caratteri di controllo, come nuova riga, tabulazione ecc., possono produrre più caratteri di output
int	Qualunque tipo intero (int, short o long) si visualizza come numero decimale con larghezza sufficiente per contenere il numero e il segno meno se l'intero fosse negativo
Stringa	La larghezza su schermo è uguale alla lunghezza della stringa
float	I numeri reali in virgola mobile si visualizzano con la precisione di sei cifre decimali. Gli zeri non significativi non vengono visualizzati. Se il numero è molto grande o molto piccolo, si visualizza il numero con un esponente di due cifre (o tre cifre se il tipo è double) dopo la lettera "e". La larghezza è sempre sufficiente per mantenere un segno meno e/o un esponente

```

stampante << "Il suo nome completo è: \n";
stampante << cognome << ", " << nome << endl;
return 0;
}

```

## 10.5 Formattazione dell'output

Se non si istruisce l'inseritore per realizzare operazioni di formattazione specifiche, l'output avviene in maniera diversa a seconda del tipo di dato convertito, secondo quanto riportato nella Tabella 10.3.

Ovviamente per molti scopi è necessario avere più flessibilità nella formattazione della visualizzazione dei dati, e questo si può fare tramite i manipolatori, funzioni speciali progettate specificamente per modificare il formato di un oggetto flusso (Tabella 10.4). La libreria iomanip contiene alcuni manipolatori e altri se ne possono aggiungere facilmente. I manipolatori si

**Tabella 10.4** Manipolatori di flussi

Manipolatore	Azione
dec	utilizza conversione decimale ( <i>per default</i> )
hex	utilizza conversione esadecimale
oct	utilizza conversione ottale
ws	estrae caratteri spazi in bianco
endl	aggiunge il carattere <i>newline</i> al flusso di output ('\n')
ends	aggiunge il carattere terminale nullo al flusso di output ('\0')
flush	svuota il buffer di output inviando immediatamente i dati nel flusso di output
setbase(n)	stabilisce la base di conversione a <i>n</i> (0, 8, 10 oppure 16). 0 significa decimale <i>per default</i>
setprecision( <i>n</i> )	stabilisce la precisione di virgola mobile a <i>n</i>
setw( <i>n</i> )	stabilisce la larghezza del campo a <i>n</i>
setfill(c)	stabilisce il carattere di riempimento a <i>c</i>
setiosflags(f)	setta i bit di formato specificati dall'argomento <i>f</i> di tipo long
resetiosflags(f)	azzerà i bit di formato specificati dall'argomento <i>f</i> di tipo long

utilizzano prevalentemente per indicare formati come la larghezza di un campo, la precisione dei numeri in virgola mobile ecc.

I manipolatori si inseriscono nell'istruzione cout come qualunque altro elemento. La sintassi tipica è:

```
cout << setw (larghezza del campo) << dato_in_output;
```

ma bisogna ricordarsi di includere header file <iomanip>

```
#include <iomanip>
```

```
cout << setw(3) << i << setw(5) << i*j*i;
```

Normalmente, gli interi si visualizzano come decimali (numeri scritti in base 10); è però possibile selezionare una base di numerazione diversa (ottale, esadecimale) invocando i manipolatori dec, hex oppure oct. Così, per esempio,

```
cout << dec << Totale << endl;
```

visualizza il valore di Totale in base 10. dec è importante per riselectare la base 10 dopo aver lavorato con altre basi. Per esempio, se Totale prende il valore decimale 255, l'istruzione

```
cout << hex << Totale << endl;
```

scrive a schermo:

*ff* (*valore esadecimale corrispondente a 255 decimale*)

che è la rappresentazione esadecimale di 255. Si possono mescolare in una stessa istruzione diversi manipolatori:

```
cout << "Base 10 = " << dec << Totale
     << "Base 16 = " << hex << Totale << endl;
```

### Esempio 10.5

```
#include <iomanip>
int main()
{
    cout << "Valore hex di " << 40 << " in decimale è: "
         << hex << 40 << endl;
    cout << "Valore ottale di " << hex << 34 << " in esadecimale è: "
         << oct << 34 << endl;
    cout << dec; // Si ristabilisce la numerazione decimale
}
```

### Esecuzione

Valore hex di 40 in decimale è: 28

Valore ottale di 34 in esadecimale è: 42

Il manipolatore setbase(int n) fissa la base numerica a 8, 10 o 15. Questo manipolatore parametrizzato funziona come oct, dec, e hex. Per esempio:

```
cout << setbase(10);
cin >> setbase(10);
```

Il manipolatore `setw()` controlla la larghezza del formato di output. Il prototipo è:

```
setw(int n)
```

Un esempio dell'uso di `setw()` per visualizzare un campo lungo otto caratteri è

```
cout << "12345678901234567890" << endl;
cout << setw(8) << "Ciao";
cout << " Mondo";
```

che produce l'output seguente:

```
12345678901234567890
    Ciao Mondo
```

cioè, Ciao è stato scritto all'interno di un campo di otto caratteri allineato a destra.

### Esempio 10.6

```
#include <iomanip>
int main()
{
    cout << setw(10) << "M" << setw(10) << "N" << endl;
    cout << setw(10) << 1 << setw(10) << 7.77 << endl;
    cout << setw(10) << 10 << setw(10) << 77.77 << endl;
    cout << setw(10) << 100 << setw(10) << 777.77 << endl;
}
```

### Esecuzione

```
M      N
 1      7.77
10     77.77
100    777.77
```

Si può utilizzare la funzione `width()` per specificare la larghezza del campo della successiva operazione dell'inseritore. Per esempio:

```
cout << "ABC";
cout.width(6);
cout << "DEF" << "GHI";
```

visualizza

```
ABC  DEFGHI
```

dove DEF è preceduto da due spazi, mentre le altre operazioni sono normali. Il carattere di riempimento può essere specificato tramite la funzione `fill()`. Per esempio:

```
cout << "ABC";
cout.width(5);
cout.fill('@');
cout << "DEF" << "GHI";
```

visualizza

```
ABC@DEFGHI
```

Le istruzioni:

```
cout << "12345678901234567890" << endl;
cout.width(15);
cout.fill('*');
cout << "Ciao Mackoy" << endl;
float Z = 99.99;
cout.width(15);
cout << Z;
```

produrranno:

```
12345678901234567890
*****Ciao Mackoy
*****99.989998
```

Il carattere di riempimento permane fino al successivo cambiamento.

I numeri in virgola mobile sono visualizzati per default fino a sei cifre di precisione. Si può modificare il numero delle cifre con il manipolatore setprecision(). Il formato è:

```
cout << setprecision(int i);
```

#### Esempio 10.7

```
#include <iomanip>
int main()
{
    float prova = 814.159265;
    cout << setprecision(2) //2 cifre significative
        << prova << endl;
    cout << setprecision(3) //3 cifre significative
        << prova << endl;
    cout << setprecision(4) //4 cifre significative
        << prova << endl;
    cout << setprecision(5) //5 cifre significative
        << prova << endl;
}
```

#### Esecuzione

```
8.1e+02
814
814.2
814.16
```

Per stabilire la precisione si può utilizzare anche il metodo `precision()`. Per esempio, l'istruzione che fissa a 3 la precisione per l'output è:

```
cout.precision(3);
```

## 10.6 Indicatori di formato

Ogni oggetto flusso delle classi `istream` e `ostream` contiene un insieme di flags che specificano quale sia in ogni momento il suo «formato». Questo modo di procedere è diverso da quello impiegato dalle funzioni C, come `printf` o `scanf`, nelle quali a ogni operazione di input/output si forniscono gli indicatori di formattazione appropriati.

Ognuno di questi *flags* si può settare e azzerare utilizzando un manipolatore come `setiosflags(long)` e `resetiosflags(long)`.

Per esempio, il manipolatore `setiosflags()` si definisce come:

```
setiosflags(long f)
```

e serve per specificare l'allineamento del dato nel suo campo (il default è a destra). La seguente istruzione attiva l'opzione di allineamento a sinistra:

```
cout << setiosflags(ios::left);
```

Gli argomenti dei manipolatori (bit indicatori di `ios`) sono elencati nella Tabella 10.5.

**Tabella 10.5 Bit di stato (parola di stato della formattazione)**

Bit di stato (argomento)	Proposito
<code>skipws</code>	Salta spazi in bianco in operazioni di input
<code>left</code>	Allinea l'output a sinistra del campo
<code>right</code>	Allinea l'output a destra del campo
<code>internal</code>	Riempie il campo dopo il segno o l'indicatore base
<code>dec</code>	Attiva conversione decimale
<code>oct</code>	Attiva conversione ottale
<code>hex</code>	Attiva conversione esadecimale
<code>showbase</code>	Visualizza l'indicatore di base numerica <i>Esempio:</i> 044 (numero ottale) 0x2ea7 (numero hex)
<code>showpoint</code>	Visualizza punto decimale in valori di virgola mobile <i>Esempio:</i> 456.00
<code>uppercase</code>	Visualizza valori esadecimali in maiuscolo <i>Esempio:</i> 4BFF
<code>showpos</code>	Visualizza numeri interi positivi preceduti dal segno +
<code>scientific</code>	Notazione scientifica per i numeri in virgola mobile <i>Esempio:</i> 3.1416 e + 00
<code>fixed</code>	Utilizza notazione in virgola fissa per numeri in virgola mobile <i>Esempio:</i> 1234.8
<code>unitbuf</code>	Svuota i buffer dopo ogni scrittura
<code>stdlo</code>	Svuota i buffer dopo ogni scrittura su <code>stdout</code> o <code>stderr</code>

Si deve far precedere ogni argomento (bit di stato) dalla clausola `ios::` che risolve la sua associazione con la classe `ios`. Per esempio,

```
float pi = 3.14159;
cout << setiosflags(ios::fixed) << pi << endl;
```

seleziona un valore di virgola mobile con notazione fissa e l'output sarà:  
3.14159

Si seleziona la notazione scientifica utilizzando l'istruzione seguente:

```
cout << setiosflags(ios::scientific) << pi << endl;
```

che visualizza l'output seguente:

```
3.14159e+00
```

Si possono anche usare più indicatori in un'unica operazione tramite l'operatore `|`. Per esempio,

```
cout << setiosflags(ios::dec|ios::showbase) << Totale << endl;
```

Per resettare gli indicatori di stato si usa `resetiosflags()`. Per esempio, per pulire il parametro `showbase`, scrivere:

```
cout << resetiosflags(ios::showbase) << Totale << endl;
```

Così, per esempio, se si attiva l'opzione di allineamento a sinistra:

```
cout << setiosflags(ios::left);
```

si disattiverà poi con l'istruzione:

```
cout << resetiosflags(ios::left);
```

Un altro esempio per illustrare l'uso di differenti basi di numerazione:

```
cout << setiosflags(ios::showbase)
    << "\n" << v << " "
    << oct << v << " "
    << hex << v << endl;
```

che produce l'output:

```
100 0144 0x64
```

Il seguente programma mostra un sistema per formattare dati di output in diverse forme.

### Esempio 10.3

```
#include <iomanip>

int main()
{
    float v1 = 4500.25;
    float v2 = 325.993;
    float v3 = 54225;
    cout << setiosflags(ios::showpoint|ios::fixed)
```

```

    << setprecision(2)
    << setfill('*')
    << setiosflags(ios::right);
cout << "Saldo Finale: $" << setw(10) << v1 << endl;
cout << "Saldo Finale: $" << setw(10) << v2 << endl;
cout << "Saldo Finale: $" << setw(10) << v3 << endl;
}

```

**Esecuzione**

Saldo Finale: \$\*\*\*4500.25  
 Saldo Finale: \$\*\*\*\*325.99  
 Saldo Finale: \$\*\*54225.00

**Esempio 10.9**

```

#include <iomanip>
int main()
{
    const float p = 3.14159;
    cout << setiosflags(ios::showpos|ios::scientific)
        << "\n Il valore di PI è "
        << setprecision(3)
        << setw(15) << setfill('*')
        << setiosflags(ios::right)
        << p;
}

```

**Esecuzione**

Il valore di PI è \*\*\*\*\*+3.142e+00

Anche le funzioni membro `setf()` e `unsetf()` modificano gli indicatori di flusso. Queste funzioni sono simili ai manipolatori `setiosflags()` e `resetiosflags()`, con la differenza che `setf()` e `unsetf()` sono vere funzioni membro. Le si accede direttamente:

```

cout.setf(ios::scientific);
cout.unsetf(ios::scientific);

```

**10.7 I/O da file**

C++ utilizza i flussi anche per gestire i file. In C++, un file è semplicemente un flusso esterno: una sequenza di byte in memoria di massa. Se il file viene aperto per scriverci, è un flusso di output, se viene aperto per leggervi dentro, è un flusso di input.

La libreria dei flussi di I/O contiene tre classi, `ifstream`, `ofstream` e `fstream`, con metodi associati per creare file e manipolarli. Queste classi sono nell'header file `<fstream>` (che include `<iostream>`).

Prima che un programma possa leggere o scrivere su un disco deve *aprire* il file. Per aprire un file di testo in lettura, si crea un oggetto flusso della classe `ifstream`; per aprire un file in scrittura si crea un oggetto della classe `ofstream`. Si possono poi utilizzare i nomi dei flussi con gli operatori di inserimento ed estrazione.

Come i flussi `cin` e `cout`, anche i flussi sui file possono trasferire dati solo in una direzione. Ciò significa che si debbono aprire e manipolare flussi indipendenti per la lettura e la scrittura di file di testo. La libreria di flussi C++ offre un insieme di funzioni membro comuni a tutte le operazioni di I/O su file.

Per *aprire il file in lettura* bisogna dichiarare un oggetto flusso e associarlo a un file:

```
ifstream fin ("demo");
```

l'oggetto si chiama `fin` ed è associato a un file il cui nome è "demo". Il nome è il parametro che si passa al costruttore di classe per consentirgli di localizzare e aprire il file.

Per *aprire il file in scrittura* non basta dichiarare un oggetto flusso e associarlo a un file ma bisogna specificare anche la modalità d'apertura in un secondo parametro:

```
ofstream fout ("demo", ios::out);
```

perché un flusso `ofstream` in scrittura si può aprire in due modi: *output* (`ios::out`) e *append* (`ios::app`). Per default, un file `ofstream` si apre in modo *output*:

```
ofstream fout ("demo");
```

in questo modo, se un file esiste esso viene totalmente riscritto. Se si vuole *aggiungere* invece che *rimpiazzare* i dati dentro un file esistente, lo si deve aprire in modalità *append*; in questa maniera i dati mandati nel flusso si aggiungeranno dopo la fine del file. Se il file non esiste verrà creato in entrambe le modalità.

### ATTENZIONE

Prima di tentare di leggere o scrivere un file, è sempre buona idea verificare che sia aperto con successo. Si può verificare `fout` nel modo seguente:

```
if (!fout) { // apertura fallita
    cerr << "non si può aprire demo per output\n";
    exit (-1);
}
```

### Esempio 10.10

Il seguente programma ottiene caratteri dall'input standard e li scrive nel file `demo`.

```
#include <iostream>
int main()
```

```

{
    ofstream fout ("demo"); // apre file demo per output
    if (!fout) {
        cerr << "Non si può aprire demo per output:" << endl;
        return -1;
    }
    char car;
    while (cin.get(car)) fout.put (car);
    return 0;
}

```

**Esecuzione**

non so cosa scrivere  
questa è un'altra riga

demo

non so cosa scrivere  
questa è un'altra riga

**Esempio 10.11**

Il seguente programma legge un file specificato dall'utente e scrive il suo contenuto nell'output standard.

```

#include <iostream>
int main()
{
    cout << "Scriva il nome del file: ";
    char nome_file[80];
    cin >> nome_file;
    ifstream FileIn (nome_file); // apre un file per input
    if (!FileIn) {
        cerr << "Non si può aprire file di input: "
            << nome_file << endl;
        return -1;
    }
    char car;
    while (FileIn.get(car)) cout.put (car);
    return 0;
}

```

**Esecuzione**

Scriva il nome del file: **demo**  
non so cosa scrivere  
questa è un'altra riga.

La classe `fstream` permette di accedere ai file sia in input sia in output.

Tabella 10.6 Valori dell'argomento *modo* di open

Argomento	Modo
ios::in	Modo input
ios::app	Modo append
ios::out	Modo output
ios::ate	Aprire e cercare la fine del file
ios::nocreate	Genera un errore se non esiste il file
ios::trunc	Tronca il file a 0 se esiste già
ios::noreplace	Genera un errore se il file esiste già
ios::binary	Il file si apre in modo binario

Si può chiamare il costruttore della classe (vedremo successivamente il significato di questo termine) senza argomenti, solo per creare un flusso che poi si associerà a qualche file mediante metodi opportuni. Per esempio:

```
ifstream ent; // crea un flusso di input
ofstream usc; // crea un flusso di output
fstream entrambi; // crea un flusso di input e output
```

Una volta creato il flusso, si può utilizzare la funzione `open()` per associarlo a un file e aprirlo. Questa funzione è membro delle tre classi di flusso. Il suo prototipo standard è:

```
void open(const char* nome, int modo, int accesso = filebuf::openprot);
```

dove *nome* specifica il pathname del file da associare al flusso, *modo* determina la modalità d'apertura del file e *accesso* specifica il permesso d'accesso che si assegna al file (il valore per default è `filebuf::openprot`). I possibili valori dell'argomento *modo* si definiscono come enumeratori della classe `ios` (Tabella 10.6).

Alcuni esempi:

```
// Apre il file AUTOEXEC.BAT per input di testo
char cAutoExec[] = "AUTOEXEC.BAT";
fstream f;
f.open (cAutoExec, ios::in);

// Apre il file DEMO.DAT per output di testo
fstream f;
f.open ("DEMO.DAT", ios::out);

// Apre il file PROVE.DAT per input e output binario
fstream f;
f.open ("PROVE.DAT", ios::in | ios::out | ios::binary);
```

Si noti che per aprire un flusso in entrambi i modi, input e output, si debbono specificare i due valori dell'argomento *modo*, `ios::in` e `ios::out`.

Se `open()` fallisce il flusso sarà nullo.

Comunque, benché sia sintatticamente corretto aprire un file tramite la funzione `open()`, normalmente non si fa poiché, come abbiamo visto, le classi `ifstream`, `ofstream` e `fstream` dispongono di funzioni costruttore che aprono

automaticamente il file. Le funzioni costruttore hanno gli stessi parametri e valori di default della funzione open(). Dunque, la forma consueta per aprire un file sarà quella che abbiamo visto prima:

```
ifstream mioflusso ("miofile"); // apre il file per input
```

Se per qualche motivo non si può aprire il file, il valore della variabile di flusso sarà zero.

#### Esempio 10.12

Il seguente pezzo di codice mostra l'uso delle classi ifstream e ofstream. Nell'esempio, un file denominato "sorgente" si apre in lettura e un altro file che si denomina "destinazione" si apre in scrittura. Se entrambi i file si aprono con successo, il contenuto del file sorgente si copia nel file destinazione. Se un file non si può aprire con successo, si segnala un errore:

```
ifstream sorgF("sorgente");
ofstream destF("destinazione");
if(!sorgF || !destF)
    cerr << "Errore sorgente o destinazione open failed\n";
else
    for(char c = 0; destinazione && sorgente.get(c);)
        destinazione.put(c);
```

Il metodo close chiude il file collegato all'oggetto flusso. La sua sintassi è:

```
void close();
```

Per esempio:

```
char cAutoExec = "AUTOEXEC.BAT";
fstream f;
f.open(cAutoExec, ios::in);
...
f.close() // chiude buffer del flusso del file
```

**La funzione close () non utilizza parametri né restituisce valore.**

Oltre le funzioni membro open e close, le classi per l'I/O su file ereditano i metodi già visti per iostream. In particolare richiamiamo:

- il metodo good, che restituisce un valore diverso da zero se non occorre alcun errore in operazioni di flusso. Il prototipo è:  
int good();
- il metodo fail, che restituisce un valore diverso da zero se vi è un errore in un'operazione di flusso. Il prototipo è:  
int fail();
- il metodo eof, che restituisce un valore diverso da zero se il flusso ha raggiunto la fine del file. Il prototipo è:  
int eof();

- l'operatore sovraccaricato `!`, che determina lo stato dell'errore dell'oggetto flusso preso come argomento.

Oltre le funzioni membro ereditate dalle classi `iostream`, le classi `ifstream`, `ofstream` e `fstream` definiscono anche le loro proprie funzioni specifiche:

`void attach(int fd)`

collega l'oggetto flusso al flusso referenziato dal descrittore del file `fd`.

`filebuf* rdbuf()`

restituisce un array `filebuf` associato con l'oggetto flusso.

La lettura e la scrittura su file di testo si farà poi con gli operatori `<< e >>`.

#### Esempio 10.13

Il seguente programma scrive un intero, un valore in virgola mobile e una stringa in un file chiamato `demo`.

```
#include <fstream>
int main()
{
    ofstream usc ("demo");
    if (!usc) {
        cout << "Non si può aprire il file" << endl;
        return 1;
    }
    usc << 10 << " " << 325.45 << endl;
    usc << "Esempio di file di testo" << endl;
    usc.close();
    return 0;
}
```

#### Esecuzione

`demo`

10 325.45
Esempio di file di testo

#### Esempio 10.14

Il programma seguente legge un numero intero, un numero in virgola mobile, un carattere e una stringa dal file `demo`.

```
#include <fstream>
int main()
{
    int i;
    float f;
    char str[40];
    ifstream ent ("demo");
```

```

if (!ent)
{
    cout << "Non si può aprire il file" << endl;
    return 1;
}
ent >> i;
ent >> f;
ent.getline(str,80,'!');
cout << i << ' ' << f << ' ' << str << endl;
ent.close();
return 0;
}

```

**Esecuzione**

10 325.45

Esempio di file di testo

**Esempio 10.15**

Scrivere un programma per salvare su un file i voti riportati dagli studenti.

```

#include <iostream>
int main()
{
    ofstream fileusc("voti.dat", ios::out);
    if (!fileusc) {
        cerr << "Errore: non si può aprire file di output" << endl;
        exit(1);
    }
    char mat[6], nome[20];
    int voto;
    cout << "\n1:";
    int n = 1;
    while (cin >> nome >> mat >> voto) {
        fileusc << nome << " " << mat << " " << voto << endl;
        cout << "\n" << ++n << " ";
    }
    fileusc.close();
}

```

**Esecuzione**

1:	Mackoy	951234	7
2:	Carrigan	962146	8
3:	Mortimer	991045	9
4:	Mackena	981146	9
5:	Garcia	982045	5
6:	Rodriguez	991122	1
7:	Lopez	961333	6

**voti.dat**

Mackoy	961234	7
Carrigan	962146	8
Mortimer	991045	9
Mackena	981146	9
Garcia	982045	5
Rodriguez	991122	1
Lopez	961333	6

**Esempio 10.16**

Leggere i voti riportati dagli studenti dal file voti.dat e scrivere in output la media.

```
#include <iostream>
int main()
{
    ifstream FileIn ("voti.dat", ios::in);
    if (!FileIn) {
        cerr << "Errore: non si può aprire file di input" << endl;
        exit(1);
    }
    char mat[6], nome[20];
    int voto, somma = 0, conto = 0;
    while (FileIn >> nome >> mat >> voto) {
        somma += voto;
        ++conto;
    }
    FileIn.close();
    cout << "Il voto medio è " << float(somma)/conto << endl;
}
```

**Esecuzione**

Il voto medio è 6.42857

**10.8 I/O binario**

File che rappresentano immagini, suoni o filmati non sono file di testo, ovvero i byte non sono pensati per rappresentare testo. Per scrivere e leggere dati in formato binario da/a un file si utilizzano le funzioni put(), get(), read() e write().

**Esempio 10.17**

Il seguente programma mostra a schermo il contenuto di un file che riceve come parametro.

```
#include <iostream>
int main(int argc, char* argv[])
{
```

```

char c;
ifstream ent(argv[1], ios::in | ios::binary);
if (!ent)
{
    cout << "Non si può aprire il file" << endl;
    return 1;
}
while (ent) // ent è 0 quando si raggiunge eof
{
    ent.get(c);
    cout << c;
}
ent.close();
return 0;
}

```

Supponendo di aver compilato questo programma generando l'eseguibile `mostra_file` e che si abbia nella stessa cartella un file `immagine.jpg`, allora si potrà lanciare il programma da terminale:

```
s: mostra_file immagine.jpg
```

Il ciclo `while` termina quando `ent` raggiunge la fine del file. Un modo più breve per implementare il ciclo di lettura è:

```
while (ent.get(c)) cout << c;
```

#### Esempio 10.18

Scrive una stringa con caratteri non ASCII nel file `prova.jpg`.

```

#include <iostream>
int main()
{
    char *p = "Ciao amici \n\r\xff";
    ofstream usc("prova.jpg", ios::out | ios::binary);
    if (!usc)
    {
        cout << "Non si può aprire il file" << endl;
        return 1;
    }
    while (*p) usc.put(*p++);
    usc.close();
    return 0;
}

```

#### Esecuzione

Aprendo il file `prova.jpg` con un editore esadecimale se ne può vedere il contenuto byte per byte:

43 69 61 6F 20 61 6D 69 63 69 20 0A 0D FF
---

C i a o a m i c i \n \r \xff
------------------------------

**ESEMPIO 10.10**

Scrittura e lettura di un array di interi in formato binario.

```
#include <iostream>
int main()
{
    int n[6] = {15, 25, 35, 45, 50, 60};
    ofstream output ("numeri", ios::out | ios::binary);
    for (int i = 0; i < 6; i++) output.write ((char *)&n[i], 4);
    output.close();
    ifstream input ("numeri", ios::in | ios::binary);
    for (int i = 0; i < 6; i++) input.read ((char *)&n[i], 4);
    input.close();
    for (int i = 0; i < 6; i++) cout << n[i] << " ";
    return 0;
}
```

**Esecuzione**

15 25 35 45 50 60

Aprendo il file numeri con un editore esadecimale se ne può vedere il contenuto byte per byte; esso contiene i sei numeri interi scritti in binario in modalità "*little endian*", ovvero con l'ordine dei byte invertito (dal meno significativo al più significativo).

0F 00 00 00	19 00 00 00	23 00 00 00	2D 00 00 00	32 00 00 00	3C 00 00 00
15	25	35	45	50	60

**10.9 Accesso diretto**

Nel sistema di I/O del C++ si può accedere direttamente ai singoli byte dei flussi file tramite le funzioni `seekg()` e `tellg()` ereditate dalla classe `istream`.

L'accesso diretto fa uso di un *puntatore al file* (da non confondere con il già noto puntatore alla memoria) che è un semplice indice della *posizione corrente* (cioè del byte corrente) del file. La posizione corrente è il punto nel quale si effettuerà (o inizierà) il successivo accesso al file. Possiamo distinguere fra posizione corrente in lettura (*current get position*) e posizione corrente in scrittura (*current put position*). Il puntatore `get` specifica la posizione corrente in lettura, mentre il puntatore `put` specifica la posizione corrente in scrittura. Ogni volta che si realizza un'operazione di I/O, il puntatore corrispondente avanza automaticamente.

Le funzioni di accesso diretto a file `seekg()` e `tellg()` hanno la seguente sintassi:

```
istream& seekg (streampos pos);
istream& seekg (streamoff scostamento, seekdir ind);
streampos tellg();
```

La versione a un solo argomento di `seekg()` muove il puntatore `get` alla posizione assoluta `pos`, nel file, partendo da quella iniziale che ha indice 0. L'argomento `pos` è di tipo `streampos`, un `long` definito in `iostream`, come pure il primo argomento della seconda versione di `seekg()`. Il secondo argomento è del tipo enumerativo:

```
enum seekdir
{
    beg,      // inizio del file
    cur,      // posizione attuale
    end,      // fine del file
};
```

La funzione muove il puntatore `get` di *scostamento* byte:

- dalla posizione 0 in avanti se `ind` è `ios::beg`;
- dalla posizione corrente in lettura in avanti se `ind` è `ios::cur`;
- dalla fine del file indietro se `ind` è `ios::end`.

La funzione `tellg()` restituisce la posizione corrente in lettura oppure -1 in caso di errore.

#### Esempio 10.20

Supponiamo di avere questo file di testo:

*poesia*

```
Vaghe stelle dell'Orsa
io non credea tornar ancor per uso a contemplarvi
sul paterno giardino scintillanti
e ragionar con voi dalle finestre di quest'albergo
ove abitai fanciullo
e delle gioie mie vidi la fine.
```

```
#include <fstream>
int main()
{
    char nome_file[] = "poesia";
    ifstream file_in (nome_file);
    // determina la lunghezza del file
    streampos inizio = file_in.seekg(0, ios::beg).tellg();
    streampos fine   = file_in.seekg(0, ios::end).tellg();
    streamoff spostamento;
    cout << "introdurre uno spostamento (+) dall'inizio del file \""
        << nome_file << "\"" << endl << "la cui lunghezza è "
        << (fine-inizio) << " caratteri: ";
    cin  >> spostamento;
    file_in.seekg (spostamento, ios::beg); // va allo spostamento
    char car;
```

```

while (file_in)
{
    file_in.get(car);
    cout << car;
}
file_in.close();
}

```

### Esecuzione

introdurre uno spostamento (+) dall'inizio del file "poesia"  
 la cui lunghezza è 211 caratteri: 24  
 o non credea tornar ancor per uso a contemplarvi  
 sul paterno giardino scintillanti  
 e ragionar con voi dalle finestre di quest'albergo  
 ove abitai fanciullo  
 e delle gioie mie vidi la fine.

La classe ostream ha due metodi analoghi: seekp() e tellp(). La loro sintassi è:

```

ostream& seekp(streampos pos);
ostream& seekp(streamoff scostamento, seek_dir ind);
streampos tellp();

```

La versione a un solo argomento di seekp() muove il puntatore *put* alla posizione assoluta *pos* nel file. La versione a due argomenti sposta la posizione corrente in scrittura di *spost* byte secondo quanto specificato da *ind*. La funzione membro tellp() restituisce la posizione corrente in scrittura oppure -1 in caso di errore.

### Esempio 10.21

Il programma seguente è pensato per essere lanciato come un comando cambio da terminale. Esso prende come primo argomento il nome di un file e come secondo argomento un intero positivo che indica una posizione all'interno di quel file. Il comando apre il file, si posiziona in quel punto e vi scrive il carattere W alterando quindi il file.

```

int main(int argc, char *argv[])
{
    if(argc != 3) {
        cout << "Lanciare il comando: cambio <nome_file> <byte>\n";
        return 1;
    }
    fstream usc(argv[1], ios::in | ios::out | ios::binary);
    if (!usc) {
        cout << "Non si può aprire il file\n";
        return 1;
    }
}

```

```

    usc.seekp(atoi(argv[2]), ios::beg);
    usc.put('W');
    usc.close();
    return 0;
}

```

### Esecuzione di prova

\$: cambia

Lanciare il comando: cambia <nome\_file> <byte>

\$: cambia poesia 100

il file diventa:

poesia

Vaghe stelle dell'Orsa  
 io non credea tornar ancor per uso a contemplarvi  
 sul paterno giardino scinti lanti  
 e ragionar con voi dalle finestre di quest'albergo  
 ove abitai fanciullo  
 e delle gioie mie vidi la fine.

In questo capitolo sono state introdotte le operazioni base per gestire l'input e l'output. Abbiamo visto che la libreria `iostream` contiene quattro oggetti flusso: `cout`, `cin`, `cerr` e `clog`. L'output a schermo utilizza l'identificatore di flusso `cout` e l'inseritore `<<`, mentre l'input da tastiera utilizza l'identificatore di flusso `cin` e l'estrattore `>>`. Il metodo `get()` si utilizza per leggere singoli caratteri dall'input, mentre il metodo `put()` si utilizza per scrivere singoli caratteri sull'output. Per leggere una riga completa di input si utilizza la funzione `getline()`. Abbiamo anche visto che esistono dei manipolatori per

modificare il formato dei flussi di input e di output.

Per aprire e chiudere file si creano oggetti delle classi `ifstream` e `ofstream`. Per esempio, per scrivere su di un file si deve creare prima un oggetto `ofstream`, e poi associare quell'oggetto a uno specifico file. Per utilizzare oggetti `ofstream` bisogna includere l'header file `fstream` (che include `iostream`).

I flussi dovranno poi essere aperti e chiusi. Queste operazioni si realizzano con le funzioni `open()` e `close()`. Altre funzioni implicate nelle operazioni di lettura e scrittura sono `read`, `write`, `seekg` e `tellg`.

- Input/output
- Flusso (*stream*)
- Flusso binario
- Flusso di testo

- Manipolatore
- Operatore di estrazione
- Operatore di inserimento

## Esercizi

**Esercizio 1** Quali sono gli operatori di inserimento ed estrazione? A cosa servono?

**Esercizio 2** Spiegare le differenze tra `cin.get()` e `cin.getline()`.

**Esercizio 3** Scrivere un programma che scriva nei tre flussi standard: `cin`, `cout`, `cerr`.

**Esercizio 4** Scrivere un programma che utilizzi le funzioni `setf()`, `fill()` e `width()` e produca il seguente output formattato:

Capitolo 5 Classi .....	300
Capitolo 6 Ereditarietà .....	328
Capitolo 7 Flussi .....	333

**Esercizio 5** Scrivere il codice per ognuno dei seguenti casi.

- Stampare l'intero 12345 in un campo di 12 cifre allineate a sinistra.
- Stampare 3.14159 in un campo di 12 cifre con zeri precedenti.
- Leggere un intero in decimale e stamparlo in ottale.
- Leggere un intero in esadecimale e stamparlo in decimale.

**Esercizio 6** Aprire un file di testo, leggerlo riga per riga in un array di caratteri e scriverlo di nuovo in un altro file.

**Esercizio 7** Copiare un file, in modo binario, carattere per carattere.

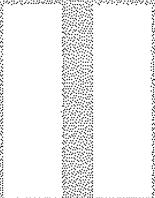
**Esercizio 8** Scrivere un programma che legga un testo dalla console e lo immagazzini in un nuovo file di testo con il nome `miofile.txt`. Il nuovo file deve avere la stessa struttura di righe del testo scritto sulla console. Inoltre, tutte le lettere minuscole debbono essere convertite in maiuscole.

**Esercizio 9** Si dispone di un file `telefono.txt`, con nomi e numeri di telefono ordinati in ordine alfabetico. Scrivere un programma che aggiunga un nuovo elemento in rubrica; il programma deve leggere un nome e un numero di telefono della nuova persona dalla console, poi inserire questa informazione nel posto corretto del file in modo che rimanga ordinato. Suggerimento: utilizzare un file temporale.

**Esercizio 10** Scrivere le istruzioni necessarie per aprire un file di caratteri i cui nome e accesso s'introducono dalla tastiera in modo lettura. Nel caso in cui il risultato dell'operazione sia erroneo, aprire il file in modo scrittura.

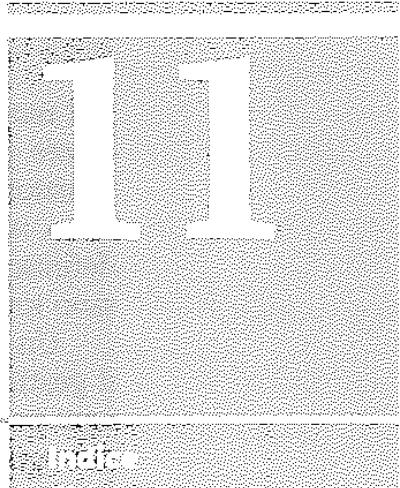
**Esercizio 11** Un file di testo contiene interi positivi e negativi. Utilizzare l'operatore di estrazione `>>` per leggere il file, visualizzarlo e determinare il numero di interi negativi e positivi che ha.

**PARTE**



# **Fondamenti di programmazione orientata agli oggetti**

# Classi e oggetti



Indice

- |                                       |   |
|---------------------------------------|---|
| <b>11.1</b> Classi e oggetti          | <b>11.5</b> Overloading<br>di funzioni membro     |
| <b>11.2</b> Definizione di una classe | <b>11.6</b> Errori frequenti di<br>programmazione |
| <b>11.3</b> Costruttori               |   |
| <b>11.4</b> Distruttori               |   |

## Introduzione

Fin qui abbiamo trattato gli argomenti tipici della programmazione strutturata, paradigma computazionale fondamentale che il C++ eredita dal C. Con questo capitolo inizia la parte relativa alla cosiddetta “*programmazione orientata agli oggetti*” introducendo il suo costrutto fondamentale, quello di **classe**. Una **classe** è un “tipo di dato” che contiene anche codice, cioè funzioni specifiche per quel tipo di dato. Essa

permette d’incapsulare dati e codice per gestire un oggetto del programma, come, per esempio, una finestra sullo schermo, un dispositivo collegato al computer o la figura geometrica di un programma di disegno. In questo capitolo vedremo come creare e utilizzare classi individuali, nei successivi vedremo come definire e utilizzare gerarchie e altre relazioni tra classi.

## 11.1 Classi e oggetti

Il paradigma computazionale noto con il nome di *Object Oriented Programming* (OOP) nacque nel 1969 dal norvegese Kristin Nygaard che, tentando di scrivere un programma che descrivesse il movimento delle navi attraverso un fiordo, capì che era molto difficile simulare le maree, i movimenti delle navi e le forme della linea di costa con i metodi di programmazione esistenti in quel momento. Scoprì che gli elementi dell’ambiente che tentava di modellare – navi, maree e linea di costa dei fiordi – e le azioni che ogni elemento poteva eseguire, costituivano delle relazioni che erano più facili da maneggiare.

La programmazione orientata agli oggetti si è evoluta molto da allora, ma la sostanza del modello computazionale è sempre la stessa: combinare le descrizioni degli elementi (i dati) con le azioni eseguibili su quegli elementi (le funzioni). Le classi e le loro istanze, cioè gli oggetti, sono gli elementi chiave su cui poggia l’OOP.

### 11.1.1 Cosa sono gli oggetti?

Nel mondo reale, le persone identificano gli oggetti come cose che possono essere percepiti mediante i sensi. Gli oggetti hanno proprietà specifiche: posizione, dimensione, colore, forma, consistenza ecc., che definiscono il loro stato, ma possono essere caratterizzati anche da certi *comportamenti*. L'OOP definisce un oggetto come qualsiasi cosa, reale o astratta, nella quale si possono immagazzinare dati, insieme con le operazioni che li manipoleranno.

Un programma orientato agli oggetti può gestirne parecchi. Per esempio, il gestionale di un esercizio di vendita al dettaglio utilizza un oggetto per ogni prodotto gestito nel magazzino. Il programma manipolerà gli stessi dati per ogni oggetto, includendo il numero di unità, la descrizione del prodotto, il prezzo, il numero di articoli nello *stock* e la data per il nuovo ordine.

Ogni oggetto sa anche come eseguire azioni con i propri dati. L'oggetto prodotto dal gestionale, per esempio, sa come creare sé stesso e stabilire i valori iniziali di tutti i suoi dati, come modificare i suoi dati e come valutare se vi sono articoli sufficienti nello *stock* o se bisogna programmare un nuovo ordine. Quindi un oggetto consiste di dati e di azioni che esso può compiere.

### 11.1.2 Cosa sono le classi?

Normalmente per classe si intende un insieme di oggetti che condividono una struttura e un comportamento. Nell'OOP una *classe* è un tipo definito dall'utente. Essa è il mattone fondamentale dell'OOP.

Una classe contiene la specifica dei dati che descrivono l'oggetto che ne fa parte, insieme alla descrizione delle sue funzioni. In C++ i dati si denominano *attributi*, mentre le funzioni si dicono *metodi* (dallo Smalltalk, uno dei primi linguaggi orientati agli oggetti).

Le classi possono separare l'interfaccia dall'implementazione; solo il programmatore della classe conoscerà i dettagli implementativi, l'utilizzatore conoscerà l'interfaccia, ovvero l'insieme delle funzionalità che il programmatore della classe decide di "esporre" nascondendo i dettagli implementativi.

Le classi sono esempi di *Abstract Data Type*. Un *tipo di dato astratto* tratta dati e operazioni sui dati come un'entità unica: si descrive in modo astratto quello che fa la classe invece che specificare come funziona la classe internamente. La *classe* è cioè il mezzo naturale per tradurre l'astrazione di un tipo che combina la rappresentazione dei dati (*attributi*) con le funzioni (*metodi*) che manipolano quei dati in un'unica entità. La collocazione di dati e funzioni in una sola entità (la classe) è l'idea centrale dell'OOP. La Figura 11.1 mostra la rappresentazione grafica di una classe.

## 11.2 Definizione di una classe

La definizione di una classe si compone di:

- *dichiarazione*: descrive i dati e l'interfaccia (cioè le "funzioni membro", anche dette "metodi");
- *definizioni dei metodi*: descrive l'implementazione delle funzioni membro.

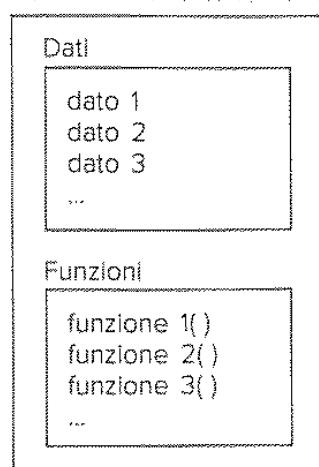


Figura 11.1

Le classi contengono dati e funzioni.

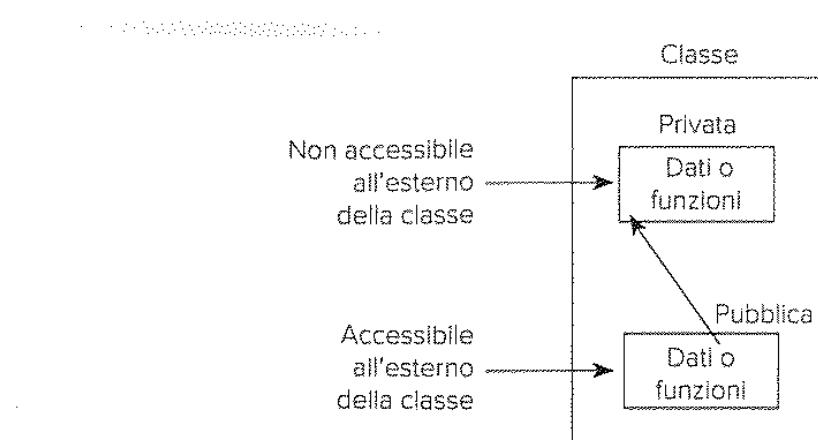
La dichiarazione di una classe utilizza la parola riservata `class`, seguita dall'elenco dei dati e dei metodi della classe, racchiusi tra parentesi graffe e divisi in sezioni.

### Scrivere

```
class NomeClasse
{
    public: dichiara i membri pubblici (per poterli accedere dall'esterno)
    attributi; memoria per i dati
    metodi; funzioni che operano sui dati
    protected: memoria e funzioni visibili solo da classi derivate
    attributi;
    metodi;
    private: memoria e funzioni visibili solo all'interno della classe
    attributi;
    metodi;
}; Punto e virgola obbligatorio
```

Per default, i membri di una classe sono nascosti all'esterno, cioè i suoi dati e i suoi metodi sono *privati*. È possibile comunque controllare la *visibilità* esterna mediante gli specificatori d'accesso: `public`, `protected` e `private`. La sezione specificata come `public` contiene membri a cui si può accedere dall'esterno della classe, mentre la sezione `private` contiene membri ai quali si può accedere solo dall'interno della classe. Ai membri che seguono lo specificatore `protected` si può accedere anche da metodi di classi *derivate* della stessa (concetti che approfondiremo nel prossimo capitolo).

Questa caratteristica della classe si chiama *occultamento di dati (information hiding)* ed è una proprietà dell'OOP. La Figura 11.2 mostra graficamente il funzionamento delle sezioni pubblica e privata di una classe.

**Figura 11.2**

Sezioni pubblica e privata di una classe.

Una classe potrebbe essere definita anche con la parola riservata `struct`, ma in questo caso tutti i membri della struttura sarebbero visibili dall'esterno al fine di mantenere la compatibilità con C. Questa è l'unica differenza esistente in C++ tra le parole riservate `class` e `struct`.

**Esempio 11.1**

Dichiarazione di una classe Semaforo

```

class Semaforo
{
public:
    void mostrareColore(colore);
private:
    enum colore {VERDE, ROSSO, GIALLO};
    void cambiareColore(colore);
};
  
```

La sezione pubblica (interfaccia) dichiara la definizione dei servizi (mostrare il colore del semaforo), e la sezione privata i dettagli interni del semaforo (cambiare i colori del semaforo, definizione dei colori).

1. Le dichiarazioni dei metodi (cioè le intestazioni delle funzioni), normalmente, si collocano nella sezione pubblica e le dichiarazioni dei dati (attributi), normalmente, si mettono nella sezione privata.
2. È indifferente collocare prima la sezione pubblica o quella privata; è però consigliabile collocare la sezione pubblica prima per mettere in evidenza le operazioni che fanno parte dell'interfaccia utente pubblica.
3. Le parole chiave `public` e `private`, seguite da due punti, segnalano l'inizio delle rispettive sezioni pubbliche e private; benché non sia comune, una classe può avere varie sezioni pubbliche e private.

### Controllo di accesso ai membri: pubblico o privato?

Le funzioni membro che costituiscono l'interfaccia della classe vanno quindi nella sezione pubblica, altrimenti non potrebbero essere invocate dal programma. Possono però esserci funzioni membro private, per gestire dettagli interni che non fanno parte dell'interfaccia pubblica. In generale si potrebbe dire che i dati sono privati (per proteggerli da manipolazioni accidentali da parte del programma) mentre le funzioni sono pubbliche (per poterle mandare in esecuzione dall'esterno della classe), tuttavia non è una regola stretta e in alcune circostanze si può aver bisogno di utilizzare funzioni private e dati pubblici.

#### Esempio 11.2

Diverse classi con differenti sezioni

```
class Prodotto
{
    private:
        float prezzo;
        char nome[20];
    public:
        void calcolareIVA(int);
};

class GiornoDellAnno
{
    public:
        void leggere();
        void scrivere();
        int mese;
        int giorno;
};
```

La parola riservata `private` non sarebbe necessaria perché il controllo d'accesso è privato per default, tuttavia utilizzeremo spesso l'etichetta `private` al fine di evidenziare bene l'occultamento.

Per utilizzare una classe bisogna chiedersi tre cose:

1. quale è il suo nome;
2. dove è definita;
3. che operazioni supporta.

Il *nome* della classe, per esempio, *Semaforo*, è probabilmente definito in un header file che ha nome e suffisso (estensione). Normalmente il nome del file è lo stesso nome della classe. L'estensione normalmente è `.h`, nell'esempio precedente il file si chiamerebbe *Semaforo.h*.

Eccetto alcuni casi speciali che vedremo più avanti, nella dichiarazione della classe si collocano solo le intestazioni dei metodi. Le loro definizioni si collocano, normalmente, in un file a parte chiamato *file di implementazione* il cui nome è lo stesso dell'header file ma con l'estensione `.cpp` invece che `.h`. Nell'esempio *Semaforo.cpp*, i due file costituiscono una libreria di classe, e i programmi che utilizzano la classe definita nella libreria si dicono *programmi cliente*. Questi ultimi dovranno includere l'intestazione della libreria con la direttiva `#include`:

```
#include "Semaforo.h"
```

**Nota**

La differenza fra

```
#include <iostream>
```

e

```
#include "Semaforo.h"
```

è che nel primo caso si indica al compilatore C++ che `iostream` è una libreria standard, mentre nel secondo si indica che la libreria è definita dal programmatore.

**Esempio 11.3**

Definizione di una classe chiamata *Punto* (dichiarata nel file *Punto.h*) che contiene le coordinate *x* e *y* di un punto in un piano.

```
class Punto {
public:
    int Leggere_x();           // restituisce il valore di x
    void Assegnare(int,int);   // assegna coordinate
private:
    int x;                     // coordinata x
    int y;                     // coordinata y
};
```

La definizione di una classe non riserva spazio in memoria. L'allocazione avviene quando si crea un oggetto di una classe (*istanza* di una classe).

**Esempio 11.4**

Definizione della classe *eta* (dichiarata nel file *eta.h*)

```
class eta
{
private:
    int etaFiglio, etaMadre, etaPadre; // attributi
public:
    void iniziare(int, int, int);      // metodi
    int leggereFiglio();
    int leggereMadre();
    int leggerePadre();
};
```

Sottolineiamo che l'accesso per default è privato. Nella seguente classe *Studente*, per esempio, tutti i dati sono privati, mentre i metodi sono pubblici:

```
class Studente {
```

```
    long numId;
```

```

char nome[40];
int eta;

public:
    long LeggereNumId();
    char * LeggereNome();
    int LeggereEta();

};

```

Uno specificatore d'accesso può apparire più di una volta in una definizione di classe, ma questo la renderà più difficile da leggere. Lo specificatore si applica infatti ai membri che vengono dopo di lui fino al successivo specificatore o fino alla fine della classe.

```

class Studente{
private:
    long numId;
public:
    long LeggereNumId();
private:
    char nome[40];
    int eta;
public:
    char * LeggereNome();
    int LeggereEta();

};

```

#### 11.2.1 Oggetti

Gli oggetti sono istanze della classe (così come le variabili sono istanze di un tipo di dato):

```

nome_classe identificatore_oggetto;

```

Così, la definizione di un oggetto Punto è:

```

Punto P;
Semaforo S1;

```

Quindi un oggetto sta alla sua classe come una variabile sta al suo tipo. Un oggetto ha quindi membri funzione e membri dato. L'OOP concepisce i programmi come collezioni di oggetti interagenti. Gli oggetti possono interagire poiché sono capaci di invocare le funzioni membro (pubbliche) degli altri.

Quello che nelle struct era l'operatore di accesso al campo, ovvero il punto (.), qui diventa l'operatore *di accesso* al membro. Le seguenti istruzioni, per esempio, creano un punto p1, che fissa la sua coordinata x e poi la visualizza.

```

Punto p1;
p1.Assegnare(50,100)
cout << "L'ascissa di p1 è " << p1.Leggerex();

```

Così come le strutture, anche gli oggetti possono essere assegnati; il C++ fa una copia bit a bit di tutti i membri. In altre parole, tutti i membri fisicamente contenuti nell'area dati dell'oggetto originale vengono copiati nell'oggetto destinatario. Per esempio, il seguente codice crea un punto (Punto) chiamato P2 e lo inizializza con il contenuto di P:

```
Punto p2;
p2 = p1;
```

### 11.2.2 Attributi e metodi

Gli attributi (o dati membro) possono essere di qualunque tipo valido, con eccezione del tipo della classe che si sta definendo (poiché incompleto); è possibile tuttavia definire membri puntatori o riferimenti al tipo della classe che si sta definendo. Possono essere dichiarati costanti. Se un dato è const il suo valore, stabilito durante la costituzione dell'oggetto, non può essere modificato.

Non c'è limite al numero di attributi in una classe.

I metodi possono essere sia dichiarati sia definiti all'interno delle classi. Così come quella di una qualunque funzione, la definizione di un metodo consiste di quattro parti:

- il tipo di ritorno della funzione;
- il nome della funzione;
- la lista dei parametri formali (eventualmente vuota) separati da virgole;
- il corpo della funzione racchiuso tra parentesi graffe.

Le tre prime parti costituiscono il prototipo della funzione che *deve essere* definito dentro la classe, mentre il corpo della funzione può essere definito anche altrove.

#### Esempio 11.5

Dichiarazione di una classe Articolo\_Vendite.

```
class Articolo_Vendite {
public:
    double prezzo_medio(); // prototipo
    bool articolo_uguale (const Articolo_Vendite & art) //definizione
        {return 0.2;}
private: // membri privati
    // ...
};
```

In questo esempio la funzione articolo\_uguale è definita dentro la classe, mentre la funzione prezzo\_medio è dichiarata dentro la classe ma definita altrove. I metodi definiti dentro la classe si considerano implicitamente *funzioni in linea*, quindi si crea una copia del codice per ogni oggetto di quella classe.

La definizione dei metodi (ovvero la specifica del loro codice) se è fatto altrove (non dentro la definizione della classe) deve contenere il riferimento

alla classe alla quale appartengono mediante l'operatore di risoluzione di visibilità (::). Per esempio, la definizione della classe Semaforo conteneva il prototipo del metodo mostrareColore(colore); in altro punto del codice sarà definito il metodo.

#### Esempio 11.6

```
void Semaforo::mostrareColore(colore c)
{
    switch (c)
    {
        case VERDE: cout << "Puoi passare! "; break;
        case ARANCIO: cout << "Affrettati! "; break;
        case ROSSO: cout << "STOP! "; break;
    }
};
```

Se il codice è semplice il metodo può essere definito dentro la classe.

#### Esempio 11.7

La classe Punto definisce le coordinate di un punto in un piano. Per ogni coordinata fornisce un metodo per determinarla e un altro per restituirla.

```
class Punto {
public:
    int LeggeX() { return x; }
    int LeggeY() { return y; }
    void AssegnaX (int valx) { x = valx; }
    void AssegnaY (int valy) { y = valy; }
private:
    int x;
    int y;
};
```

### 11.2.3 Chiamate a funzioni membro

I metodi di una classe si invocano come si accede ai dati di un oggetto, ovvero tramite l'operatore punto (.).

#### Esempio 11.8

```
class Demo
{
private:
// ...
public:
    void funz1 (int P1) {...}
    void funz2 (int P2) {...}
```

```

};

int main()
{
    ...
    Demo d1, d2;      // definizione degli oggetti d1 e d2
    ...
    d1.funz1(2021);
    d2.funz1(2022);
}

```

`funz1()` è un metodo della classe `Demo`, quindi la si deve sempre chiamare in connessione a un oggetto di quella classe. Non avrebbe senso una chiamata diretta `funz1(2005)`, e infatti essa produrrebbe un errore. Un metodo si invoca per lavorare su di uno specifico oggetto, e non sulla classe in generale. Per utilizzare una funzione membro, l'operatore di accesso collega la funzione membro all'oggetto. Alcuni denominano "messaggi" le invocazioni alle funzioni membri:

`d1.funz1();`

in quanto queste invocazioni sono viste come messaggi inviati all'oggetto `d1` perché esso esegua il compito `funz1()`.

#### Esempio 11.9

Definire una classe che rappresenti una radio.

```

class radio {
private:
    float frequenza;
    int volume;
public:
    void Accende();
    void AumentaFrequenza();
    void DiminuisceFrequenza();
    void AbbassaVolume();
    void AlzaVolume();
};

void radio::Accende()
{
    frequenza = 99.99;
    volume = 45;
}

void radio::AlzaVolume()
{
    volume++;
}

void radio::AbbassaVolume()
{
    volume--;
}

```

```

    volume--;
}
void radio::DiminuisceFrequenza()
{
    frequenza--;
}
void radio::AumentaFrequenza()
{
    frequenza++;
}
int main ()
{
    radio miaStazione;
    miaStazione.Accende();
    miaStazione.AlzaVolume();
    miaStazione.AbbassaVolume();
    miaStazione.AumentaFrequenza();
    miaStazione.DiminuisceFrequenza();
}

```

#### 11.2.4 Tipi di metodo

Possiamo classificare i metodi a seconda del tipo di operazione che eseguono.

- *Costruttori e distruttori*: si invocano automaticamente rispettivamente alla creazione e alla distruzione di un oggetto.
- *Selettori (get)*: restituiscono valori di membri dato.
- *Modificatori (set)*: modificano valori di membri dato.
- *Operatori*: definiscono operatori standard C++.
- *Iteratori*: elaborano collezioni di oggetti come gli array.

#### 11.2.5 Funzioni in linea e fuori linea

Come anticipato, i metodi definiti nella classe sono funzioni *in linea*. Per funzioni grandi è preferibile codificare nella classe solo il prototipo della funzione. Ciò permette anche di nascondere l'implementazione della funzione fornendo solo il codice sorgente dell'header file e un file di implementazione della classe precompilata.

#### Esempio 11.10

Definire una classe GiornoAnno che contenga gli attributi *mese* e *giorno* (come numeri interi) e un metodo Visualizzare. Scrivere un programma che faccia uso della classe e visualizzi il suo output.

```

class GiornoAnno
{
    public:
        void visualizzare(){
            cout << "\nmese = " << mese << "\ngiorno = " << giorno << endl;
        }
        int mese;
        int giorno;
};

int main() // programma che usa GiornoAnno
{
    GiornoAnno oggi, compleanno;
    cout << "Introduca data del giorno di oggi\n";
    cout << "Numero del mese : ";
    cin >> oggi.mese;
    cout << "Giorno del mese : ";
    cin >> oggi.giorno;
    cout << "Introduca la sua data di nascita:\n";
    cout << "Numero del mese : ";
    cin >> compleanno.mese;
    cout << "Giorno del mese : ";
    cin >> compleanno.giorno;
    cout << "La data di oggi è ";
    oggi.visualizzare();
    cout << "La sua data di nascita è ";
    compleanno.visualizzare();
    if(oggi.mese == compleanno.mese && oggi.giorno == compleanno.giorno)
        cout << "\nBuon compleanno! \n";
    else
        cout << "\nBuon giorno! \n";
    return 0;
}

```

### Esecuzioni di prova

---

Introduca data del giorno di oggi  
 Numero del mese : 6  
 Giorno del mese : 22  
 Introduca la sua data di nascita:  
 Numero del mese : 6  
 Giorno del mese : 22  
 La data di oggi è  
 mese = 6  
 giorno = 22  
 La sua data di nascita è  
 mese = 6  
 giorno = 22

Buon compleanno!

---  
Introduca data del giorno di oggi

Numero del mese : 1

Giorno del mese : 22

Introduca la sua data di nascita:

Numero del mese : 6

Giorno del mese : 22

La data di oggi è

mese = 1

giorno = 22

La sua data di nascita è

mese = 6

giorno = 22

Buon giorno!

#### 11.2.6 La parola riservata inline

La decisione di scegliere tra funzioni in linea e fuori linea è dettata dall'efficienza. Una funzione in linea è normalmente più veloce poiché non si deve eseguire l'istruzione `call` per chiamarla né eseguire l'istruzione `return` per ritornare al programma chiamante. In realtà la definizione di un metodo dentro una classe non garantisce che il compilatore lo renda realmente in linea; questa è una decisione che ogni compilatore C++ prende autonomamente in base alle istruzioni dentro la funzione. Un metodo definito fuori dal blocco della sua classe può beneficiare dei vantaggi delle funzioni in linea se è preceduto dalla parola riservata `inline`:

```
inline void Punto::AssegnareX (int valx)
{
    x = valx;
}
```

A seconda del compilatore, le funzioni che utilizzano la parola riservata `inline` possono essere collocate nello stesso header file della definizione della classe.

### 11.3 Costruttori

Sarebbe bene che un oggetto possa essere inizializzato automaticamente all'atto della sua creazione, senza dover effettuare una successiva chiamata a una sua qualche funzione membro. Un *costruttore* è appunto un metodo che viene automaticamente eseguito all'atto della creazione di un oggetto per inizializzare i suoi membri dato; si caratterizza per due fatti:

1. ha lo stesso nome della propria classe;
2. non restituisce alcun valore, neanche `void`.

**Esempio 11.11**

Questa classe *Rettangolo* ha un costruttore che accetta in input quattro parametri.

```
class Rettangolo
{
    private:
        int Sinistro;
        int Superiore;
        int Destro;
        int Inferiore;
    public:
        Rettangolo(int Sin, int Sup, int Des, int Inf); // Costruttore
        .... // definizioni di altre funzioni membro
};
```

Quando si definisce un oggetto, si passano i valori dei parametri al costruttore utilizzando la sintassi di una normale chiamata di funzione:

```
Rettangolo rect(25, 75, 25, 75);
```

Questa definizione crea un'istanza dell'oggetto *Rettangolo* e invoca il costruttore della classe passandogli i parametri con valori specifici.

Come caso particolare, si può anche passare i valori dei parametri al costruttore quando si crea l'istanza di una classe utilizzando l'operatore new:

```
Rettangolo* Nrect = new Rettangolo(25, 75, 25, 75);
```

L'operatore new invoca automaticamente il costruttore dell'oggetto creato.

Un costruttore che non ha parametri si chiama *costruttore di default* e normalmente inizializza gli attributi assegnandogli valori di default.

**Esempio 11.12**

Questo costruttore di default inizializza *x* e *y* a 0.

```
class Punto {
    public:
        Punto() // costruttore
    {
        x = 0;
        y = 0;
    }
    private:
        int x;
        int y;
};
```

Un costruttore con parametri si dice invece *costruttore alternativo*.

```
Punto P(50, 250); // definisce e inizializza P
```

e si può chiamare esplicitamente:

```
Punto x = Punto(50, 250); // costruisce un punto e lo assegna a x
```

Un altro costruttore alternativo della classe Punto, che visualizza un messaggio dopo essere chiamato, è:

```
Punto::Punto(int valx, int valy)
{
    AssegnareX(valx);
    AssegnareY(valy);
    cout << "Eseguito il costruttore di Punto.\n";
}
```

Un costruttore, così come qualunque altro metodo - a eccezione dei distruttori, che vedremo nel prossimo paragrafo - può essere sovraccaricato al pari di una qualsiasi funzione. I costruttori sovraccaricati forniscono modi alternativi per inizializzare nuovi oggetti di una classe. Ovviamente, indipendentemente da quanti essi siano, all'atto della creazione di un oggetto viene eseguito un solo costruttore.

#### Esempio 11.13

```
class Punto {
public:
    Punto(); // costruttore di default
    Punto(int valx, int valy); // costruttore sovraccaricato
    // ...
private:
    int x;
    int y;
};
```

La definizione di un oggetto Punto può chiamare qualunque costruttore:

```
Punto P; // chiamata al costruttore default
Punto Q(25,50); // chiamata al costruttore alternativo
```

*Un costruttore di copia* è un costruttore, creato automaticamente dal compilatore, che viene chiamato quando si passa un oggetto per valore a una funzione (si costruisce una copia dell'oggetto locale alla funzione). Il costruttore di copia si invoca anche quando si definisce un oggetto inizializzandolo a un altro oggetto dello stesso tipo. Per esempio:

```
Punto P;
QualcheFunzione(P);
Punto Q = P;
```

Per default, in questi casi C++ costruisce una copia bit a bit dell'oggetto.

Il costruttore inizializza i membri dato utilizzando *espressioni di assegnamento*. Tuttavia, costanti e riferimenti non possono essere assegnati ma solo inizializzati. Per risolvere questo problema, C++ fornisce una speciale caratteristica, nota come *vettore inizializzatore di membri*, che permette di *inizializzare* (invece che assegnare) uno o più membri dato. Esso si colloca immediatamente dopo la lista dei parametri, nella definizione del costruttore, e consiste del carattere due punti (:) seguito da uno o più *inizializzatori di attributi*, separati da virgole. Un inizializzatore di attributo consiste nel nome di un membro dato seguito da un valore iniziale tra parentesi tonde.

#### Esempio 11.14

```
class miaClasse {
private:
    int T;
    const int CInt;
    int& Dint;
    // ...
public:
    miaClasse (int Param): T (Param), CInt(25), Dint(T)
    {
        // codice del costruttore
    }
    // ...
};
```

La definizione seguente crea un oggetto

```
miaClasse Oggetto(0);
```

con gli attributi T e CInt inizializzati rispettivamente a 0 e 25, e l'attributo DInt inizializzato in modo da riferirsi a T.

#### Esempio 11.15

*Progettare una classe contatore (ogni volta che si produce un evento il contatore si incrementa di 1). Il contatore deve poi poter essere consultato.*

```
class Contatore {
private:
    unsigned int cont;
public:
    Contatore() {cont = 0;} // costruttore
    void inc_cont() {cont++;} // conta
    int legge_cont() {return cont;} // restituisce valore di cont
};
int main()
{
    Contatore c1, c2; // definisce e inizializza
```

```

cout << "\nc1 = " << c1.legge_cont();
cout << "\nc2 = " << c2.legge_cont();

c1.inc_cont(); // incrementa c1
c2.inc_cont(); // incrementa c2
c2.inc_cont(); // incrementa c2

cout << "\nc1 = " << c1.legge_cont(); // visualizza di nuovo
cout << "\nc2 = " << c2.legge_cont();
}

```

La classe Contatore ha un attributo cont, del tipo unsigned int. Ha tre metodi: Contatore(), inc\_cont(), che aggiunge 1 a cont; legge\_cont(), che restituisce il valore attuale di cont.

## 11.4 Distruttori

Nella classe si può definire anche una funzione membro speciale, nota come *distruttore*, che viene chiamata automaticamente quando si elimina un oggetto con lo scopo di recuperarne la memoria occupata. Il distruttore ha lo stesso nome della sua classe preceduto dal carattere ~. Anche il distruttore non ha tipo di ritorno (neanche void) ma, al contrario del costruttore, non accetta parametri e non ve ne può essere più di uno nella classe (non è sovraccaricabile).

### Esempio 11.16

```

class Demo
{
private:
    int dati;
public:
    Demo() {dati = 0;}          // costruttore
    ~Demo() {}                 // distruttore
};

```

### INFO

I distruttori non hanno né valore di ritorno né argomenti.

Se non si dichiara esplicitamente un distruttore, C++ ne crea automaticamente uno vuoto.

Quando viene chiamato il distruttore di un oggetto?

- Se l'oggetto ha *visibilità locale*, il suo distruttore viene invocato quando il flusso del programma esce dal suo blocco di definizione.
- Se l'oggetto ha *visibilità di file*, il suo distruttore viene invocato quando termina il programma principale (`main`).
- Se l'oggetto è stato *creato dinamicamente* utilizzando `new`, il suo distruttore entra in gioco quando si invoca l'operatore `delete`.

**Esempio 11.17**

```
// questo distruttore notifica quando viene distrutto l'oggetto
// della classe Punto
class Punto {
public:
    ~Punto()
    {
        cout << "Chiamato il distruttore di Punto \n";
    }
    // ...
};
```

**11.4.1 Classi composte**

Così come le struct, anche le class possono essere *composte*. Prima del corpo di un costruttore di una classe composta, si debbono definire le classi membro della stessa. Per esempio, la seguente classe Studente contiene membri dato di tipo Dossier e Indirizzo:

```
class Dossier {
// ...
};

class Indirizzo {
// ...
};

class Studente {
public:
    Studente()
    {
        DareId(0);
        DareVotoMedio(0.0);
    }
    void DareId(long);
    void DareVotoMedio(float);
private:
    long id;
    Dossier dos; // attributo classe
    Indirizzo ind; // attributo classe
    float VotoMedio;
};
```

Anche se Studente contiene membri di tipo Dossier e Indirizzo, il costruttore di Studente non ha accesso ai membri privati o protetti di Dossier o Indirizzo. Il distruttore ~Studente sarà eseguito prima dei distruttori di Dossier e Indirizzo. In altre parole, l'ordine delle chiamate a distruttori di classi composte è esattamente il reciproco dell'ordine delle chiamate ai costruttori.

## 11.5 Overloading di funzioni membro

Come tutte le funzioni, anche i metodi di una classe possono essere sovraccaricati, ma *soltanto nella loro propria classe*, con le stesse regole utilizzate per sovraccaricare funzioni ordinarie: due funzioni membro sovraccaricate non possono avere lo stesso numero e tipo di parametri. L'*overloading* permette cioè di utilizzare uno stesso nome per più metodi che si distingueranno solo per i parametri passati all'atto della chiamata.

Per esempio, nella seguente classe Prodotto appaiono diverse funzioni membro sovraccaricate:

```
class Prodotto {
public:
    int prodotto (int m, int n);           // metodo 1
    int prodotto (int m, int p, int q);    // metodo 2
    int prodotto (float m, float n);       // metodo 3
    int prodotto (float m, float n, float p); // metodo 4
}
```

ma come abbiamo già visto, sono i costruttori i metodi che vengono più spesso sovraccaricati.

## 11.6 Errori frequenti di programmazione

La facilità nell'uso della struttura *classe* la rende incline a una quantità di possibilità di errori.

1. La parola riservata *class* non è necessaria per definire oggetti. Si può scrivere

```
class C {
    ...
};

C c1, c2;           // definizioni di oggetti
```

invece di

```
class C c1, c2;     // non è obbligatorio class
```

2. *Uso di costruttori e distruttori.*

Non si può specificare un tipo di ritorno in un costruttore o distruttore, né nella sua dichiarazione né nella sua definizione. Se C è una classe, allora non sarà legale scrivere:

```
void C::C I(int n)      // ERRORE
{
    ...
}
```

3. Non si possono neanche specificare argomenti, nemmeno void, nella dichiarazione o definizione del distruttore di una classe.

4. Inizializzatori di membri.

```
class C {           // ERRORE
    int x = 5;
};

class C {           // CORRETTO
    int x;
    ...
public:
    C() {x = 5;}
};
```

5. Non si può accedere a un membro dato private dal di fuori della sua classe (eccetto che per il tramite di una funzione amica friend che vedremo poi).

```
class C {
    int x;
};

int main()
{
    C c;
    c.x = 10;           // ERRORE, x è privato in C
    ...
}
```

6. Non si può accedere a un membro dato protected dal di fuori della sua gerarchia di classi (eccetto che per il tramite di una funzione amica friend che vedremo poi); per esempio:

```
class C {           // classe base
protected:
    int x;
};

class D: public C { // classe derivata, vedremo poi
    ...
};

int main()
{
    C c1;
    c1.x = 10         // ERRORE, x è protetto
    ...
}
```

contiene un errore, dato che x è accessibile solo da metodi di C e funzioni amiche di classi derivate da C, come D.

7. Non si può sovraccaricare alcuno di questi operatori:

. .\* :: ?: sizeof

8. Non si può utilizzare `this` (vedremo poi) come parametro, poiché è una parola riservata. Non si può neanche utilizzare in un'istruzione di assegnamento del tipo

```
this = ...; // this è costante; ERRORE
```

dato che `this` è una costante.

9. Un costruttore di una classe `C` non può avere un argomento di un tipo `C`, ma può avere un argomento di tipo riferimento, `C&`. Questo costruttore è il costruttore di copia.

```
class C {
    ...
};

C::C(C c)           // ERRORE
{
    ...
}

C::C(C& c)         // CORRETTO
{
    ...
}
```

10. Un membro dato static si può dichiarare ma non definire dentro la dichiarazione della classe.

```
class C {
    static int x = 7;      // ERRORE
    ...
};

class D {
    static int x;
};

// definizione con inizializzazione
int D::x = 7;
```

11. I costruttori o distruttori non si possono dichiarare static.

12. *Mancanza di punto e virgola nella definizione di classe.* Le parentesi graffe `{}` sono frequenti in C++, e, normalmente, non si mette il punto e virgola dopo quella chiusa. Tuttavia, la definizione di class termina sempre con `};`

```
class Prodotto
{
    public:
        // ...
    private:
        //...
} // dimenticanza del punto e virgola
```

13. *Inizializzazione parziale dei campi in un costruttore.* Il costruttore dovrebbe assicurare che tutti gli attributi siano inizializzati.

```
class Impiegato {
public:
    Impiegato();
    Impiegato(string n); // dimentica il parametro salario
    ...
private:
    string nome;
    float salario;
};
```

14. *Mancata inclusione del necessario header file della classe.* Questa mancanza genera tipicamente numerosi messaggi di errore del tipo:

undefined symbol

15. *Errore nella definizione del metodo di una classe:*

- a) errore nel riferimento al nome della sua classe e/o nell'uso dell'operatore di risoluzione di visibilità (::);
- b) scrittura errata del nome della funzione;
- c) omissione completa della definizione della funzione della classe.

In tutti i casi, il risultato è lo stesso: un messaggio di errore del compilatore indicante che la funzione chiamata non è nella classe indicata.

16. *Dimenticanza di punti e virgola in prototipi e intestazioni di funzioni.* L'omissione di un punto e virgola alla fine del prototipo di una funzione può produrre il messaggio di errore

«Statement missing»

oppure

«Declaration terminated incorrectly».

#### Riassunto

In questo capitolo abbiamo iniziato a conoscere un nuovo paradigma computazionale, quello della programmazione orientata agli oggetti, che ha decisamente affiancato la programmazione strutturata a partire dagli anni Novanta. L'elemento fondamentale di questo paradigma è il concetto di "oggetto", rappresentato in C++ dalla struttura denominata "classe", tipo di dato definito dall'utente che ha due componenti: un insieme di

attributi (o "membri dato") e un insieme di metodi (o "funzioni membro") che operano su quegli attributi. Un "oggetto" è poi un'istanza di una classe. La dichiarazione di una classe si compone di tre sezioni: public, private e protected. La sezione pubblica contiene dichiarazioni di attributi e metodi accessibili agli utenti dell'oggetto. La sezione privata contiene attributi e metodi nascosti agli utenti dell'oggetto e accessibili solo dalle fun-

zioni membro dell'oggetto stesso. Nessuna sezione è necessaria. L'accesso ai membri di una classe è **private** per default. Una funzione definita dentro una classe è implicitamente **inline**. Per definire una funzione membro esternamente alla definizione della classe si utilizza l'operatore di risoluzione di visibilità **::**.

Abbiamo visto poi l'importanza del **costruttore**, funzione membro con lo stesso nome della sua classe. Un costruttore non può restituire alcun tipo ma può

essere sovraccaricato. Esso si invoca automaticamente alla creazione di un oggetto per inizializzare gli attributi di un oggetto. Gli argomenti di default rendono il costruttore più utile e flessibile. La creazione di un oggetto corrisponde all'**istanziazione** di una classe. Da ultimo abbiamo visto che il **distruttore** è una funzione membro speciale che si chiama automaticamente per distruggere un oggetto della classe.

Soluzioni degli esercizi sul sito web [www.mheducation.it](http://www.mheducation.it)

- Analisi e progettazione orientata agli oggetti
- Concetto di "classe"
- Concetto di "oggetto"
- Costruttori
- Distruttori
- Incapsulamento

- Specificazioni di accesso **public**, **protected**, **private**
- Funzioni **const**
- Funzioni membro
- Membri dato
- *Information hiding*

## Esercizi

**Q&A** Dov'è l'errore nella seguente dichiarazione di classe?

```
union float_byte {
private:
    char mantissa[3], char esponente ;
public:
    float num;
    char esp();
    char mant();
};
```

**Q&A** Cosa non va nella seguente definizione di classe?

```
#include <cstring>
struct buffer {
    char dati[255];
    int cursore ;
    void iniziare(char *s);
```

```
    inline int Lung();
    char *contenuto();
};

void buffer::iniziare(char *s)
{
    strcpy(dati,s) ; cursor = 0;
}
char *buffer::contenuto(void)
{
    if(Lung()) return dati; else return 0;
}

inline int buffer::Lung(void)
{return cursor;}
```

**Q&A** Definire una classe che implementi una pila di 100 caratteri tramite un array. Le funzioni membro della classe debbono essere:

inserire, rimuovere, stack\_vuoto e stack\_pieno.

**Esercizio 1** Individuare gli eventuali errori nella seguente dichiarazione di classe:

```
class punto {
    int x, y;
    void punto(int x1, int y1){x = x1; y = y1;}
};
```

**Esercizio 2** È legale il seguente main ()?

```
class punto {
public:
    int x, int y;
    punto(int x1, int y1) {x = x1; y = y1;}
};
int main()
{
    punto(25, 15); //è legale quest'istruzione?
}
```

**Esercizio 3** Avendo risposto alla precedente domanda, quale sarà l'output del seguente programma?

```
class punto{
public:
    int x, int y;
    punto(int x1, int y1) {x = x1; y = y1;}
    punto(int z) {punto (z, z);}
};
int main()
{
    punto p(25);
    cout << "x = " << p.x << ", y = " << p.y << endl;
}
```

**Esercizio 4** Classificare i seguenti costruttori, i cui prototipi sono:

```
rect::rect(int a, int b);
nodo::nodo();
calcolatrice::calcolatrice(float *val i = 0.0);
stringa::stringa(stringa &c);
abc::abc(float f);
```

**Esercizio 5** Data la seguente dichiarazione di classe, perché non si può costruire l'array?

```
class punto {
public:
    int x, y;
    punto(int a, int b) {x = a; y = b;}
};
punto poligono[8]; // è valida quest'istruzione?
```

**Esercizio 6** Cambiare il costruttore dell'esercizio precedente in un costruttore di default. Si può costruire, in questo caso, l'array?

**Esercizio 7** Quale costruttore viene attivato nel seguente esempio?

```
class prova_totale {
private:
    int num;
public:
    prova_totale() {num = 0;}
    prova_totale(int n = 0) {num = n;}
    int valore() {return num;}
};
```

prova\_totale prova;

**Esercizio 8** Creare una classe ora che abbia membri dati di tipo int per ore, minuti e secondi. Un costruttore inizializzerà questi dati a 0, mentre un altro li inizializzerà a valori predefiniti. Una funzione membro dovrà visualizzare l'ora in formato 11:59:59. Un'altra funzione membro sommerà due oggetti di tipo ora passati come argomenti. Una funzione principale main() deve creare due oggetti inizializzati e un terzo che non lo sia. Sommare i due valori inizializzati e lasciare il risultato nell'oggetto non inizializzato. Da ultimo visualizzare il valore risultante.

**Esercizio 9** Creare una classe chiamata impiegato che contenga come membri dato il nome e il numero di impiegato, e come funzioni membro Leggeredati() e vederedati() che legga i dati dalla tastiera e li visualizzi su schermo, rispettivamente.

Scrivere un programma che utilizzi la classe, creando un array di tipo impiegato e poi riempiendolo con dati corrispondenti a 50 impiegati. Una volta riempito l'array, visualizzare i dati di tutti gli impiegati.

**Esercizio 10** Si vuole realizzare una classe vettore3d che permetta di manipolare vettori di tre coordinate x, y, z secondo le seguenti regole:

- vi sarà solo una funzione costruttore e sarà in linea;
- vi sarà una funzione membro uguale che permette di sapere se due vettori hanno le loro componenti o coordinate uguali (la dichiarazione di uguale si realizzerà utilizzando: a. trasmissione per valore; b. tra-

smissione per indirizzo; c. trasmissione per riferimento).

**Esercizio 1** Includere nella classe vettore3d dell'esercizio precedente una funzione membro denominata normamax che permetta di ottenere la norma maggiore dei due vettori. (Nota: La norma di un vettore  $v = x, y, z$  è  $x^2+y^2+z^2$  oppure  $x*x+y*y+z*z$ ).

**Esercizio 2** Progettare una classe vettore3d che permetta di manipolare vettori 3D (di tipo float) e che contenga una funzione costruttore con valori per default (0) e le funzioni membri somma (somma di due vettori), prodottoscalare (prodotto scalare di due vettori:  $v_1 = x_1, y_1, z_1; v_2 = x_2, y_2, z_2; v_1 \cdot v_2 = x_1 * x_2 + y_1 * y_2 + z_1 * z_2$ ).

**Esercizio 3** Indicare qual è la differenza di significato tra la struttura:

```
struct a {
    int i, j, k;
};
```

e la classe:

```
class a {
    int i, j, k;
};
```

Spiegare la ragione per cui la dichiarazione della classe è inutile. Come si può utilizzare la parola public per convertire la dichiarazione della classe in una dichiarazione equivalente a struct a?

**Esercizio 4** Creare una classe Complesso per la gestione di numeri complessi (un numero complesso è costituito di due numeri reali double: parte reale e parte immaginaria). Le operazioni da implementare sono le seguenti:

- una funzione assegnare() che assegna valori al numero Complesso;
- una funzione stampare() che visualizzi un oggetto di tipo Complesso;
- due funzioni aggregare() (sovrafficate) per, rispettivamente, aggiungere un Complesso a un altro e aggiungere due componenti double a un Complesso.

**Esercizio 5** Creare una classe vettore in grado di:

- creare un vettore semplice che contenga zero o più elementi di un certo tipo;
- creare un vettore vuoto;
- aggiungere elementi al vettore;
- determinare se il vettore è vuoto;

- determinare se il vettore è pieno;
- accedere a ogni elemento del vettore e realizzare qualche azione su di essi.

**Esercizio 6** Implementare una classe Random per generare numeri pseudoaleatori.

**Esercizio 7** Implementare una classe Data con membri dato per il mese, il giorno e l'anno. Ogni oggetto di questa classe rappresenta una data, che rappresenta giorno, mese e anno come interi (giorno, g, mese, m, e anno, a). Si deve includere un costruttore di default, un costruttore di copia, funzioni di accesso, una funzione reset (int g, int m, int a) per reimpostare la data di un oggetto, una funzione avanzare(int g, int m, int a) per far avanzare la data e una funzione stampare(). Utilizzare una funzione di utilità normalizzare() che assicuri che i membri dato stiano nell'intervallo corretto  $1 \leq a, 1 \leq m \leq 12, g \leq \text{giorni}(\text{Mese})$ , dove giorni(Mese) è un'altra funzione che restituisce il numero di giorni di ogni mese.

**Esercizio 8** Ampliare il programma precedente in modo che possa accettare anni bisestili. (Nota: un anno è bisestile se è divisibile per 400, o se è divisibile per 4 ma non per 100. Per esempio, gli anni 1992 e 2000 sono bisestili e 1997 e 1900 non lo sono).

**Esercizio 9** Definire una classe Razionale che rappresenti numeri razionali (frazionari). I membri privati saranno il numeratore e il denominatore della frazione, e nella parte pubblica si deve disporre almeno delle seguenti funzioni membri: assegnare, convertire, invertire, stampare, che realizzeranno le funzioni di: assegnare i valori dei parametri a numeratore e denominatore rispettivamente (per esempio, 22/7); convertire a decimale il numero razionale (per esempio, 3.14286); calcolare l'inverso della frazione (per esempio, 7/22) e da ultimo, visualizzare la frazione (per esempio, si deve vedere 22/7, 6/15 ecc.).

**Esercizio 10** Implementare una classe Punto che rappresenti punti in tre dimensioni (x, y, z). Includere un costruttore di default, un costruttore di copia, una funzione negare() che trasformi il punto nel suo contrario (negativo), una funzione norma() che ritorni la distanza dal punto all'origine (0,0,0) e una funzione visualizzare().

**Esercizio 1** Implementare una classe `Ora`. Ogni oggetto di questa classe rappresenta un'ora specifica di un giorno, immagazzinando le ore, minuti e secondi come interi. Includere un costruttore, funzioni di accesso, una funzione `avanzare()` per avanzare (anticipare) l'ora attuale di un oggetto esistente, una funzione `reset` per mettere a zero l'ora attuale di un oggetto esistente e una funzione `visualizzare()`.

**Esercizio 2** Implementare una classe `Persona` che rappresenti dati personali di una persona. I membri dati debbono includere il nome della persona, la data di nascita e l'anno di laurea all'università. Si deve includere un costruttore di default, un distruttore, funzioni di accesso e una funzione di visualizzazione.

**Esercizio 3** Implementare una classe `Stringa`. Ogni oggetto della classe rappresenta una stringa di caratteri. I membri dato sono la lunghezza della stringa e la stringa di caratteri attuale. Inoltre, si debbono aggiungere costruttori, distruttori, funzioni di accesso e funzioni per visualizzare, e anche includere una funzione `carattere(int i)` che restituisca il carattere situato nell'indice `i` della stringa.

**Esercizio 4** Implementare una classe `Matrice 2 × 2` che includa un costruttore di default, un costruttore di copia, una funzione `inversa()` che restituisca l'inverso della matrice, una funzione `det()` che ritorna il determinante della matrice, una funzione logica (bool-

ean) `DetZero` che ritorna true o false (a seconda di se il determinante sia zero o meno) e una funzione `visualizzare()`.

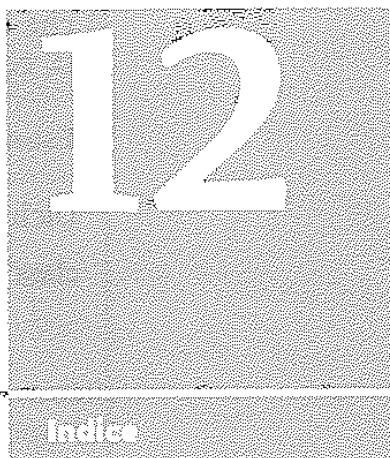
**Esercizio 5** Si supponga che  $a = (A, Bi)$  e  $c = (C, Di)$ . Si vuole aggiungere alla classe `Complesso` le operazioni:

- somma:  $a + c = (A + C, (B + D)i)$ ;
- sottrazione:  $a - c = (A - C, (B - D)i)$ ;
- moltiplicazione:  $a * c = (A * C - B * D, (A * D + B * C)i)$ ;
- moltiplicazione:  $x * c = (x * C, x * Di)$ , dove  $x$  è un numero reale;
- coniugato:  $\bar{a} = (A, -Bi)$ .

**Esercizio 6** Scrivere una classe `Insieme` che gestisca un insieme di interi (`int`) con l'aiuto di una tabella di dimensione fissa (un insieme contiene un vettore ordinato di elementi e si caratterizza per il fatto che ogni elemento è unico: non si deve trovare due volte lo stesso valore nella tabella). Le operazioni da implementare sono le seguenti:

- `svuotare()` svuota l'insieme;
- `aggiungere()` aggiunge un intero all'insieme;
- `eliminare()` rimuove un intero dall'insieme;
- `copiare()` riepiloga un insieme in un altro;
- `e_membro()` restituisce un valore booleano (valore logico che indica se l'insieme contiene un certo intero);
- `e_uguale()` restituisce un valore booleano che indica se un insieme è uguale a un altro;
- `stampare()` stampa a schermo l'insieme in maniera leggibile.

# Classi derivate: ereditarietà e polimorfismo



**12.1** Classi derivate  
**12.2** Tipi di ereditarietà  
**12.3** Distruttori

**12.4** Ereditarietà multipla  
**12.5** Polimorfismo  
**12.6** Vantaggi del polimorfismo

Indice

## Introduzione

Questo capitolo mostra come creare *classi derivate* che ereditino attributi e comportamenti da *classi base* già esistenti, generando così "gerarchie di classi" che riducono la ridon-

danza del codice. L'ereditarietà è importante per produrre un software affidabile, comprensibile, economico, adattabile e riutilizzabile.

### 12.1 Classi derivate

Il concetto di *ereditarietà* nasce dal bisogno di costruire una nuova classe più specifica a partire da una più generale; la nuova *classe derivata* eredita attributi e metodi dalla *classe base* già esistente. Così, per esempio, se esiste una classe FiguraGeometrica e si vuole creare una classe Triangolo, quest'ultima può derivare dalla prima poiché avranno in comune uno stato e un comportamento, anche se poi avrà le sue proprie caratteristiche. Triangolo è *un tipo* di Figura.

Evidentemente, la classe base e la classe derivata avranno metodi e attributi comuni, quindi se si creasse la derivata indipendentemente dalla base si duplicherebbe parecchio codice.

La dichiarazione di una classe derivata deve includere il nome della classe base da cui deriva e, eventualmente, uno *specificatore d'accesso* indicante il tipo di ereditarietà, public, private o protected, secondo la seguente sintassi:

```
class ClasseDerivata : specifikatoreAccesso ClasseBase {  
    membri;  
};
```

- *public ereditarietà pubblica*
- *private ereditarietà privata (default)*
- *protected ereditarietà protetta*.

Se si omette lo specificatore di accesso opzionale l'ereditarietà sarà per default privata, quindi si dovrebbe includere la parola riservata public per far

sì che i membri pubblici nella classe base rimangano tali nella classe derivata, come cerchiamo di spiegare nel seguito.

#### Esempio 12.1

Dichiarazione della classe Triangolo.

```
class Triangolo : public FiguraGeometrica
{
    public:
        nuove funzioni membro
    private:
        nuovi membri dato
};
```

#### Esempio 12.2

Rappresentare la gerarchia di classi di un software per gestire archivi bibliotecari.

*Tutte le pubblicazioni hanno in comune un editore e una data di pubblicazione. Le riviste hanno una determinata periodicità, una tiratura annua, e, per esempio, il numero di copie vendute. I libri invece hanno un codice ISBN e il nome dell'autore.*



```
class Pubblicazione {
    public:
        void InserireEditore(const char *s);
        void InserireData(unsigned long dat);
    private:
        string editore;
        unsigned long data;
};

class Rivista : public Pubblicazione {
    public:
        void InserireTiratura(unsigned n);
        void InserireVenduto(unsigned long n);
    private:
        unsigned tiratura;
        unsigned long venduto;
};
```

```

class Libro : public Pubblicazione {
public:
    void InserireISBN(const char *s);
    void InserireAutore (const char *s);

private:
    string ISBN;
    string autore;
};

```

Un oggetto della classe Libro contiene membri dato e funzioni ereditate dalla classe Pubblicazione, e in più ISBN e nome dell'autore. Saranno dunque possibili le seguenti operazioni:

```

Libro L;
L.DareNomeEditore("McGraw-Hill");
L.StabilireData(990606);
L.DareISBN("84-481-2015-9");
L.DareAutore("Mackoy, Jose Luis");

```

Invece, le seguenti operazioni si possono eseguire solo su oggetti Rivista:

```

Rivista R;
R.NumeriAllAnno(12);
R.FissareCircolazione(300000L);

```

Se non vi fosse l'ereditarietà sarebbe necessario fare una copia del codice sorgente di una classe, dargli un nuovo nome e aggiungergli nuove operazioni e/o membri dato. Ciò renderebbe difficile il mantenimento delle classi, poiché eventuali cambi nella base dovrebbero essere effettuati a mano in tutte le classi derivate.

A volte è difficile decidere qual'è la miglior relazione d'ereditarietà. Consideriamo, per esempio, il caso di lavoratori di un'impresa. Vi sono differenti classificazioni in base a vari *discriminatori*: tipo di pagamento (stipendio fisso, a ore, a cottimo), relazione lavorativa con l'impresa (tempo indeterminato o determinato), status (tempo pieno o parziale) ecc., e queste classificazioni possono avere ovviamente elementi in comune, cioè uno stesso impiegato può appartenere a differenti classi di lavoratori. Un impiegato a tempo pieno può essere remunerato con un salario mensile, un operaio a tempo parziale può essere remunerato a cottimo e un dipendente a tempo determinato può essere remunerato a ore. Allora la domanda è: qual è la relazione di ereditarietà che meglio descrive le trasmissioni di attributi e operazioni alle varie classi derivate? Evidentemente, la risposta dipenderà dall'applicazione reale da sviluppare.

## 12.2 Tipi di ereditarietà

Come già sappiamo, in una classe vi possono essere tre sezioni: pubblica, privata e protetta. Gli elementi pubblici sono accessibili a tutte le funzioni; gli elementi privati sono accessibili soltanto ai membri della stessa classe e

(questo non lo potevamo spiegare nel precedente capitolo) gli elementi protetti possono essere acceduti *anche da classi derivate*. Analogamente vi sono tre tipi di ereditarietà: *pubblica*, *privata* e *protetta*, agli effetti delle conseguenze sull'accessibilità dei membri della classe base dai metodi della classe derivata come sarà riassunto dalla Tabella 12.1.

### 12.2.1 Ereditarietà pubblica

Ereditarietà *pubblica* significa che la classe derivata ha accesso agli elementi pubblici e protetti della sua classe base. Gli elementi pubblici si ereditano come pubblici e gli elementi protetti come protetti, in altre parole i membri ereditati hanno la stessa protezione che nella classe base.

#### Esempio 12.3

Si definisca una gerarchia di classi `ogg_geometrico`, `quadrato` e `cerchio` e un programma che la utilizzi.



```

class ogg_geom {
public:
    ogg_geom(float x=0, float y=0) : xc(x), yc(y) {}
    void stampacentro() const {
        cout << xc << " " << yc << endl;
    }
protected:
    float xc, yc;
};

const float PI = 3.14159265;

class cerchio : public ogg_geom {
public:
    cerchio(float x_C, float y_C, float r) : ogg_geom (x_C, y_C) {
        raggio = r;
    }
    float area() const {return PI * raggio * raggio; }
private:
    float raggio;
};

class quadrato : public ogg_geom {
public :
    quadrato(float x_C, float y_C, float x, float y) : ogg_geom(x_C,y_C) {
        x1 = x;
    }
};
  
```

```

        y1 = y;
    }
float area() const {
    float a, b;
    a = x1 - xC;
    b = y1 - yC;
    return 2 * (a * a + b * b);
}
private:
    float x1, y1;
};

int main()
{
    cerchio C(2, 2.5, 2);
    quadrato Q(3, 3.5, 4.37, 3.85);
    cout << "Centro del cerchio : " ;
    C.stampacentro();
    cout << "Centro del quadrato : " ;
    Q.stampacentro();
    cout << "Area del cerchio : " << C.area() << endl;
    cout << "Area del quadrato : " << Q.area() << endl;
    return 0;
}

```

### Esecuzione

```

Centro del cerchio : 2 2.5
Centro del quadrato : 3 3.5
Area del cerchio : 12.5664
Area del quadrato : 3.9988

```

Un cerchio è caratterizzato dal suo centro e dal suo raggio, mentre un quadrato si può determinare dal suo centro e da uno dei suoi quattro vertici. Dichiariamo le due figure geometriche come classi derivate (Figura 12.1).

Tutti i membri pubblici della classe base `ogg_geom` sono anche membri pubblici della classe derivata `quadrato`. La classe `quadrato` deriva pubblicamente da `ogg_geom`.

Nel `main()` si può definire:

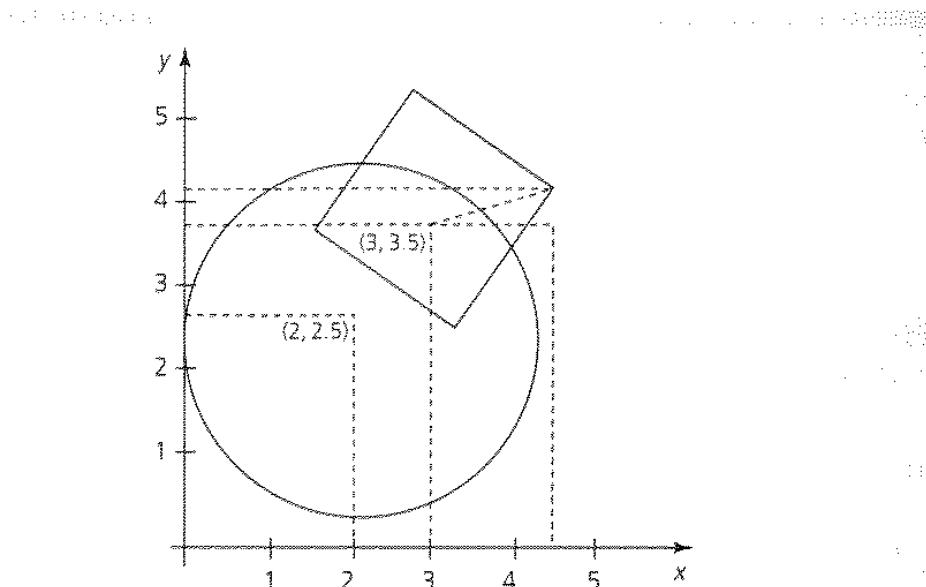
```

quadrato Q(3, 3.5, 4.37, 3.85);
Q.stampacentro();

```

Anche se `stampacentro` non è un metodo della classe `quadrato`, è però ereditato dalla classe `ogg_geom`, dalla quale `quadrato` deriva in maniera `public` e nella quale esso è definito come metodo `public`.

Un altro punto da osservare è l'uso dei membri protetti `xC` e `yC` nel metodo `area` della classe `quadrato`; essi sono membri `protected` della classe base `ogg_geom`, dunque si può accedere a essi dalla classe derivata in maniera `public`.



**Figura 12.1**  
Cerchio (centro: 2, 2.5), quadrato (centro: 3, 3.5).

### 12.2.2 Ereditarietà privata

Con l'ereditarietà privata, un utente della classe derivata non ha accesso ad alcun elemento della classe base. I membri pubblici e protetti della classe base, ovvero gli unici ereditabili, diventano membri privati della classe derivata, quindi gli utenti della classe derivata non hanno accesso alle facilità fornite dalla classe base.

Sebbene non sia molto utilizzata, l'ereditarietà privata è *quella di default*. Essa occulta gli elementi della classe base all'utente della classe derivata perché sia possibile cambiare l'implementazione della classe base senza che le applicazioni che utilizzano la classe derivata debbano modificare una singola riga di codice.

### 12.2.3 Ereditarietà protetta

Con l'ereditarietà protetta, gli elementi ereditabili della classe base diventano membri protetti della classe derivata. Questo significa che non saranno utilizzabili dalle applicazioni (come d'altronde non lo erano neanche se erano *protected* già nella classe base) ma saranno utilizzabili dalla classe derivata e, a loro volta, ulteriormente ereditabili da parte di altre classi derivate da questa. L'ereditarietà *protected* è quindi appropriata quando gli elementi della classe base sono utili nell'implementazione della classe derivata, ma non sono parte dell'interfaccia che quella classe derivata espone a favore delle applicazioni che l'utilizzano. L'ereditarietà protetta è ancora meno frequente dell'ereditarietà privata.

La Tabella 12.1 riassume gli effetti dei tre tipi di ereditarietà sull'accessibilità dei membri della classe derivata.

Un'ultima annotazione: una classe derivata non può accedere agli elementi private della sua classe base. Per occultare elementi agli utenti di una

Tabella 12.1 Tipi di ereditarietà e accessi che permettono

Tipo di ereditarietà	Accesso a membro classe base	Accesso a membro classe derivata
public	public	public
	protected	protected
	private	<i>inaccessibile</i>
protected	public	protected
	protected	protected
	private	<i>inaccessibile</i>
private	public	private
	protected	private
	private	<i>inaccessibile</i>

classe base il programmatore dovrebbe valutare l'opportunità di dichiararli protected invece che private per renderli eventualmente disponibili a classi derivate con ereditarietà pubblica o protetta.



Per default, l'ereditarietà è privata. Se si dimentica la parola riservata public, gli elementi della classe base saranno inaccessibili. Il tipo di ereditarietà è, pertanto, una delle prime cose da verificare quando il compilatore restituisce un messaggio di errore indicante che variabili o funzioni sono inaccessibili.



#### Esempio 12.4

```
class Base {
public:
    int i1;
protected:
    int i2;
private:
    int i3;
};
class D1 : private Base {
    void f();
};
class D2 : protected Base {
    void g();
};
class D3 : public Base {
    void h();
};
```

Nessuna delle sottoclassi ha accesso al membro i3 della classe Base. Le tre classi possono accedere invece ai membri i1 e i2. Nella definizione della funzione membro f() si ha:

```
void D1::f() {
    i1 = 0;          // Corretto
    i2 = 0;          // Corretto
    i3 = 0;          // Errore
};
```

I livelli di accessibilità a i1, i2 e i3 da un'applicazione (int main()) che utilizza le tre classi sono:

```
int main()
{
    Base b;
    b.i1 = 0;      // Corretto
    b.i2 = 0;      // ERRORE
    b.i3 = 0;      // ERRORE
    D1 d1;
    d1.i1 = 0;    // ERRORE
    d1.i2 = 0;    // ERRORE
    d1.i3 = 0;    // ERRORE
    D2 d2;
    d2.i1 = 0;    // ERRORE
    d2.i2 = 0;    // ERRORE
    d2.i3 = 0;    // ERRORE
    D3 d3;
    d3.i1 = 0;    // Corretto
    d3.i2 = 0;    // ERRORE
    d3.i3 = 0;    // ERRORE
};
```

#### 12.2.4 Operatore di risoluzione di visibilità

C'è però un metodo per rendere accessibili alla classe derivata i membri della classe base anche con ereditarietà privata o protetta: la *dichiarazione d'accesso*. Si tratta semplicemente di nominare in maniera appropriata uno dei membri della classe base dentro la derivata.

```
class D4 : protected Base {
public:
    Base::i1;
```

Dichiarare Base::i1 nella sezione pubblica di D4 rende i1 pubblico in D4 anche sotto ereditarietà protetta. Si può, allora, scrivere nel main()

```
D4 d4;
d4.i1 = 0;          // CORRETTO
```

In modo simile si può rendere i2 protetto in D5, anche sotto ereditarietà privata, nominando Base::i2 nella sezione protetta di D5.

```
class D5 : private Base {
protected:
    Base::i2;
};
```

### 12.2.5 Costruttori in ereditarietà

Una classe derivata è una specializzazione della corrispondente classe base. Di conseguenza, bisogna prima che parta il costruttore della classe base (per creare un oggetto della classe base) perché si attiverà poi il costruttore della classe derivata. L'oggetto della classe base deve *esistere* prima di specializzarsi come oggetto della classe derivata.



1. Quando si definisce un oggetto di una classe discendente, la sequenza d'esecuzione dei costruttori segue quella genealogica, dalla classe ancestrale a quella dell'oggetto.
2. I costruttori non si ereditano.

### Esempio 12.5

```
class A {
public:
    A() { cout << "Costruttore A" << endl; }
};

class B : public A {
public:
    B() { cout << "Costruttore B" << endl; }
};

class C : public B {
public:
    C() { cout << "Costruttore C" << endl; }
};

int main() {
    C c1;
    return 0;
}
```

### Esecuzione

Costruttore A  
Costruttore B  
Costruttore C

Se lo si dichiara fuori dalla classe, la sintassi del costruttore di una classe derivata è:

```
ClasseDer:: ClasseDer (paramD): ClasseBase (paramB), Inizial {
    //corpo del costruttore della classe derivata
};
```

l'intestazione include la chiamata al costruttore della classe base. Questo viene eseguito prima del costruttore della classe derivata perché l'oggetto derivato ha bisogno dell'oggetto base per diventare oggetto derivato. Il costruttore di una classe derivata dovrà dunque:

- inizializzare l'oggetto base;
- inizializzare tutti i membri dati.

La classe derivata ha un *costruttore-inizializzatore*, che invoca uno o più costruttori della classe base. L'inizializzatore appare immediatamente dopo i parametri del costruttore della classe derivata ed è preceduto dall'operatore due punti (:).

La sintassi per la definizione dei metodi di una classe derivata è identica a quella di una classe base.

```
Tipo ClasseDer::NomeFunzione(parametri) {
    // corpo della funzione membro della classe derivata
};
```

#### Esempio 12.6

```
class Punto2D {
public:
    Punto2D(int x, int y) {ascissa=x; ordinata=y;};
    int ascissa;
    int ordinata;
};

class Punto3D: public Punto2D {
public:
    Punto3D(int, int, int); // prototipo
    float modulo();
private:
    int quota;
};
// costruttore classe derivata
Punto3D::Punto3D(int x, int y, int z): Punto2D(x, y){
    quota = z;
}
// metodo classe derivata
float Punto3D::modulo() {
    int r = ascissa*ascissa + ordinata*ordinata + quota*quota;
    return sqrt(r);
}

int main()
```

```

Punto3D p(10,20,30);
cout << p.modulo();
return 0;
}

```

**Esecuzione**

37.4166

**12.3 Distruttori**

I distruttori sono molto più facili da trattare dei costruttori perché non si ereditano. Se il programmatore non lo specifica il compilatore ne costruisce uno di default con l'unico compito di liberare la memoria assegnata dal costruttore. Per oggetti di classi discendenti da altre classi l'ordine di esecuzione è inverso a quello dei costruttori, vale a dire che si eseguono i distruttori a partire dalla classe dell'oggetto fino alla prima classe genitrice.

**Esempio 12.7**

```

class C1 { // classe base
public:
    C1(int n); // prototipo costruttore
    ~C1(); // prototipo distruttore
private:
    int *pi, l;
};

class C2 : public C1 { // classe derivata
public:
    C2(int n); // prototipo costruttore
    ~C2(); // prototipo distruttore
private:
    char *pc;
    int l;
};

C1::C1(int n) :l(n) { // costruttore classe base
    cout << "Alloco vettore di " << l << " interi\n";
    pi = new int[l];
}

C2 ::C2(int n) :C1(n), l(n) { // costruttore classe derivata
    cout << "Alloco vettore di " << l << " caratteri\n";
    pc = new char[l];
}

C1::~C1() { // distruttore classe base
    cout << "Dealloco vettore di " << l << " interi\n";
}

```

```

    delete[] pi;
}

C2::~C2() {           // distruttore classe derivata
    cout << "Dealloco vettore di " << l << " caratteri\n";
    delete[] pc;
}

int main()
{
    C2 a(50), b(100);
}

```

### Esecuzione

Allocò vettore di 50 interi  
 Allocò vettore di 50 caratteri  
 Allocò vettore di 100 interi  
 Allocò vettore di 100 caratteri  
 Deallocò vettore di 100 caratteri  
 Deallocò vettore di 100 interi  
 Deallocò vettore di 50 caratteri  
 Deallocò vettore di 50 interi

## 12.4 Ereditarietà multipla

Con l'**ereditarietà multipla** una classe eredita attributi e comportamento di più di una classe base. In altre parole, vi è ereditarietà multipla quando vi sono molteplici classi base per una stessa classe derivata (Figura 12.2).

L'ereditarietà multipla comporta complicazioni nell'implementazione del compilatore. Essa aumenta le operazioni ausiliarie e complementari e

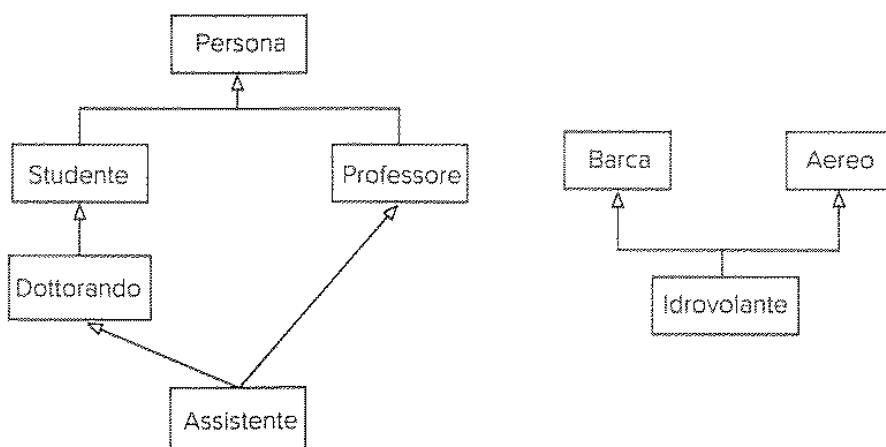


Figura 12.2  
Esempi di ereditarietà multipla.

introduce potenziali ambiguità. Inoltre, la programmazione di classi derivate per derivazione multipla tende a produrre più classi di quante necessarie con ereditarietà semplice. Nonostante questi inconvenienti l'ereditarietà multipla può semplificare i programmi e fornire soluzioni per risolvere problemi difficili.

Con l'ereditarietà semplice lo scenario è abbastanza comprensibile, sia in termini concettuali sia implementativi. Però l'ereditarietà multipla aggiunge forza ai programmi e, utilizzata con precauzione sia in fase di analisi del problema sia di progettazione del codice, facilita la soluzione di parecchi problemi che mostrano un'intrinseca natura di ereditarietà multipla.

La sintassi è:

```
class Derivata : [virtual][tipo_accesso] Base1,  
[virtual][tipo_accesso] Base2,  
[virtual] [tipo_accesso] Basen {  
...  
};
```

#### *Derivata*

Nome della classe derivata.

#### *tipo\_accesso*

public, private o protected, con le stesse regole dell'ereditarietà semplice

Base1, Base2,...

Classi base.

#### *virtual*

Opcionale (vedremo poi).

Funzioni o dati membro che abbiano lo stesso nome in Base1, Base2, Basen, costituiranno motivo di ambiguità, ed è questo il motivo per cui alcuni linguaggi di programmazione a oggetti non ammettono l'ereditarietà multipla.

#### Esempio 12.8

```
class A : public B, public C {  
... // A eredita pubblicamente sia da B sia da C  
};  
class D : public E, private F, public G {  
... // D eredita pubblicamente da E e G, privatamente da F  
};  
class X : public E, Z [  
... // X eredita pubblicamente da E e privatamente da Z  
};  
class M : virtual public N, virtual public P {  
... // N e P sono classi base virtuali di M  
};
```

**CONSIGLI**

È meglio specificare il tipo di accesso in tutte le classi base ed evitare l'accesso privato per default; si utilizzi esplicitamente private al bisogno per migliorare la leggibilità.

```
class Derivata : public Base1, private Base2 { ... }
```

**Esempio 12.9**

```
class A {
public:
    A() { cout << "Costruttore A" << endl; }
};

class B {
public:
    B() { cout << "Costruttore B" << endl; }
};

class C : public A {
public:
    C() { cout << "Costruttore C" << endl; }
};

class D : public C, public B {
public:
    D() { cout << "Costruttore D" << endl; }
};

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

**Esecuzione**

```
Costruttore A
Costruttore C
Costruttore B
Costruttore D
```

**12.4.1 Caratteristiche dell'ereditarietà multipla**

L'ereditarietà multipla comporta diversi problemi come l'*ambiguità* per l'uso di nomi identici in differenti classi base, e la *precedenza* di funzioni o dati.

**Ambiguità**

L'ereditarietà multipla ha il potenziale problema delle ambiguità.

```
class Finestra {
private:
    ...
```

```

public :
void aprire(); // apre la finestra
...
};

class Rubinetto {
private:
...
public:
void aprire(); // apre il rubinetto
...
};

```

Sia la classe Finestra sia la classe Rubinetto posseggono un metodo che si chiama aprire(), ma le due funzioni membro potrebbero essere implementate in maniera diversa. Se si crea una classe FinRub con ereditarietà multipla, il riferimento al metodo aprire() sarà ambiguo:

```

class FinRub : public Finestra, public Rubinetto {
...
};

FinRub f;
f.aprire(); // si produce un errore, quale?

```

La chiamata a aprire è ambigua perché il compilatore non saprà quale delle due funzioni deve chiamare. L'ambiguità si risolve però facilmente con l'operatore di risoluzione di visibilità.

```

f.Rubinetto::aprire(); // chiamata ad aprire di Rubinetto
f.Finestra::aprire(); // chiamata ad aprire di Finestra

```

**ATTENZIONE**  
Non è un errore definire un oggetto derivato con ambiguità potenziali perché producono errori in fase di compilazione solo se sono invocate in maniera ambigua. È però meglio risolvere le ambiguità nelle definizioni stesse dei metodi. Per esempio nel caso precedente sarebbe meglio specificare così:

```

class FinRub : public Finestra, public Rubinetto {
...
void f_aprire() { Finestra::aprire(); }
void r_aprire() { Rubinetto::aprire(); }
};

```

L'ambiguità vale anche per gli attributi, come mostrato nell'esempio seguente.

#### Esempio 12.10

```

class lavoratore {

```

```

public:
    const int num_ss;
    const char* nome;
    ...
};

class studente {
public:
    const char* nome;
    ...
};

class stu_lavoratore : public studente, public lavoratore {
public:
    void stampare() {
        cout << "numero ss " << num_ss << endl;
        cout << nome;      // ERRORE di AMBIGUITÀ
        ...
    }
    ...
};

```

Per evitare errori nell'invocazione di nome si dovrebbe fare uso dell'operatore di risoluzione di visibilità.

```

studente::nome
lavoratore::nome

```

L'ambiguità vale anche se i prototipi delle funzioni sono diversi:

```

class A {
public:
    ...
    void f(int);
};

class B {
public:
    ...
    void f(const String &);

};

class C : public A, public B {
public:
    ...
};

int main() {
C c;
c.f(15); // ERRORE di AMBIGUITÀ
}

```

### 12.4.2 Precedenza

Un problema dell'ereditarietà è la *precedenza* che entra in gioco quando vi siano funzioni in classi derivate con lo stesso *nome* di altre funzioni delle classi base. Consideriamo dapprima un caso in ereditarietà semplice:

```
class Base {
public:
    void f(int);
    void f(char);
    void f(const String &);

};

class Derivata : public Base {
public:
    void f(const String &); // occulta tutte le Base::f()
};

int main() {
Derivata d;
d.f(5); // errore, Base::f(int) è occultata
d.f('x'); // errore, Base::f(char) è occultata
d.f("abc"); // corretto, Derivata::f(const String &);

}
```

In ereditarietà semplice, le funzioni delle classi derivate hanno *precedenza* e quindi rimpiazzano tutte le eventuali omonime della classe base. Il problema nasce con l'ereditarietà multipla quando la precedenza si combina con l'ambiguità, cioè se una classe derivata rimpiazza funzioni omonime ereditate da classi base differenti. La funzione della derivata dovrebbe comunque rimpiazzarle tutte, ma la cosa genera imbarazzo al compilatore che potrebbe segnalare l'ambiguità come errore.

#### RISPOSTA

Quando la precedenza si associa all'ambiguità nelle classi base, è bene compiere chiamate dirette nella classe Derivata alle rispettive funzioni della classe Base.

```
class C : public A, public B {
public:
    void f(int i) {
        A::f();
    }

    void f(const String &s) {
        B::f(s);
    }
};
```

L'ereditarietà multipla potrebbe confondere il programmatore anche nel trattare i costruttori e i distruttori delle classi base. Un esempio chiarirà questo problema:

```

class A { // classe base
public:
    A(int); // costruttore con argomenti
    ~A(); // distruttore
};

class B { // classe base
public:
    B(); // costruttore senza argomenti
    ~B(); // distruttore
};

class C { // classe base
public:
    C(double); // costruttore con argomenti
    ~C(); // distruttore
};

class D : public A, public B, public C { // ereditarietà multipla
public:
    D(int i, double m) :A(i), C(m) {} // costruisce A,B,C e D
    ~D(); // distrugge D,C,B,A
};

```

La classe D deriva pubblicamente dalle classi base A, B e C. I costruttori di A e C hanno ciascuno un argomento, mentre B ha un costruttore void. Per costruire oggetti D, il costruttore D passa il suo argomento int al costruttore di A e il suo argomento double al costruttore di C. Il vettore di inizializzazione dei membri non ha bisogno di passare argomenti al costruttore di B. È la *definizione della classe* che determina l'ordine delle chiamate ai costruttori e ai distruttori, non il vettore di inizializzazione dei membri. Quindi i vettori di inizializzazione di membri possono inizializzare le classi base in *qualsiasi* ordine. In altre parole, il seguente costruttore modificato di D chiama i costruttori base nello stesso ordine di prima.

```
D(int i, double m) : C(m), A(i) {} // costruisce A,B,C,D
```

L'ordine nella definizione della classe, influenza invece l'ordine delle chiamate dei costruttori.

```

class D : public C, public B, public A {
public:
    D(int i, double m) : A(i), C(m) {} // costruisce C,B,A,D
    ~D(); // distrugge D,A,B,C
};

```

### ATTENZIONE

Se l'ordine delle chiamate ai costruttori base è importante, in una dichiarazione di ereditarietà multipla è bene assicurarsi che l'elenco delle classi base sia quello corretto.

## 12.5 Polimorfismo

Nell'OOP il *polimorfismo* è la proprietà in base alla quale oggetti di classi diverse possono rispondere in maniera diversa a una stessa invocazione. Il polimorfismo è particolarmente importante se utilizzato insieme all'ereditarietà e ai puntatori, ma nasconde dei trabocchetti.

### Esempio 12.11

```
class A {
public:
    int x = 22;
    void Dinamica() {cout << "DINAMICA della classe A\n";}
    void Statica() {cout << "STATICA della classe A\n";}
};

class B : public A {
public:
    int x = 44;
    void Dinamica() { cout << "DINAMICA della classe B\n"; }
    void Statica() { cout << "STATICA della classe B\n"; }
};

int main() {
    A a;
    B b;
    cout << "Funzioni dell'oggetto della classe A\n";
    cout << a.x << endl;
    a.Dinamica();
    a.Statica();
    cout << endl;
    cout << "Funzioni dell'oggetto della classe B\n";
    cout << b.x << endl;
    b.Dinamica();
    b.Statica();
    cout << endl;
    a = b; // assegna all'oggetto a i valori dell'oggetto b
    cout << "Funzioni dell'oggetto della classe A dopo a=b\n";
    cout << a.x << endl;
    a.Dinamica();
    a.Statica();
}
```

### Esecuzione

```
22
DINAMICA della classe A
STATICA della classe A
Funzioni dell'oggetto della classe B
```

44

DINAMICA della classe B

STATICA della classe B

Funzioni dell'oggetto della classe A dopo a=b

22

DINAMICA della classe A

STATICA della classe A

Gli elementi della classe derivata B hanno gli stessi identificatori della classe A da cui deriva, quindi li occultano alla vista dell'applicazione `main()`; ma quando si assegna all'oggetto a l'oggetto b, e quindi si dovrebbe operare una riscrittura bit a bit del contenuto di b in a, si osserva che quest'ultimo mantiene intatti i suoi elementi, sia l'attributo sia i metodi. Questo accade perché b ha ereditato i tre elementi di a ma gliene ha aggiunti altri tre, i quali appunto *occultano* quelli ereditati, perché ne condividono il nome, ma non li *cancellano*; quindi quando si assegnano ad a gli elementi di b, in realtà non si assegna proprio nulla poiché i tre elementi di a ereditati da b erano rimasti inalterati.

Nell'esempio che segue modifichiamo l'applicazione.

#### Esempio 12.12

```
int main() {
    A a;
    B b;
    cout << "Funzioni dell'oggetto della classe A\n";
    cout << a.x << endl;
    a.Dinamica();
    a.Statica();
    cout << endl;
    cout << "Funzioni dell'oggetto della classe B\n";
    cout << b.A::x << endl;
    b.A::Dinamica();
    b.A::Statica();
    cout << endl;
    a = b;
    cout << "Funzioni dell'oggetto della classe A dopo a=b\n";
    cout << a.x << endl;
    a.Dinamica();
    a.Statica();
}
```

#### Esecuzione

22

DINAMICA della classe A

STATICA della classe A

Funzioni dell'oggetto della classe B

22

DINAMICA della classe A

STATICA della classe A

Funzioni dell'oggetto della classe A dopo a=b

22

DINAMICA della classe A

STATICA della classe A

Utilizzando opportunamente l'operatore di risoluzione di visibilità nella chiamata agli elementi di b ci riferiamo agli elementi che b ha ereditato da a occultandoli per sovraccaricamento. Purtroppo però questa opportunità non dà la flessibilità che si vorrebbe perché non la si potrebbe gestire all'interno di un ciclo su un vettore di oggetti di classi diverse. Se infatti volessimo spaziare da un oggetto di una classe a uno di un'altra mediante un puntatore credendo che l'invocazione di metodo per dereferenziamento faccia partire un metodo oppure un altro a seconda dell'oggetto puntato rimarremmo delusi.

**Esempio 12.13**

```
int main() {
    A a; A* pa = &a;
    B b; B* pb = &b;
    cout << "Funzioni dell'oggetto della classe A\n";
    pa -> Dinamica();
    pa -> Statica();
    cout << endl;
    cout << "Funzioni dell'oggetto della classe B\n";
    pb -> Dinamica();
    pb -> Statica();
    cout << endl;
    pa = pb; // manda pa a puntare l'oggetto della classe B
    cout << "Funzioni dell'oggetto della classe A dopo pa=pb\n";
    pa -> Dinamica();
    pa -> Statica();
}
```

**Esecuzione**

DINAMICA della classe A

STATICA della classe A

Funzioni dell'oggetto della classe B

DINAMICA della classe B

STATICA della classe B

Funzioni dell'oggetto della classe A dopo pa=pb

DINAMICA della classe A

STATICA della classe A

Dall'esempio si può osservare che dopo aver mandato il puntatore `pa` a puntare l'oggetto della classe `B` (assegnamento `pa=pb`), dereferenziandolo si fa partire i metodi della classe `A` che `b` ha ereditato e occultato, non i metodi specifici dell'oggetto `b` che era nostro intendimento far partire. Questo accade perché il puntatore `pa` è stato assegnato dal compilatore (anzi, dal linker) a un oggetto della classe `A`, non della classe `B`, e quindi, dereferenziandolo, si accede agli elementi della classe base originale e non a quelli aggiuntivi dell'oggetto della classe derivata. È avvenuto quello che si chiama "binding statico".

#### 12.5.1 Binding dinamico e metodi virtuali

Per binding si intende la connessione tra la chiamata di una funzione e il codice che la implementa (ovvero l'indirizzo in memoria della sua prima istruzione a livello macchina). Il binding può essere:

- *statico*, se il collegamento viene fatto dal "linker" in fase di compilazione, quando associa un indirizzo di memoria al nome della funzione o del metodo della classe;
- *dinamico*, se la connessione avviene durante l'esecuzione dereferenziando un puntatore che ha come valore l'indirizzo della funzione o del metodo.

Nel binding statico, tutti i riferimenti vengono risolti a tempo di compilazione (quindi anche quelli dei metodi dell'oggetto della classe base); il compilatore e il linker determinano direttamente la posizione fissa del codice che deve essere eseguito alla chiamata della funzione. Nel binding dinamico il codice da eseguire verrà determinato solo all'atto della chiamata. Solo l'esecuzione del programma (nel caso che stiamo trattando il valore di un puntatore a una classe base) determinerebbe il binding effettivo tra le diverse possibilità (una per ogni classe derivata). Il vantaggio del binding dinamico sarebbe quello di offrire un alto grado di flessibilità e praticità nella gestione delle gerarchie di classi, con lo svantaggio di dover sempre dereferenziare un puntatore per sapere quale codice far partire, con conseguente leggera inefficienza temporale. Nel C++ il binding è statico, per motivi di efficienza, ma c'è un modo per implementare il binding dinamico, e cioè quello di utilizzare la parola riservata `virtual` anteposta alla dichiarazione del metodo nella classe base.

#### Esempio 12.14

```
class A {
public:
    virtual void Dinamica() {cout << "DINAMICA della classe A\n";}
    void Statica() {cout << "STATICA della classe A\n";}
};

class B : public A {
public:
    void Dinamica() { cout << "DINAMICA della classe B\n"; }
    void Statica() { cout << "STATICA della classe B\n"; }
```

```

};

int main() {
    A a; A* pa = &a;
    B b; B* pb = &b;
    cout << "Funzioni dell'oggetto della classe A\n";
    pa -> Dinamica();
    pa -> Statica();
    cout << endl;
    cout << "Funzioni dell'oggetto della classe B\n";
    pb -> Dinamica();
    pb -> Statica();
    cout << endl;
    pa = pb; // manda pa a puntare l'oggetto della classe B
    cout << "Funzioni dell'oggetto della classe A dopo pa=pb\n";
    pa -> Dinamica();
    pa -> Statica();
}

```

## Esecuzione

DINAMICA della classe A

STATICA della classe A

Funzioni dell'oggetto della classe B

DINAMICA della classe B

STATICA della classe B

Funzioni dell'oggetto della classe A dopo pa=pb

DINAMICA della classe B

STATICA della classe A

Questa volta, dopo aver assegnato a pa il valore del puntatore alla classe B (pa=pb), il binding relativo al metodo dichiarato normalmente rimane statico, quindi dereferenziando il puntatore si fa partire quello della classe base (perché pa è puntatore alla classe base), mentre il binding relativo al metodo dichiarato virtual è dinamico, quindi dereferenziando il puntatore si fa partire quello della classe derivata! Si implementa così un *metodo dinamico*. In pratica, la parola riservata *virtual* prima della dichiarazione del metodo nella classe base indica al compilatore che esso può essere mandato in esecuzione mediante dereferenziamento di un puntatore e il **tipo** dell'oggetto, ovvero la sua **classe**, sarà specificato solo a tempo d'esecuzione.

## 12.6 Vantaggi del polimorfismo

Il polimorfismo permette quindi di utilizzare una stessa interfaccia per lavorare con oggetti di diverse classi. Esso si può implementare con un array di puntatori a oggetti di classi diverse che condividono il nome di un metodo.

Se le classi sono derivate da una base comune bisogna dichiarare il metodo condiviso come *virtual* nella classe base per poter fruire dei benefici del binding dinamico. Ovvero in C++ si dovrebbero seguire le seguenti regole:

1. creare una gerarchia di classi con l'interfaccia comune implementata da metodi dichiarati *virtual* nella classe base;
2. differenziare i metodi *virtual* nelle classi derivate;
3. riferire gli oggetti delle classi derivate tramite riferimenti o puntatori alla classe base.

Il polimorfismo rende il sistema più flessibile. Il C++ conserva i vantaggi della compilazione statica mentre altri linguaggi object-oriented adottano un polimorfismo assoluto, che si scontra però con l'idea della verifica dei tipi; essi offrono cioè flessibilità e libertà, ma tendono ad aggirare le verifiche del codice sorgente, rimandandole al run-time. Si ha più libertà ma anche più probabilità di errore a seguito di chiamate a funzioni indefinite o mal definite (per ragioni che vanno dal semplice errore di sintassi all'errore logico-concettuale) e il programma tende a essere più lento a causa delle verifiche fatte durante l'esecuzione. Per esempio, nello Smalltalk, che ignora la verifica statica dei tipi, un messaggio può non avere un metodo corrispondente, ma l'errore apparirà solo se e quando si tenterà di eseguire l'istruzione difettosa.

Anche se in C++ v'è meno libertà, il polimorfismo rimane uno strumento potente. Le sue applicazioni più frequenti sono:

- **gestione delle gerarchie di classi:** si aumenta l'efficienza della sottoclasse conservando un alto grado di flessibilità;
- **strutture di dati eterogenei:** consente di manipolare in maniera comoda e sicura insiemi di oggetti simili ma rappresentati da strutture di dati eterogenee.

In C++, le funzioni virtuali devono essere dichiarate come *virtual* nella classe di livello più alto della gerarchia.

### Ereditarietà

In questo capitolo abbiamo introdotto due importanti relazioni fra le classi: l'ereditarietà e il polimorfismo. L'ereditabilità di metodi e attributi riduce la ridondanza del codice ed è un requisito essenziale di ogni linguaggio orientato agli oggetti. Una classe creata a partire da un'altra già esistente utilizzando l'ereditarietà, si dice *classe derivata* o *sottoclasse*, mentre la classe genitrice si dice *classe base* o *superclasse*. Si parla di ereditarietà *privata* quando si impedisce agli utenti della classe derivata di accedere ai membri della classe base. L'ereditarietà è

*multiplo* quando una classe deriva da due o più classi base. Benché sia uno strumento potente, può creare problemi quando nomi identici appaiono in più di una classe base.

Il *polimorfismo* è la proprietà per cui una stessa invocazione può significare cose diverse a seconda dell'oggetto invocato. Il polimorfismo mediante metodi virtuali è utile per manipolare oggetti di classi derivate tramite operazioni definite nella classe base. Ogni classe derivata può implementare diversamente le operazioni definite nella classe base.

- Classe astratta
- Classe base
- Classe derivata
- Costruttore
- Dichiarazione di accesso
- Distruttore
- Specificatori di accesso
- Funzione virtuale

- Ereditarietà
- Ereditarietà multipla
- Ereditarietà pubblica e privata
- Ereditarietà semplice
- Binding dinamico
- Polimorfismo
- Relazione *is-a*

## ESERCIZI

**E1** Definire una classe base persona che contenga un'anagrafica generale comune a tutte le persone (nome, indirizzo, data di nascita, sesso ecc.). Progettare una gerarchia di classi che contempli le classi seguenti: `studente`, `impiegato`, `studente_impiegato`. Scrivere un programma che legga da file informazioni per un vettore di persone: a) generale; b) studenti; c) impiegati; d) studenti impiegati. Il programma deve permettere di ordinare alfabeticamente per cognome.

**E2** Implementare una gerarchia `Biblioteca` che abbia almeno una dozzina di classi (collezioni di libri di letteratura, scienze umane, tecnologia ecc.).

**E3** Implementare una gerarchia `Impiegato` di un qualunque tipo di impresa. La gerarchia deve avere almeno quattro livelli, con ereditarietà di membri dato, e metodi, i metodi debbono poter calcolare salari, licenziamenti, promuovere, licenziare, mandare in pensione ecc. I metodi debbono permettere anche di calcolare aumenti salariali agli `Impiegati` in base alla loro categoria e ruolo. La gerarchia deve poter essere utilizzata per fornire differenti tipi di accesso agli `Impiegati` per almeno quattro tipi differenti di accesso.

**E4** Scrivere una classe `FigGeometrica` che rappresenti figure geometriche tipo *punto*, *riga*, *rettangolo*, *triangolo* e simili. Deve

fornire metodi che permettano di disegnare, ampliare, muovere e distruggere tali oggetti. La gerarchia deve consistere di almeno una dozzina di classi.

**E5** Implementare una gerarchia di tipi di dato numerici che estenda i tipi di dato fondamentali come `int` e `float`, disponibili in C++. Le classi da progettare possono essere `Complesso`, `Frazione`, `Vettore`, `Matrice`, con i relativi metodi (operazioni su quel tipi).

**E6** Implementare una gerarchia di ereditarietà di animali che contenga almeno sei livelli di derivazione e dodici classi.

**E7** Progettare la seguente gerarchia di classi:

```
Persona
  Nome
  età
  visualizzare()
```

Studente	Professore	
nome	ereditato nome	ereditato
età	ereditato età	ereditato
id	definito	salario
<code>visualizzare()</code> ridefinito <code>visualizzare()</code> ereditata		

Scrivere un programma che gestisca la gerarchia di classi, legga un oggetto di ogni classe e lo visualizzi a schermo:

- senza utilizzare funzioni virtuali;
- utilizzando funzioni virtuali.

**Esercizio** Il seguente programma mostra le differenze tra le chiamate a una funzione virtuale e a una funzione normale.

```
class Base {
public:
    virtual void f() { cout << "f(): classe base!" 
        << endl; }
    void g() { cout << "g(): classe base!" <<
        endl; }
};

class Derivata1 : public Base {
public:
    virtual void f() { cout << "f(): classe Deri-
        vata!" << endl; }
    void g() { cout << "g(): classe Derivata!" <<
        endl; }
};

class Derivata2 : public Derivata1 {
public:
    virtual void f() { cout << "f(): classe Deri-
        vata2!" << endl; }
    void g() { cout << "g(): classe Derivata2!" <<
        endl; }
};

int main()
{
    Base b;
    Derivata1 d1;
```

```
Derivata2 d2;
Base *p = &b;
p->f();
p->g();
p= &d1;
p->f();
p->g();
p= &d2;
p->f();
p->g();
```

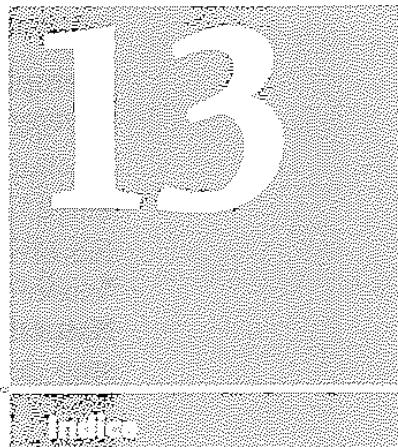
Qual è il risultato dell'esecuzione di questo programma? Perché?

**Esercizio** Dichiare le classi Punto\_unidimensionale, Punto\_bidimensionale e Punto\_tridimensionale, per trattare punti negli spazi mono, bi e tridimensionale, mediante una gerarchia di classi.

**Esercizio** Dichiare la classe astratta Figura con le funzioni virtuali Area e perimetro che abbia come classi derivate le figure geometriche cerchio, quadrato e rettangolo.

**Esercizio** Progettare una gerarchia di classi con la classe base Sport, dalla quale derivano le classi Calcio, Pallacanestro, Pallavolo, con binding dinamico.  
anche qui il testo è da riprendere da questo file

# Template



- 13.1 Genericità
- 13.2 Template in C++
- 13.3 Template di funzione

- 13.4 Template di classe
- 13.5 Template e polimorfismo

13.1-13.5

## Introduzione

Una delle idee chiave nel mondo della programmazione è la possibilità di progettare classi e funzioni che lavorino su *tipi generici*. La maggior parte dei linguaggi per l'OOP supportano la genericità. Anche C++ lo fa mediante i *template*; questi si uti-

lizzano per implementare strutture e algoritmi indipendenti dal tipo degli oggetti su cui operano. Per esempio, un template Stack può descrivere come implementare una pila d'oggetti arbitrari: numeri reali, stringhe, interi, puntatori ecc.

### 13.1 Genericità

La genericità è una tecnica di programmazione che agevola il riutilizzo del software. Essa permette di definire una classe (o una funzione) senza specificare il tipo di dato di uno o più dei suoi membri (o parametri). In questo modo si potrà riutilizzare la classe per altri usi senza doverla riscrivere.

La genericità sfrutta il fatto che gli algoritmi di risoluzione di numerosi problemi non dipendono dal tipo di dato da elaborare. Per esempio, un algoritmo che gestisce una pila di caratteri sarà essenzialmente lo stesso che gestisce una pila di interi o di qualunque altro tipo, ma in vecchi linguaggi procedurali come Pascal, C e COBOL è necessario sviluppare un programma diverso per ogni tipo di dato da inserire nella pila. Nei linguaggi OOP, come C++ e Java (come pure nel "vecchio" Ada), esistono i *template*; essi permettono di definire *classi generiche* (o *parametriche*) che implementano strutture e classi indipendenti dal tipo dell'elemento da processare. I template dovranno poi essere istanziati dall'utente per produrre sottoprogrammi o classi che lavorino con specifici tipi di dato. Queste classi generiche si denominano anche **container**, ed esempi tipici sono *pile*, *code*, *liste*, *insiemi*, *dizionari* ecc.

### 13.2 Template in C++

I template furono proposti da Stroustrup<sup>1</sup> nella conferenza USENIX C++ di Denver, nel 1988, quando pose le seguenti domande.

- Può essere utile parametrizzare i tipi?
- Il C++ può integrare tipi parametrizzati senza alterare sensibilmente la velocità di compilazione?
- Il compilatore continuerà a essere portabile?

Naturalmente Stroustrup rispondeva "sì" a tutte quelle domande. Il meccanismo dei template presentato nell'*Annotated Reference Manual* nel luglio del 1990 fu accettato dal comitato ANSI C++, e la versione 3.0 del C++ di AT&T introdusse la nozione di *template* come costrutto per scrivere funzioni e classi generiche da applicarsi a dati di tipo diverso. Vi sono due tipi di template: *template di classe* e *template di funzione*.

L'interesse nei template nasce dal fatto che questa generalità non implica perdita di rendimento e non obbliga a sacrificare i vantaggi del C++ in tema di controllo stretto dei tipi di dato.

Così come una classe è un modello per istanziare oggetti (della classe) a tempo d'esecuzione, un *template* è un modello per istanziare classi o funzioni (del template) a tempo di compilazione. I template sono quindi funzioni e classi generiche, implementate per un tipo di dato da definirsi in seguito. Per utilizzarli il programmatore deve solo specificare i tipi con i quali essi debbono lavorare.

### 13.3 Template di funzione

Il concetto di *template di funzione* è stato introdotto nel Paragrafo 5.13. Nei linguaggi di programmazione, molto spesso, le funzioni debbono essere ridefinite al cambiare dei parametri. Per esempio, la procedura che scambia tra loro i valori di due variabili si deve re-implementare per ogni tipo di coppia di dati. Per dati di tipo intero sarebbe:

```
void scambia (int& m, int& n)
{
    int aux = m;
    m = n;
    n = aux;
}
```

ma per valori di tipo char si deve sovraccaricare la procedura così:

```
void scambia (char& m, char& n)
{
    char aux = m;
    m = n;
    n = aux;
}
```

---

<sup>1</sup> Stroustrup B., *Parametrized Types for C++*, Proceeding Acts, USENIX C++ Conference, Denver, ottobre 1988.

se poi vogliamo applicarla a coppie di variabili di tipo double o float, si dovrà scrivere una terza e una quarta definizione della procedura quasi identiche a quelle precedenti.

L'idea base del template di funzione è quella di essere una funzione alla quale si possa passare parametri di qualunque tipo, un insieme indeterminato di funzioni sovraccaricate che descrive l'algoritmo specifico di una funzione generica. Ogni funzione specifica di questo insieme è un'istanza del template di funzione e viene prodotta automaticamente se e quando necessario. Si può sovraccaricare un template di funzione con una funzione normale o con altri template di funzioni.

Per meglio capire il vantaggio si pensi che, a fronte di un'unica abs() del C++, il C schiera abs(), fabs(), labs() e cabs() per i vari tipi numerici di cui restituire il valore assoluto (Figura 13.1).

L'idea è di rappresentare il tipo di dato utilizzato dalla funzione con un nome che rappresenti "qualunque tipo". Normalmente, questo "qualunque tipo" si rappresenta con T, ma va bene anche qualsiasi identificatore diverso da una parola riservata. Per specificare che si tratta di un tipo generico si può utilizzare o la parola riservata class (quella che propose inizialmente da Stroustrup per non creare una nuova parola riservata) o la parola typename (che venne proposta in seguito per disambiguare l'utilizzo della parola class). La definizione comincia con la parola riservata template seguita da una lista di parametri del template separati da virgole e racchiusi tra parentesi angolari. *La lista di parametri non può essere vuota* e si può avere più di un parametro di tipo.

Vediamo per esempio due template per restituire il minimo e il massimo di due valori.

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

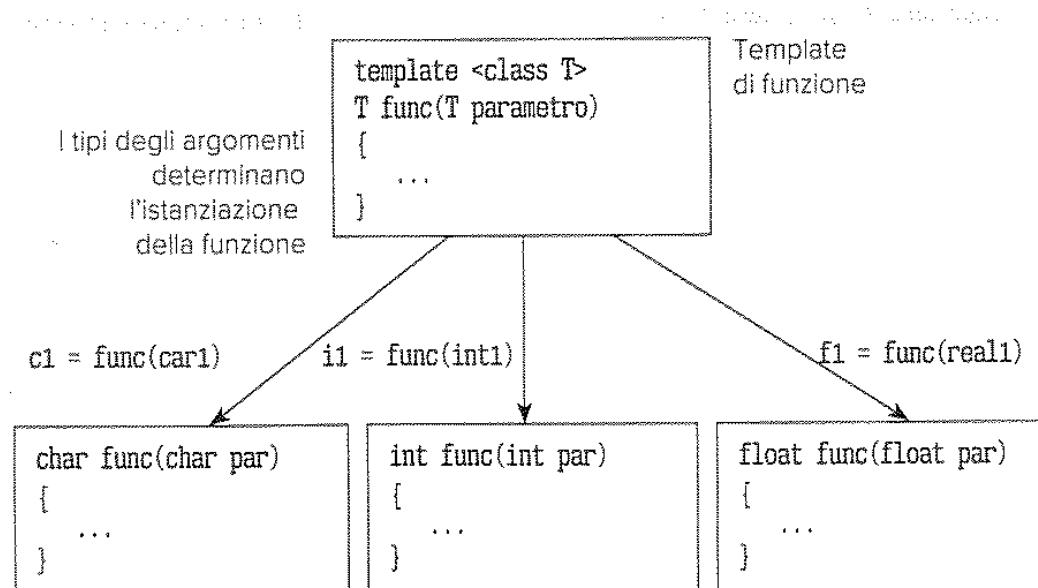


Figura 13.1: Template di funzione.

Quando il compilatore trova una chiamata della forma `min(a, b)`, istanzia il tipo generico `T` al tipo da parametri `a` e `b`. Il seguente template tiene utilmente conto del fatto che i parametri non verranno modificati dalla funzione:

```
template <typename T> const T& max(const T& a, const T& b)
{ return a > b ? a : b;}
```

Per utilizzare un template di funzioni, si deve specificarne un'istanza fornendo argomenti per ogni parametro del template elencandoli dentro le parentesi angolari. Esempi di chiamate sono:

```
long x = max <long> (40, 50);
int x = max (40, 50);
```

### Esempio 13.1

Mostrare il funzionamento completo di un template di funzione per lo scambio dei valori di due variabili.

```
template <typename T> void scambia (T& v1, T& v2)
{
    T aux = v1;
    v1 = v2;
    v2 = aux;
}
int main()
{
    int numero1 = 5, numero2 = 8;
    cout << "valori originali: " << numero1 << " " << numero2 << endl;
    scambia(numero1, numero2);
    cout << "valori scambiati: " << numero1 << " " << numero2 << endl;
    char car1 = 'L', car2 = 'J';
    cout << "valori originali: " << car1 << " " << car2 << endl;
    scambia (car1, car2);
    cout << "valori scambiati: " << car1 << " " << car2 << endl;
    return 0;
}
```

### Esecuzione

```
valori originali: 5 8
valori scambiati: 8 5
valori originali: L J
valori scambiati: J L
```

### Esempio 13.2

Scrivere una funzione per confrontare due valori restituendo -1, +1 o 0 a seconda che il primo sia minore, uguale o maggiore del secondo.

```
template <typename T> int confrontare (const T &v1, const T &v2)
{
```

```

if (v1 < v2) return -1;
if (v2 < v1) return 1;
return 0;
}

```

Si può anche specificare un puntatore a tipo generico ( $T^*$  *parametro*) o un riferimento ( $T&$  *parametro*). La funzione può avere più di un parametro formale e restituire un valore di tipo parametrico. Per esempio:

```

template < typename T> T& f(T parametro)
{
    // corpo della funzione
}

template < typename T> T f(int a, T b)
{
    // corpo della funzione
}

template < typename T> T f(T a, T b)
{
    // corpo della funzione
}

```

Si possono utilizzare anche tipi parametrici diversi, per esempio:

```

template < typename T1, typename T2> T1 f(T1 a, T2 b)
{
    // corpo della funzione
}

```

Anche i template di funzione si possono dichiarare extern, inline o static. Lo specificatore si colloca dopo la riga dei parametri formali (mai prima della parola riservata template).

```

template < typename T> inline T f(T a, T b) // corretta
{
    // corpo della funzione
}

extern template < typename T> T f(T a, T b) // scorretta
{
    // corpo della funzione
}

```

### Esempio 13.3

La funzione template si può dichiarare anche in un header file:

```

min.h
template < typename T> T min(const T a, const T b)
{
    return a < b ? a : b;
}

```

con le seguenti chiamate:

```
#include "min.h"
int i, j, k;
double u, v, w;
char a, b, c;

...
k = min(i, j);
w = min(u, v);
c = min(a, b);
```

C++ genera tre funzioni template con i seguenti prototipi:

```
int min(const int a, const int b);
char min(const char a, const char b);
double min(const double a, const double b);
```

#### Esempio 13.4

minmax.h

```
#ifndef _MINMAX_H
#define _MINMAX_H //evita #include multipli
template <class T> T max(T a, T b)
{
    if(a > b) return a;
    else return b;
}

template <class T> T min(T a, T b)
{
    if(a < b) return a;
    else return b;
}
#endif // _MINMAX_H
```

```
#include "minmax.h"
int max(int a, int b);
double max(double a, double b);
char max(char a, char b);
int main()
{
    int i1 = 100, i2 = 200;
    double d1 = 3.141592, d2 = 2.718283;
    cout << "max(i1, i2) è uguale a: " << max(i1, i2) << '\n';
    cout << "max(d1, d2) è uguale a: " << max(d1, d2) << '\n';
    cout << "max(c1, c2) è uguale a: " << max(c1, c2) << '\n';
    return 0;
}
```

**Esempio 13.5**

I template di funzione si utilizzano molto per ordinare vettori o cercare elementi in vettori. Si tratta di un algoritmo che vedremo nel Capitolo 16 (ShellSort), ma qui anticipiamo il concetto perché illustra come un template di funzione (*ordina*) possa utilizzare un altro template di funzione (*scambia*):

```
template <typename T> void scambia (T& v1, T& v2)
{
    T aux = v1;
    v1 = v2;
    v2 = aux;
}

template <typename T> void ordina(T* v, int n)
{
    for (int salto = n/2; salto > 0; salto /=2)
        for(int i = salto; i < n; i++)
            for(int j= i-salto; j>=0 && v[j+salto] < v[j]; j -= salto)
                scambia(v[j], v[j+intervallo]);
}

int main()
{
    int vettore_int[10]=[2,5,4,3,6,4,8,1,9,0];
    char vettore_car[7]={'a','g','e','r','w','x','c'};
    ordina(vettore_int, 10);
    ordina(vettore_car, 7);
    for (int i=0; i<10;i++) cout << vettore_int[i] << ' ';
    cout << endl;
    for (int i=0; i<7 ;i++) cout << vettore_car[i] << ' ';
    cout << endl;
    return 0;
}
```

**Esecuzione**

```
0 1 2 3 4 4 5 6 8 9
a c e g r w x
```

Con riferimento al template di funzione dell'Esempio 13.1, supponiamo di effettuare le due chiamate:

```
cout << max(4, 5.9) << endl;
cout << max('A', 66) << endl;
```

si generano errori in fase di compilazione perché si istanzia il tipo del primo argomento che però poi non collima con quello del secondo. Una possibile soluzione è quella di definire un nuovo template di funzione che ammetta

la possibilità che i tipi possano essere diversi. Quando i tipi sono più di uno, al posto di typename si deve usare la parola riservata class.

#### Esempio 13.6

```
template <class T1, class T2> T1 max(T1 primo, T2 secondo)
{
    return primo > (T1) secondo ? primo : (T1) secondo;
}

int main()
{
    cout << max(4, 5.9) << endl;
    cout << max('A', 66) << endl;
    return 0;
}
```

#### Esecuzione

```
5
B
```

### 13.4 Template di classe

I template di classe permettono di definire classi parametriche che possono gestire differenti tipi di dato. Questi *container* possono rappresentare vettori, strutture, liste ordinate, tabelle di hash per creare strutture dati astratte e complesse come pile, code, alberi, grafi ecc.

#### 13.4.1 Definizione di un template di classe

La sintassi di un template di classe è un'intestazione della dichiarazione di classe seguita da una dichiarazione o definizione di classe.

```
template <typename T> class nometipo {
    ...
};
```

dove T è il nome del tipo utilizzato dal template e *nometipo* (per esempio, *Pila*) è il nome del tipo parametrizzato del template. T non è limitato a tipi di dato predefiniti (int, char, float ecc.) ma contempla anche tipi definiti dall'utente.

Così come nei template di funzione, il codice viene sempre preceduto da un'istruzione nella quale si dichiara T come parametro di tipo, e possono esserci più parametri tipo anche nei template di classe. Per esempio:

```
template <typename T>
struct Punto {
    T x, y;
```

};

Per utilizzare il template di classe, si deve fornire un argomento per ogni tipo parametrico del template (in questo caso solo un tipo):

```
Punto<int> pt = {45, 15};
```

Il template per una classe generica Pila - la *pila* (o *stack*) - è una struttura dati in cui l'ultimo dato che entra è il primo a uscire (la vedremo meglio nel Capitolo 17); si potrebbe scrivere così:

```
template <typename T> class Pila
{
    T dati[50];
    int elementi;
public Pila(): elementi(0) {} // costruttore
void immettere(T elem); // aggiunge un elemento alla pila
T estrarre(); // ottiene un elemento dalla pila
int numero(); // numero di elementi reali nella pila
bool vuota(); // è vuota la pila?
};
```

Pila è una classe parametrizzata con il tipo parametrico T. Con questa definizione del template di classe Pila si possono creare stack di differenti tipi di dato:

```
Pila <int> pila_int; // Una pila per variabili int
Pila <float> pila_reale; // Una pila per variabili float
```

Analogamente, si può definire una classe generica Array come segue:

```
template < typename T > class Array
{
public :
    Array(int n = 16) {pa = new T[_lunghezza = n];}
    ~Array() {delete[] pa;}
    T& operatore[] (int i);
// ...
private :
    f* pa;
    int lunghezza;
};
```

Si possono poi creare diversi tipi di array così:

```
Array <int> iArray(128); // Array di 128 elementi int
Array <float> fArray(32); // Array di 32 elementi float
```

La seguente specifica del template di classe coda contiene due parametri di template:

```
template <class elem, int dimensione> class coda {
    int dimensione;
    ...
public:
    coda(int n);
    bool vuota();
    ...
};
```

elem, che specifica il tipo dell'elemento della coda, e dimensione (di tipo int) che specifica la dimensione della coda. Oggetti di tipo coda possono essere definiti così:

```
coda <int, 2048> a;
coda <char, 512> b;
coda <char, 1024> c;
coda <char, 512*2> d;
```

Due template di classe si riferiscono alla stessa classe, se e solo se i nomi sono identici e i loro argomenti hanno gli stessi valori. Per esempio, gli oggetti c e d appartengono alla stessa classe.

L'implementazione di un template di classe richiede costruttore, distruttore e metodi. La definizione di un costruttore di template ha il formato:

```
template <dichiarazioni-parametri-template>
        nome-classe <parametri-template>:: nome_classe
{
    // ...
}
```

Per esempio, il corpo del costruttore del template coda sarà:

```
template <class elem, int dimensione>
        coda <elem, dimensione>::coda(int n)
{
    ...
}
```

Le definizioni dei distruttori sono simili a quelle dei costruttori.

La definizione di un metodo di un template di classe ha il seguente formato:

```
template <dichiarazioni-parametri-template> tipo-risultato
        nome-classe <parametri-template>::
        nome-funz-membro(dichiarazioni-parametri)
{
    // ...
}
```

Per esempio, si può definire la funzione vuota della classe template coda:

```
template <class elem, int dim> bool coda <elem, dim>::vuota()
{
    // ...
}
```

La prima volta che il compilatore incontra una definizione di un oggetto di un template di classe, per esempio:

```
coda <int, 2048> a;
```

prende gli elementi del template di classe e costruisce automaticamente una classe istanziandone l'oggetto.

Riassumendo, i tre passi per la gestione di un template di classe sono:

- dichiarazione del template;
- implementazione del template;
- creazione di un'istanza specifica del template.

#### Esempio 13.7

```
#define MaxElementi 10
// dichiarazione del template Pila (in inglese Stack)
template <typename Tipo> class Pila {
public:
    Pila();
    bool immettere(const Tipo); // mettere elemento nella pila
    bool estrarre(Tipo&); // tirare elemento fuori dalla pila
private :
    Tipo elementi[MaxElementi]; // elementi di pila
    int cima; // cima della pila
};

// costruttore della Pila
template <typename Tipo> Pila <Tipo>:: Pila() {
    cima = -1;
}

// funzione metodo immettere (in inglese Push)
template <typename Tipo> bool Pila <Tipo>:: immettere(const Tipo elem) {
    if(cima < MaxElementi -1) {
        elementi[++cima] = elem;
        return true;
    }
    else return false;
}

// funzione metodo estrarre (in inglese Pop)
template <typename Tipo> bool Pila <Tipo> :: estrarre(Tipo& elem) {
    if (cima < 0) return false;
    else {
```

```

        elem = elementi[cima--];
        return true;
    }
}

// creazione di istanze specifiche di Pila
int main() {
    char c;
    int i;
    float f;
    Pila <char> pila_car;          // pila di caratteri
    Pila <int> pila_int;          // pila di interi
    Pila <float> pila_float;      // pila di reali
    pila_car.immettere('a'); pila_car.estrarre(c); cout << c << endl;
    pila_int.immettere(33); pila_int.estrarre(i); cout << i << endl;
    pila_float.immettere(3.14); pila_float.estrarre(f); cout << f << endl;
    return 0;
}

```

### Esecuzione

```

a
33
3.14

```

Questi *container* sono molto versatili. Per esempio, si può riscrivere la classe template Pila senza dover specificare un numero fisso di elementi. Si può anche aggiungere facilmente una funzione per verificare se la pila è piena. L'interfaccia della classe è rappresentata nell'esempio seguente.

### Esempio 13.B

Progettare un template di classe Pila, con separazione tra header file (pila.h), file di implementazione (pila.cpp) e applicazione (esercizio.cpp).

pila.h

```

// interfaccia di un template di classe per definire pile
// di dimensione MAX di default pari a 100

enum stato_pila {OK, PIENA, VUOTA};

template <typename T, int dim = 100> class Pila
{
public:
    Pila(): cima(0), stato(VUOTA) {tabella = new T[dim];}
    ~Pila() {delete[] tabella;}
    void immetti(T);
    T estrai();
    void visualizza();
    int num_elementi();

```

```

int leggere_lung() {return dim;}
private:
T* tabella;
int cima;
stato_pila stato;
};

```

La definizione dei metodi utilizza una sintassi più complessa per specificare i due parametri del template:

pila.cpp

```

// definizione dei metodi di un template di classe Pila
#include "pila.h"
template <typename T,int dim> void Pila <T,dim>::immetti(T elemento) {
    if (stato!= PIENA) tabella [++cima] = elemento;
    else cout << "**** Pila piena ****" << endl;
    if (cima >= dim) stato = PIENA;
    else stato = OK;
}
template <typename T, int dim> T Pila <T, dim> :: estrai() {
    T elemento = 0;
    if (stato!= VUOTA) elemento = tabella[cima--];
    else cout << "**** Pila vuota ****" << endl;
    if (cima <= 0) stato = VUOTA;
    else stato = OK;
    return elemento;
}
template <typename T, int dim> void Pila <T, dim>::visualizza() {
    for (int i = cima; i > 0; i--)
        cout << "[" << tabella[i] << "]" << endl;
}
template <typename T, int dim> int Pila<T, dim>:: num_elementi() {
    return cima;
}

```

Una proprietà interessante di questa classe è che il secondo parametro si può omettere, utilizzando il valore di default, come nel seguente esempio:

esercizio.cpp

```

// esempio d'utilizzo di un template di classe Pila con numero di
// numero generico di elementi

#include "pila.cpp"
int main()
{
    Pila <int> p1; // Pila di 100 interi
    p1.immetti(6);

```

```

p1.immetti(12);
p1.immetti(18);
cout << "Numero di elementi inseriti: " << p1.num_elementi() << endl;
p1.visualizza();
cout <<"Estrae 1 : " << p1.estrai() << endl;
cout <<"Estrae 2 : " << p1.estrai() << endl;
cout <<"Estrae 3 : " << p1.estrai() << endl;
cout <<"Numero di elementi rimasti: " << p1.num_elementi() << endl;
cout <<"Estrae 4 : " << p1.estrai() << endl;
cout << endl ;
Pila <char,25> p2; // Pila di 25 caratteri
p2.immetti('a');
p2.immetti('b');
p2.immetti('c');
cout << "Numero di elementi inseriti: " << p2.num_elementi() << endl;
p2.visualizza();
cout <<"Estrae 1 : " << p2.estrai() << endl;
cout <<"Estrae 2 : " << p2.estrai() << endl;
cout <<"Estrae 3 : " << p2.estrai() << endl;
cout <<"Numero di elementi rimasti: " << p2.num_elementi() << endl;
cout <<"Estrae 4 : " << p2.estrai() << endl;
cout << endl ;
Pila <double,5> p3; //Pila di 5 double
p3.immetti(6.6);
p3.immetti(12.12);
p3.immetti(18.18);
cout <<"Numero di elementi inseriti: " << p3.num_elementi() << endl;
p3.visualizza();
cout << "Estrae 1 : " << p3.estrai() << endl;
cout << "Estrae 2 : " << p3.estrai() << endl;
cout << "Estrae 3 : " << p3.estrai() << endl;
cout << "Numero di elementi rimasti: " << p3.num_elementi() << endl;
cout << "Estrae 4 : " << p3.estrai() << endl;
}

```

### Esecuzione

Numero di elementi inseriti: 3

[18]

[12]

[6]

Estrae 1 : 18

Estrae 2 : 12

Estrae 3 : 6

Numero di elementi rimasti: 0

Estrae 4 : \*\*\* Pila vuota \*\*\*

0

```
Numero di elementi inseriti: 3
```

```
[c]
[b]
[a]
```

```
Estrae 1 : c
```

```
Estrae 2 : b
```

```
Estrae 3 : a
```

```
Numero di elementi rimasti: 0
```

```
Estrae 4 : *** Pila vuota ***
```

```
Numero di elementi inseriti: 3
```

```
[18.18]
[12.12]
[6.6]
```

```
Estrae 1 : 18.18
```

```
Estrae 2 : 12.12
```

```
Estrae 3 : 6.6
```

```
Numero di elementi rimasti: 0
```

```
Estrae 4 : *** Pila vuota ***
```

```
0
```

## 13.5 Template e polimorfismo

Spesso ci si chiede quali siano le differenze fra template e polimorfismo, se si può implementare il secondo con un template, quale soluzione è migliore ecc. Per dare una risposta richiamiamo entrambi i concetti.

Una funzione è *polimorfa* se almeno uno dei suoi parametri può supportare tipi di dato differenti. Qualunque funzione che abbia un parametro come puntatore a una classe può essere una funzione polimorfa e si può utilizzare con tipi di dato diversi.

Una funzione è un *template* solo se è preceduta dalla parola riservata *template*. Questo significa che scrivere una funzione template implica pensare in astratto, evitando qualunque dipendenza da tipi di dato, costanti numeriche ecc. Una funzione template è solo un modello e non una vera funzione. L'istanziazione la fa automaticamente il compilatore a ogni differente chiamata, con il risultato di disporre di una famiglia di funzioni sovraccaricate che hanno tutte lo stesso nome e parametri diversi. In altre parole, la clausola *template* è un *generatore automatico di funzioni sovraccaricate*.

Le funzioni polimorfiche sono funzioni che possono essere eseguite *dinamicamente* con parametri di tipo diverso, mentre le funzioni template sono funzioni con tipi parametrizzati che si compilano *staticamente* in versioni differenti per raccordarsi ai tipi di dato per cui vengono mandate in esecuzione.

Da un punto di vista pratico si può osservare che i template tendono a generare un codice eseguibile grande, poiché duplicano le funzioni.

Spesso, quando si definisce una classe o una funzione la si vorrebbe poter utilizzare con oggetti di tipo diverso, senza dover riscrivere il codice ogni volta. Abbiamo visto in questo capitolo che le ultime versioni del C++ incorporano i *template*; essi permettono di dichiarare una classe senza specificare il tipo di uno o più membri dato, e permettono di definire una funzione senza specificare il tipo di uno o più parametri.

I template di funzione cominciano con la parola riservata `template`, seguita da un vettore di parametri formali racchiusi tra parentesi angolari (`< >`); ogni parametro formale deve essere preceduto dalla parola riservata `typename`, per esempio:

```
template< typename T >
template< typename TipoContorno, class TipoPieno>
template< typename T > void funzione(TipoContorno, TipoPieno, T);
```

mentre i template di classe si dichiarano così:

```
template< typename T > class Demo
{
    // ...
    T v;
};
```

```
public:
    ...
    Demo (const T & val):v(val) {}
    ...
};
```

Si generano *istanze* di un template di classe quando si dichiarano oggetti della classe specificando un tipo specifico. Per esempio, la dichiarazione

```
Demo <short> dc;
```

fa sì che il compilatore generi una dichiarazione della classe nella quale ogni occorrenza del tipo di parametro `T` nel template è rimpiazzata dal tipo reale `short` nella dichiarazione della classe. In questo caso, il nome della classe è `Demo<short>` e non `Demo`.

La genericità permette il riutilizzo del codice, semplificando il compito del programmatore e rendendo i programmi più affidabili. C++ ha librerie standard che includono versioni di template per compiti frequenti come ricerca e ordinamento; questi template sono detti *container*.

### ESERCIZI DI APPRENDIMENTO

- **Tipo generico**
- **Concetti fondamentali di template in C++**
- **Funzioni di template**

- **Genericità**
- **Template di funzioni**
- **Template contro polimorfismo**
- **Tipo parametrizzato**

## Esercizi

**Esercizio 1** Definire due template di funzione `min()` e `max()` che restituiscano il minore e il maggiore di due valori.

**Esercizio 2** Scrivere un programma che utilizzi le funzioni template dell'esercizio precedente per calcolare i valori massimi di copie di `int`, di `double` e di `char`.

**Esercizio 3** Definire un template di classe che possa immagazzinare un piccolo database di documenti.

**Esercizio 4** Scrivere un programma che utilizzi il template dell'esercizio precedente per creare un oggetto della classe database.

Esercizio 1 Definire un template di funzione `tscambio` che scambi due variabili di qualunque tipo.

Esercizio 2 Definire una classe template per pile.

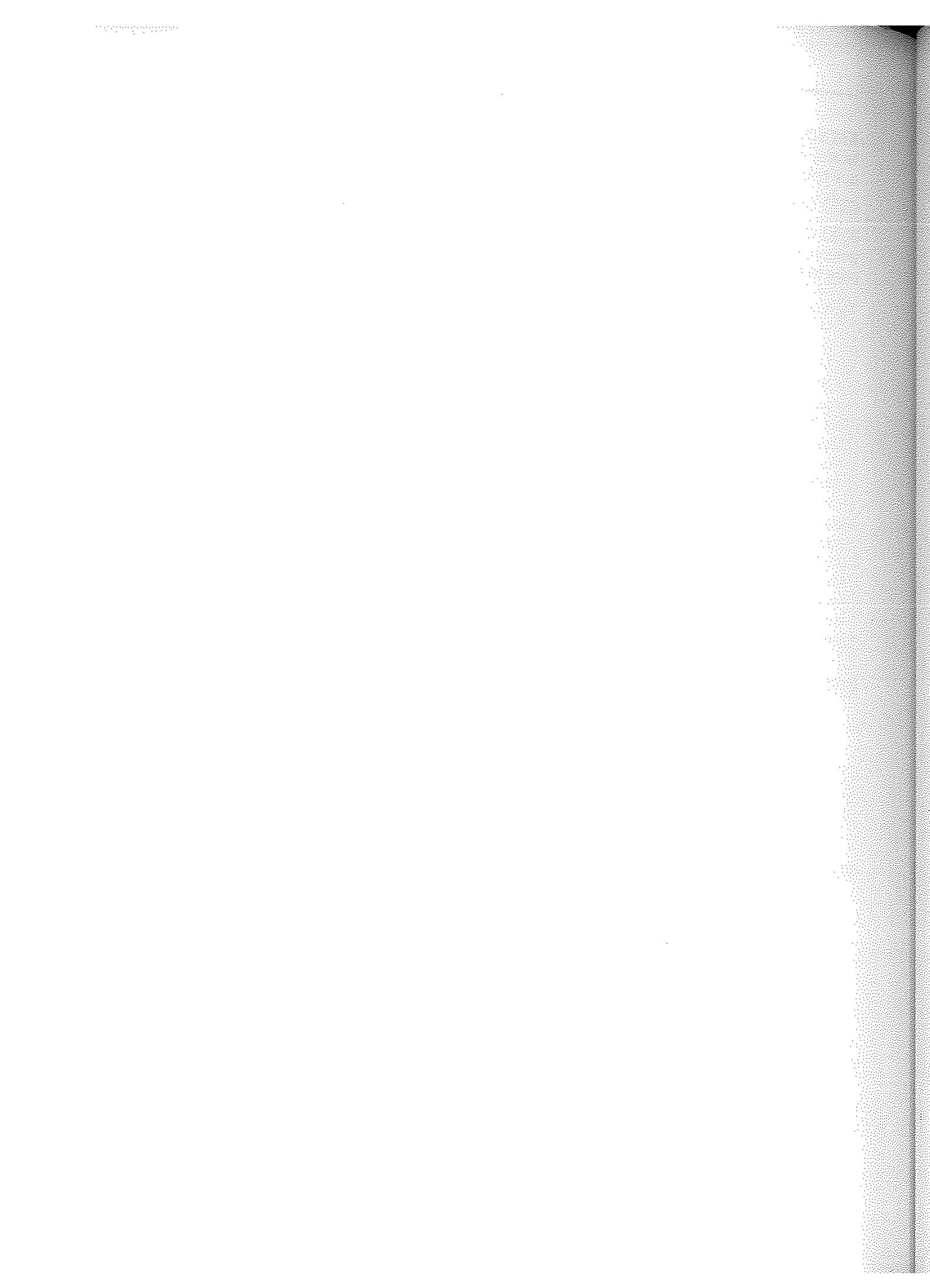
Esercizio 3 Come si implementano funzioni generiche in C++? Le confronti con i template di funzione.

Esercizio 4 Dichiarare un template di funzione `gsort` per ordinare array.

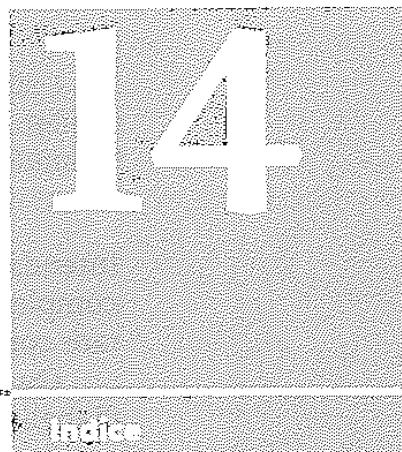
Esercizio 5 Definire un template di funzione che restituisca il valore assoluto di qualunque tipo di dato passatole.

Esercizio 6 Definire una funzione template `max()` che lavori con i tipi di dato predefiniti.

Soluzioni degli esercizi sul sito web [www.mheducation.it](http://www.mheducation.it)



# Sovraccaricamento degli operatori



Capitolo 14

- |             |  |             |   |
|-------------|--|-------------|---|
| <b>14.1</b> | Sovraccaricamento  | <b>14.7</b> | Sovraccaricamento                                     |
| <b>14.2</b> | Operatori unari  |             | di <code>new</code> e <code>delete</code>             |
| <b>14.3</b> | Sovraccaricamento degli operatori unari                        | <b>14.8</b> | Conversioni di dati e operatori di conversione        |
| <b>14.4</b> | Sovraccaricamento degli operatori binari                       |             | forzata di tipi                                       |
| <b>14.5</b> | Sovraccaricamento degli operatori di assegnamento              | <b>14.9</b> | Un'applicazione del sovraccaricamento degli operatori |
| <b>14.6</b> | Sovraccaricamento degli operatori di inserimento ed estrazione |             |   |

## Introduzione

Il "sovraffabbricamento" degli operatori, anche detto "sovraposizione" oppure "ridefinizione" degli operatori, è una delle caratteristiche più affascinanti dell'OOP e del C++ in particolare. Esso consente di manipolare gli oggetti definiti dal programmatore con operatori standard

del linguaggio come `+`, `*`, `[ ]` e `<<`. Di fatto questo significa poter definire una versione "personalizzata" del linguaggio. In questo capitolo vedremo il sovraccaricamento degli operatori, incluso quelli di *casting*, quelli relazionali e l'operatore di assegnamento.

## 14.1 Sovraccaricamento

Sappiamo bene che il C++ ha molti operatori predefiniti per operare su vari tipi di dato. Per esempio, l'operatore `+` può servire per sommare due variabili di tipo `int`, e l'operatore `<=` può confrontare due valori di tipo `double`. Questi operatori predefiniti sono già "sovraffabbricati" dal momento che ciascuno può operare con vari tipi di dato. Per esempio, l'operatore `+` può sommare due `double`, implementando un'operazione diversa da quella di sommare due `double`. Ma quando i tipi di dato sono "classi" *non ci sono operatori predefiniti*. Se abbiamo definito una classe `numero_razionale` con le sue possibili operazioni aritmetiche, dati `r1` e `r2` due oggetti di quella classe, si vorrebbe che l'operatore `+` possa servire anche per indicare il metodo `r1+r2` e che il compilatore

**Tabella 14.1** Operatori che non si possono sovraccaricare

	Accesso al membro
.	Indirezione di accesso al membro
::	Risoluzione di ambito
?:	Operatore condizionale

intenda correttamente l'operazione da effettuare. Così facendo, operandi complessi definiti con le classi (stringa, data, complesso ecc.) si potrebbero trattare come fossero operandi di tipi predefiniti.

Quasi tutti gli operatori predefiniti si possono sovraccaricare in C++ (Tabella 14.1).

Per sovraccaricare un operatore si usa la parola riservata `operator` seguita dall'operatore da sovraccaricare:

`operator operatore`

A parte il caso degli operatori `new`, `delete` e `->`, le funzioni operatore possono restituire qualunque tipo di dato. Non si possono cambiare precedenza, raggruppamento e numero degli operandi. Con l'eccezione dell'operatore `()`, le funzioni operatore non possono avere argomenti per default.

Una funzione operatore può essere o un metodo di una classe oppure una funzione che abbia almeno un parametro il cui tipo è una classe o un riferimento a una classe. Questa regola evita di cambiare il significato degli operatori che operano su tipi di dato predefiniti. Per esempio, il seguente codice non è valido:

```
int operator +(int x, int y)
{
    return x * y;
}
```

Al contrario, la seguente funzione è valida perché un parametro è di tipo classe:

```
class intero {
// ...
public:
    int valore;
};
int operator +(const intero &x, int y)
{
    return x.valore + y;
}
```

### RICORDA

- Il numero degli operandi e la priorità dell'operatore sovraccaricato devono essere le stesse dell'operatore predefinito.
- Il nuovo operatore può essere definito come una funzione membro o come una funzione amica.

## 14.2 Operatori unari

Gli *operatori unari che si possono sovraccaricare* sono indicati in Tabella 14.2.

L'esempio che segue sovraccarica l'operatore "meno unario" per definire vettori complementari, e poi lo utilizza per implementare il "meno binario"  $a - b$  a partire dal "più binario"  $a + (-b)$ .

### Esempio 14.1

```
class vettore {
public:
    vettore(float xx = 0, float yy = 0) : x(xx), y(yy) {}
    void stampa_vettore() const;
    vettore operator +(const vettore &b) const; // più binario
    vettore operator -(); // meno unario
    vettore operator -(vettore &b) const; // meno binario
private:
    float x, y;
};

void vettore::stampa_vettore() const {
    cout << x << ' ' << y << endl;
}

vettore vettore::operator +(const vettore &b) const {
    return vettore(x + b.x, y + b.y);
}
```

Tabella 14.2 Operatori unari che si possono sovraccaricare

Operatore	Significato
new	Nuova variabile dinamica
delete	Libera memoria variabile dinamica
new[]	Nuovo vettore dinamico
delete[]	Libera memoria vettore dinamico
++	Incremento
--	Decremento
()	Chiamata a funzione
[]	Indicizzazione
+	Più
-	Meno
*	Indirezione
&	Indirizzo di
!	NOT logico
~	NOT bit a bit
,	Virgola

```

vettore vettore::operator -() {
    return vettore(-x,-y);
}

vettore vettore::operator -(vettore &b) const {
    return *this + -b;
}

int main()
{
    vettore u(3,1), v(1,2), somma, neg, differ;
    somma = u + v ;
    somma.stampa_vettore();
    neg = -somma;
    neg.stampa_vettore();
    differ = u-v ;
    differ.stampa_vettore();
    return 0;
}

```

### Esecuzione

```

4 3
-4 -3
2 -1

```

### 14.3 Sovraccaricamento degli operatori unari

Consideriamo una classe t e un oggetto x di t. Definiamo un operatore unario sovraccaricato ++, e interpretiamo:

```

++x
come:
operator++(x)
oppure come:
x.operator++()

```

Un operatore unario si può definire come una funzione amica (vedremo poi) con un argomento oppure, più frequentemente, come un metodo senza argomento, come nell'esempio precedente.

Per sovraccaricare l'operatore ++, per esempio, si utilizza la dichiarazione  
**void operator ++**

Per esempio, se volessimo incrementare il timer di un oggetto della classe tempo potremmo definirne un metodo:

```

void incTempo // aggiunge 1 secondo al tempo misurato
{
    seg++ ;
}

```

```

if(secs > 59) {secs -= 60 ; ++mins;}
if(mins > 59) {mins -= 60 ; ++hrs;}
}

```

e utilizzarlo semplicemente così:

```

tempo t;
t.incTempo(); // incrementa t di un secondo

```

Nel seguente esempio utilizziamo lo stesso metodo, invocando però l'operatore `++` al posto della chiamata a `incTempo`.

#### Esempio 14.2

```

class timer
{
    private:
        int hrs;      // ore
        int mins;     // minuti
        int secs;     // secondi
    public:
        timer() {secs = 0; mins = 0; hrs = 0;} // inizializza a 00:00:00
        void visualizza()                      // visualizza il timer
        void leggetimer ()                     // legge il timer
            {cin >> hrs >> mins >> secs;}
        timer& operator++ ()                  // aggiunge un secondo
        {
            secs++;
            if (secs > 59) {secs -= 60; ++mins;}
            if (mins > 59) {mins -= 60; ++hrs;}
            return * this; // restituisce l'oggetto stesso
        };
    };
    int main()
    {
        timer t; // dichiara la variable Tempo
        cout << "\nLettura del timer (hh :mm :ss) : ";
        t.leggetimer(); // legge il timer
        cout << "il timer segna: ";
        t.visualizza();
        ++t; // incrementa il timer di 1 secondo
        cout << "\ndopo l'incremento, t = "; // visualizza di nuovo timer
        t.visualizza();
    }
}

```

#### Esecuzione di prova

Lettura del timer (hh :mm :ss) : 21 59 59

il timer segna: 21:59:59

dopo l'incremento, t = 22:0:0

Vediamo un altro esempio con una classe vettore e un operatore sovraccaricato `++` che incrementa del vettore (1,1).

#### Esempio 14.3

```
class vettore
{
    double x,y;
public:
    void inizializza(double a, double b)
    {x = a; y = b;}
    void visualizza() {
        cout << "vettore x = (" << x << ", " << y << ")\n";
    }
    vettore& operator++() {
        x++; y++;
        return *this;
    }
};

int main() {
    vettore v;
    v.inizializza(10.5, 20.3);
    v.visualizza();
    ++v;           // incrementa il vettore di 1,1
    v.visualizza();
}
```

#### Esecuzione

```
vettore x = (10.5, 20.3)
vettore x = (11.5, 21.3)
```

#### 14.3.1 Funzioni amiche

A volte si vuole che l'operatore agisca come funzione globale, e non come metodo di una classe. Però, come funzione globale non potrebbe accedere ai membri privati delle classi che gli fungono da operandi. In questi casi, mediante la parola riservata `friend`, è possibile dichiarare l'operatore come **funzione amica** della classe dei suoi operandi.

*Una funzione si dice "amica" di una classe, se è definita in un ambito diverso da quello della classe, ma può accedere ai suoi membri privati. Per ottenere ciò, bisogna inserire il prototipo della funzione nella definizione della classe (non importa se nella sezione privata o pubblica), facendo precedere lo specificatore friend. Quando una funzione è dichiarata amica di una certa classe è in grado di accedere ai suoi membri non pubblici.*

Vediamo adesso due esempi per meglio distinguere il sovraccaricamento di un operatore unario prefisso come funzione membro dal sovraccaricamento mediante definizione dello stesso come funzione amica. Iniziamo dal primo caso:

**Esempio 14.4**

```
class Unario {
    int x;
public:
    Unario() {x = 0;}
    Unario(int a) {x = a;}
    Unario& operator--() {--x; return *this;}
    void visualizza() {cout << x << ' ';}
};

int main()
{
    Unario esempio = 65;
    for(int i = 5; i > 0; i--)
    {
        (--esempio).visualizza();
    }
}
```

**Esecuzione**

64 63 62 61 60

Poiché la funzione operatore unario `--()` è dichiarata come funzione membro, il sistema passa il puntatore `this` implicitamente. Di conseguenza l'oggetto che viene passato alla funzione membro si converte nell'operando di questo operatore.

Vediamo ora come definire un operatore come funzione amica:

**Esempio 14.5**

```
class Unario {
    int x;
public:
    Unario() {x = 0;}
    Unario(int a) {x = a;}
    friend Unario& operator++(Unario&);
    void visualizza() {cout << x << ' ';}
};

Unario& operator++(Unario& X) {++X.x; return X;}

int main()
{
    Unario esempio = 65;
```

```

for(int i = 8; i > 0; i--)
{
    (++esempio).visualizza();
}
}

```

**Esecuzione**

64 63 62 61 60

In questo caso, poiché la funzione operatore `++()` è definita come funzione amica, non le si passa il puntatore `this` implicitamente. Di conseguenza si deve passare esplicitamente l'oggetto della classe `Unario`.

**Esempio 14.6**

```

class Contatore
{
private:
    unsigned int cont;
public:
    Contatore() {cont = 0;}
    int legge_contatore() {return cont;}
    Contatore& operator ++() {++cont; return *this;}
};

int main()
{
    Contatore C1, C2;
    cout << "\nC1 = " << C1.legge_contatore();
    cout << "\nC2 = " << C2.legge_contatore();
    ++C1; // Incrementa C1
    ++C2; // Incrementa C2
    ++C2; // Incrementa C2
    cout << "\nC1 = " << C1.legge_contatore();
    cout << "\nC2 = " << C2.legge_contatore();
}

```

**Esecuzione**

C1 = 0  
C2 = 0  
C1 = 1  
C2 = 2

Il prossimo esempio riepiloga sull'utilizzo degli operatori `++` e `--` sovraccaricati e con notazioni prefissa e postfissa.

**Esempio 14.7**

```
class Punto {
```

```
public :
    float x, y, z;
    Punto(float a, float b, float c){x=a; y=b; z=c;}
    Punto& operator ++();      // prefissa
    Punto operator ++(int);   // postfissa
    Punto& operator --();
    Punto operator --(int);
};

// operatore incremento prefisso (++p)
inline Punto& Punto::operator ++()
{
    x += 1.0;
    y += 1.0;
    z += 1.0;
    return *this;
}
// operatore incremento postfisso (p++)
inline Punto Punto::operator ++ (int)
{
    x += 1.0;
    y += 1.0;
    z += 1.0;
    return Punto (x + 1.0, y + 1.0, z + 1.0);
}

// operatore decremento prefisso (--p)
inline Punto& Punto::operator --()
{
    x -= 1.0;
    y -= 1.0;
    z -= 1.0;
    return *this;
}
// operatore decremento postfisso (p--)
inline Punto Punto::operator --(int)
{
    x -= 1.0;
    y -= 1.0;
    z -= 1.0;
    return Punto (x + 1.0, y + 1.0, z + 1.0);
}

int main()
{
    Punto p(5.0, 7.0, 8.0);
    Punto q(1.0, 2.0, 3.0);
    cout << " p( " << p.x << " , " << p.y << " , " << p.z << " )\n";
}
```

```

    ++p;
    cout << "++p( " << p.x << " , " << p.y << " , " << p.z << " )\n";
    cout << " q( " << q.x << " , " << q.y << " , " << q.z << " )\n";
    Punto r=q++;
    cout << " r( " << r.x << " , " << r.y << " , " << r.z << " )\n";
    cout << "q++( " << q.x << " , " << q.y << " , " << q.z << " )\n";
}

```

### Esecuzione

```

p( 5 , 7 , 8 )
++p( 6 , 8 , 9 )
q( 1 , 2 , 3 )
r( 1 , 2 , 3 )
q++( 2 , 3 , 4 )

```

## 14.4 Sovraccaricamento degli operatori binari

Gli operatori binari hanno due argomenti, uno a destra e uno a sinistra dell'operatore. La Tabella 14.3 elenca gli operatori binari che si possono sovraccaricare.

Un operatore binario può essere sovraccaricato:

- come una funzione globale di due argomenti (i due operandi);
- come un metodo di un solo argomento (il secondo operando).

**Tabella 14.3** Operatori binari che si possono sovraccaricare

Operatore	Significato	Operatore	Significato
+	Addizione	<b>!=</b>	OR bit a bit con assegnamento
-	Differenza	<b>=</b>	Uguale
*	Moltiplicazione	<b>!=</b>	Diverso
/	Divisione	<b>&gt;</b>	Maggiore di
%	Modulo	<b>&lt;</b>	Minore di
=	Assegnamento	<b>&gt;=</b>	Maggiore o uguale a
<b>+=</b>	Addizione con assegnamento	<b>&lt;=</b>	Minore o uguale a
<b>-=</b>	Differenza con assegnamento	<b>  </b>	OR logico
<b>*=</b>	Moltiplicazione con assegnamento	<b>&amp;</b>	AND logico
<b>/=</b>	Divisione con assegnamento	<b>&lt;&lt;</b>	Scorrimento a sinistra
<b>%=</b>	Modulo con assegnamento	<b>&lt;&lt;=</b>	Scorrimento a sinistra con assegnamento
<b>&amp;</b>	AND bit a bit	<b>&gt;&gt;</b>	Scorrimento a destra
<b> </b>	OR bit a bit	<b>&gt;&gt;=</b>	Scorrimento a destra con assegnamento
<b>^</b>	OR esclusivo bit a bit	<b>-&gt;</b>	Puntatore
<b>^=</b>	OR esclusivo bit a bit con assegnamento	<b>-&gt;*</b>	Puntatore a membro
<b>&amp;=</b>	AND bit a bit con assegnamento		

#### 14.4.1 Sovraccaricamento di un operatore binario come metodo di una classe

In questo caso il primo operando è l'oggetto della classe. L'esempio mostra come sovraccaricare un operatore binario come metodo.

##### Esempio 14.8

```
class Binario {
    float x;
public:
    Binario() {x = 0;}
    Binario(float a) {x = a;}
    Binario operator + (const Binario&) const;
    void visualizza() {cout << x << '\n';}
};

Binario Binario::operator+(const Binario& a) const
{
    Binario aux;
    aux.x = x + a.x;
    return aux;
}

int main()
{
    Binario primo(2.8), secondo(4.6), terzo;
    terzo = primo + secondo;
    terzo.visualizza();
}
```

##### Esecuzione

7.3

Come ulteriore esempio, definiamo un tipo di classe vettore con:

- due membri dato privati (coordinate di tipo double);
- due funzioni public:
  - una funzione membro di inizializzazione;
  - una funzione membro di visualizzazione delle coordinate;
- un operatore sovraccaricato \* prodotto scalare dei due vettori.

##### Esempio 14.9

```
class vettore
{
    double x, y;
public:
    vettore(double a, double b) {x = a; y = b;}
```

```

    void visualizza() { cout << "x=" << x << " y=" << y << endl; }
    double operator * (const vettore v) const { return (x * v.x + y * v.y); }
};

int main()
{
    vettore v1(1,2), v2(2,3);
    v1.visualizza();
    v2.visualizza();
    cout << "prodotto scalare = ";
    cout << v1*v2;
}

```

### Esecuzione

```

x=1 y=2
x=2 y=3
prodotto scalare = 8

```

Nel precedente esempio, l'operatore di moltiplicazione \* è sovraccaricato nella classe vettore. La scrittura v1\*v2 ha ora un aspetto naturale e rappresenta il prodotto scalare di due vettori. In effetti, operator\* è definito come una funzione membro che ha un argomento di tipo vettore e restituisce un'espressione di tipo double, ma la scrittura v1\*v2 è migliore della v1.operator\*(v2). Anche se l'operatore \* è sovraccaricato per il prodotto scalare, può essere sempre utilizzato nella sua accezione normale scrivendo:

i = j \* k

dove gli oggetti i, j, k sono aritmetici (int, long, double...), perché l'accezione corretta sarà individuata in funzione del tipo di operatore.

Come sappiamo, in C non si può utilizzare l'operatore + per concatenare stringhe. Per esempio, in BASIC si può scrivere:

stringa3 = stringa1 + stringa2;

dove stringa1, stringa2 e stringa3 sono variabili di tipo stringa, e stringa3 diviene il concatenamento di stringa2 e stringa3, ma in C no!

Tuttavia, definendo una classe stringa, come mostrato nel seguente programma, si può sovraccaricare l'operatore + per ottenere questo concatenamento.

### Esempio 14.10

```

#define MAXLUNGSTRINGA 80
class stringa
{
    char str[MAXLUNGSTRINGA];
    int lunghezza ;
public:

```

```

stringa() {};
stringa(const char s[]) {strcpy (str, s);}
stringa operator + (const stringa& x) const {
    stringa temp;
    strcpy(temp.str,str);
    strcat(temp.str,x.str);
    return temp;
};
void visualizza() {cout << str;}
};

int main()
{
    stringa a("Aldo ");
    stringa b("Franco ");
    stringa c("Dragon");
    stringa d=a+b+c;
    d.visualizza();
}

```

**Esecuzione**

Aldo Franco Dragoni

**14.4.2 Sovraccarico di un operatore binario con la funzione amica**

Modifichiamo l'Esempio 14.7 e definiamo l'operatore + mediante una funzione amica:

**Esempio 14.11**

```

class Binario {
    float x;
public:
    Binario()      {x = 0;}
    Binario(float a) { x = a;}
    friend Binario operator+ (const Binario&, const Binario&);
    void visualizza() {cout << x << '\n';}
};

Binario operator+(const Binario& a, const Binario& b)
{
    Binario aux;
    aux.x = a.x + b.x;
    return aux;
}

int main()
{
    Binario primo(2.8), secondo(4.5), terzo;

```

```

    terzo = primo + secondo;
    terzo.visualizza();
}

```

## Esecuzione

7.3

Definendo operatore binario + come funzione amica non le si passa implicitamente il puntatore this e le si deve passare l'oggetto Binario esplicitamente con entrambi gli argomenti. Di conseguenza, il primo argomento della funzione membro si converte nell'operando sinistro e il secondo argomento diventa l'argomento destro.

## 14.5 Sovraccaricamento degli operatori di assegnamento

In C++ si può sovraccaricare l'operatore di assegnamento = perché lavori in maniera specifica su oggetti di classi definite dal programmatore. Senza sovraccaricarlo, l'operatore di assegnamento su due oggetti della stessa classe:

```
nuovoOggetto = vecchioOggetto;
```

copierà membro a membro tutti i dati membri da vecchioOggetto a nuovoOggetto mediante l'operatore di *assegnamento di copia*. I meccanismi sono essenzialmente gli stessi dell'inizializzazione per default membro a membro, ma fanno uso di un operatore di assegnamento di copia implicito al posto del costruttore di copia. Il formato generale dell'operatore di assegnamento di copia è il seguente:

```

const nomeClasse& nomeClasse::operator=(const nomeClasse &v)
{
    if (this!= &v) // evita autoassegnamenti
    {
        // semantica di copia della classe
    }
    return *this; // restituisce l'oggetto assegnato
}

```

dove la condizione

```
if (this!= &v)
```

evita l'assegnamento di un oggetto di una classe a sé stessa. Ma questo assegnamento di copia si può sovraccaricare.

### Esempio 14.12

```

class Punto
{
public:

```

```

double x, y, z;
Punto(): x(0.0), y(0.0), z(0.0) {}
Punto (double x_arg, double y_arg, double z_arg = 0.0)
    : x(x_arg), y(y_arg), z(z_arg) {}
const Punto& operator = (const Punto& p);
};

// operatore di assegnamento sovraccaricato
const Punto& Punto::operator = (const Punto& p) {
    x = p.x*1000;
    y = p.y*1000;
    z = p.z*1000;
    cout << "ho utilizzato l'operatore = sovraccaricato " << endl;
    return *this;
}

int main() {
    Punto p1 (2, 4,6), p2, p3, p4;
    cout << "p1( " << p1.x << " , " << p1.y << " , " << p1.z << " )\n";
    cout << "p2( " << p2.x << " , " << p2.y << " , " << p2.z << " )\n";
    p4 = p3 = p2 = p1;           // 3 assegnamenti sovraccaricati
    cout << "p2( " << p2.x << " , " << p2.y << " , " << p2.z << " )\n";
    cout << "p3( " << p2.x << " , " << p2.y << " , " << p2.z << " )\n";
    cout << "p4( " << p2.x << " , " << p2.y << " , " << p2.z << " )\n";
}

```

## Esecuzione

```

p2( 0 , 0 , 0 )
ho utilizzato l'operatore = sovraccaricato
ho utilizzato l'operatore = sovraccaricato
ho utilizzato l'operatore = sovraccaricato
p2( 2000 , 4000 , 6000 )
p3( 2000 , 4000 , 6000 )
p4( 2000 , 4000 , 6000 )

```

Il seguente elenco mostra la classe stringa con un operatore di assegnamento sovraccaricato. In questo caso, l'operatore ridefinito permette l'assegnamento di una stringa tradizionale C (char\*) direttamente a un oggetto di una definita classe stringa.

## Esempio 14.13

```

class stringa {
    char * v;
    int lun;
public:
    stringa() {lun = 0; new char[10];}
    stringa(char *);

```

```

~stringa() {delete v;}
stringa operator + (stringa);
stringa operator + (char*);
const stringa& operator = (const char* );
char * Mostrare() {return v;}
};

stringa::stringa(char* str) {
    lun = strlen(str) + 1;
    v = new char[lun];
    strcpy(v,str);
}

const stringa& stringa::operator = (const char* cad) {
    lun = strlen(cad) + 1;
    delete v ;
    v = new char [lun];
    strcpy (v,cad);
    return *this;
}

int main() {
    stringa u("Questo è solo un test");
    u = " Questo non è un test";
    cout << "u = " << u.Mostrare() << endl;
}

```

### Esecuzione

u = Questo non è un test

#### 14.5.1 Sovraccarico degli operatori [] e ()

Si può anche ridefinire l'operatore binario di indice vettoriale [] per classi che rappresentino un'astrazione di contenitore multiplo di elementi dello stesso tipo in cui sia possibile accedere agli elementi individuali. Nell'uso visto a suo tempo l'utilità di questo operatore sta nel fatto che esso occulta l'aritmetica dei puntatori, ovvero se si definisce l'array:

int numeri[20];

la compilazione dell'istruzione di assegnamento:

ch = numeri[12];

produce lo stesso codice dell'istruzione:

ch = \*numeri + 12;

L'operatore [] si può sovraccaricare solo mediante funzioni membro e richiede ovviamente due operandi; infatti nell'espressione generica:

p = x[i];

[] è l'operatore, x è il primo argomento e i è il secondo; in generale, l'oggetto indicizzato è il primo argomento, e l'indice il secondo. In C++ la funzione operatore corrispondente è `operator[]` e può essere definita per una classe x in maniera tale che:

`x[i]` equivale a `x.operator[](i)`

Anche l'operatore di chiamata di funzione () può essere sovraccaricato solo mediante funzioni membri. La chiamata di funzione si considera come un operatore binario.

*espressione principale (lista di espressioni)*

dove *espressione principale* e *lista di espressioni* (che può essere vuota) sono due operandi. La funzione operatore corrispondente è `operator()` e può essere definita per una classe t solo mediante una funzione membro non statica. Così facendo le seguenti espressioni sarebbero equivalenti:

<code>x(i)</code>	equivale a	<code>x.operator()(i)</code>
<code>x(arg1, arg2)</code>	equivale a	<code>x.operator()(arg1, arg2)</code>

## 14.6 Sovraccaricamento degli operatori di inserimento ed estrazione

I flussi di I/O non sono parte del linguaggio C++ ma sono implementati come classi nella libreria `<iostream>`. In C++ si possono sovraccaricare gli operatori di inserimento ed estrazione in modo da consentir loro di manipolare qualunque istruzione di flusso che includa qualunque classe. Come definito in `<iostream>`, questi operatori lavorano con i tipi di dato predefiniti, come `int`, `long`, `double` e `char*`.

Sovraccaricando gli operatori di I/O si possono gestire specificamente non solo i tipi predefiniti (`int`, `long`, `double`, `char*` ecc.) ma anche oggetti di qualsiasi classe.

Quando si definiscono classi come la stringa dell'esempio precedente, si possono sovraccaricare le definizioni degli operatori `<<` e `>>` per farli lavorare anche con esse. Per esempio, una volta sovraccaricato l'operatore `>>`, si può leggere i caratteri di un flusso di input immagazzinandoli in un oggetto della classe stringa scrivendo:

```
stringa input_utente;
cin >> input_utente;
```

Alla stessa maniera, per visualizzare una stringa stringa si scriverà:

```
stringa saluti = "Salve a tutti!";
cout << saluti << endl;
```

A causa della pervasività dell'uso delle stringhe, il C++ ha superato le rigidezze del concetto di stringa ereditato dal C (Capitolo 9) definendo una

classe specifica **string** nella libreria `<cstring>`, ma che adesso è parte della libreria standard C++; quindi, senza includere alcuna libreria oltre `<iostream>` è possibile compilare ed eseguire il seguente programma.

#### Esempio 14.14

```
int main() {
    string u("Facciamo un test sulla classe predefinita string");
    cout << "u = " << u << endl;
    u = "Questo è il risultato di un assegnamento ad oggetto della classe string";
    cout << "u = " << u << endl;
    cout << "Per favore immetta il suo nome completo:\n";
    cin >> u;
    cout << "Questo è il nome completo letto con >>:\n" << u << endl;
    getline(cin,u);
    cout << "Questo è quanto letto con la funzione non membro getline():\n"
        << u << endl;
}
```

#### Esecuzione

```
u = Facciamo un test sulla classe predefinita string
u = Questo è il risultato di un assegnamento ad oggetto della classe string
Per favore immetta il suo nome completo:
Aldo Franco Dragoni
Questo è il nome completo letto con >>;
Aldo
Questo è quanto letto con la funzione non membro getline():
Franco Dragoni
```

Il sovraccaricamento dell'operatore di inserimento `<<` consente di visualizzare nel flusso di output qualunque classe definita dal programmatore. Il seguente esempio dichiara una classe punto che contiene due coordinate di tipo intero, `x` e `y`. Aggiungendo alla classe punto un operatore `<<` sovraccaricato, si potranno inserire in output direttamente oggetti di tipo punto.

#### Esempio 14.15

```
#include <iostream>
class punto{
private:
    int x, y;
public:
    punto() {x = y = 0;}
    punto (int xx, int yy) {x = xx; y = yy;}
    void ascissa(int xx) {x = xx;}
    void ordinata(int yy) {y = yy;}
    friend ostream& operator << (ostream& os, const punto p);
};
```

```

| ostream& operator<< (ostream& os, punto p) {
|   os << "x = " << p.x << ", y = " << p.y;
|   return os;
|
| }

int main() {
  punto p;
  cout << p << '\n';
  p.ascissa(50);
  p.ordinata(100);
  cout << p << '\n';
}

```

### Esecuzione

```

x = 0, y = 0
x = 50, y = 100

```

L'ultima linea della classe punto sovraccarica l'operatore di inserimento in output dichiarandolo con una funzione membro amica `operator<<()`. Questa funzione amica restituisce l'indirizzo di un oggetto della classe `ostream` (definita nella libreria `<ostream>`) e dichiara due parametri: un riferimento `os` al tipo `ostream` e un oggetto `p` della classe `punto`. Come abbiamo visto, le funzioni operatore sovraccaricate sono normalmente amiche della classe perché altrimenti non potrebbero accedere ai membri dato privati, a meno che non ne siano funzioni membro. Il sovraccaricamento di `>>` è simile a quello di `<<`. Il seguente programma migliora il precedente aggiungendo una funzione operatore di estrazione dall'input.

### Esempio 14.16

```

#include <iostream>
#include <iostream>
class punto {
private:
  int x, y;
public:
  punto() {x = 0; y = 0;}
  punto (int xx, int yy) {x = xx; y = yy;}
  void assegnaX(int xx) {x = xx;}
  void assegnaY(int yy) {y = yy;}
  int leggeX() {return x;}
  int leggeY() {return y;}
  friend ostream& operator<< (ostream& os, const punto &p);
  friend istream& operator>> (istream& is, punto &p);
};

int main() {
  punto p;

```

```

cout << p << '\n';
p.assegnaX(50);
p.assegnaY(100);
cout << p << '\n';
cout << "Introdurre i valori di x e y = ";
cin >> p;
cout << "\nHa introdotto: " << p;
return 0;
}

ostream& operator<< (ostream& os, const punto &p) {
    os << "x = " << p.x << ", y = " << p.y;
    return os;
}

istream& operator>>(istream& is, punto &p) {
    is >> p.x >> p.y;
    return is;
}

```

### Esecuzione

```

x = 0, y = 0
x = 50, y = 100
Introdurre i valori di x e y = 1234 4321
Ha introdotto: x = 1234, y = 4321

```

Le ultime due linee della classe punto sovraccaricano gli operatori di inserimento in output ed estrazione dall'input dichiarandoli come funzioni amiche che dichiarano due parametri entrambi riferimenti a una classe.

Un altro esempio di sovraccaricamento dell'operatore di input può essere il seguente.

### Esempio 14.17

```

#include <iostream>
class Data{
    int giorno;
    int mese;
    int anno;
public:
    friend istream & operator >> (istream & en, Data &f);
};

istream & operator>>(istream& en, Data& f)
{
    cout << endl;
    cout << "Che giorno è? ";
    en >> f.mese;
}

```

```

cout << "del mese numero? ";
en >> f.giorno;
cout << "dell'anno? ";
en >> f.anno;
return en;
}

int main () {
    Data oggi;
    cin >> oggi;
}

```

### Esecuzione

Che giorno è? 31  
 del mese numero? 1  
 dell'anno? 2021

## 14.7 Sovraccaricamento di new e delete

è possibile sovraccaricare gli operatori new e delete per controllare in maniera più efficiente la gestione dinamica della memoria.

Il compilatore C++ considera automaticamente l'operatore new come static perché viene chiamato spesso per creare istanze di classe. Esso però può essere sovraccaricato inserendo un prototipo del tipo:

```
void* operator new(size_t dimensione, ...);
```

dove il primo paramentro è del tipo standard size\_t, e l'operatore deve restituire un puntatore void\*. L'argomento per il primo parametro è il nome di una classe o un tipo di dato. Le successive chiamate di new per assegnare memoria a oggetti di classe attiveranno le funzioni sovraccaricate. Anche l'operatore sovraccaricato restituirà l'indirizzo dello spazio di memoria assegnato a quell'oggetto. Se non c'è spazio disponibile, la funzione restituirà 0.

Consideriamo, per esempio, la seguente dichiarazione di classe:

```

class Z
{
public:
    Z() {}
    static void* operator new (size_t dim);
};

int main()
{
    Z* z = new Z;
    // altre istruzioni
    delete z;
    return 0;
}

```

Il nome della classe `Z` che appare dopo l'operatore `new` è l'argomento per dimensionare il parametro. Il compilatore traduce il nome della classe nella sua dimensione e passa l'informazione al parametro `dim`.

C++ permette di dichiarare parametri addizionali per l'operatore `new`. Siccome il compilatore considera l'operatore `new` come static, esso non può accedere a membri dati non statici. Il seguente esempio contiene due versioni dell'operatore sovraccaricato `new`.

#### Esempio 14.18

```
class Matrice
{
protected:
    unsigned Mrighe;
    unsigned Mcolonne;
    double* Ptrdati;
public:
    Matrice(unsigned Rigue, unsigned Colonne) {
        Mrighe = Rigue;
        Mcolonne = Colonne;
        Ptrdati = new double [Mrighe * Mcolonne];
    }
    static void* operator new(size_t dimensione);
    static void* operator new(size_t, unsigned Rigue, unsigned Colonne);
};

int main() {
    Matrice *mat = new Matrice(15,25);
    // altre istruzioni
    delete mat;
    return 0;
}
```

In questo programma ci sono due versioni dell'operatore `new` sovraccaricato. Il costruttore di classe invoca la prima versione dell'operatore `new` per creare i dati dinamici della classe `Matrice`. La chiamata di `new` nel `main` invoca la seconda versione sovraccaricata dell'operatore per creare un'istanza dinamica di `Matrice`. La creazione dell'istanza attiva il costruttore, che invoca la prima versione dell'operatore `new` per assegnare dinamicamente spazio addizionale a cui si accede attraverso il puntatore `Ptrdati`. Per invocare l'operatore `new` globale a questo punto si deve utilizzare l'operatore di risoluzione di visibilità `::new`.

Si può sovraccaricare anche l'operatore `delete`, ma solo una versione per classe. La sintassi generale per dichiarare un operatore `delete` sovraccaricato è:

```
void operator delete(void *p, size_t dimensione)
```

Quando si utilizza questo secondo formato, il compilatore passerà in `dimensione` il numero di byte da liberare.

## 14.8 Conversioni di dati e operatori di conversione forzata di tipi

Quando in C++ si assegna un valore di un tipo standard a una variabile di un altro tipo standard compatibile, C++ converte automaticamente il valore al tipo della variabile ricevente. Per esempio, le seguenti istruzioni eseguono conversioni di tipo numerico:

```
long conteggio = 8;           // il valore 8 viene convertito al tipo long
double ora = 15;             // il valore 15 viene convertito al tipo double
int lato = 4.44;              // il double 4.44 viene convertito al tipo int 4
```

C++ esegue automaticamente queste conversioni che in alcuni casi possono anche far perdere precisione, come nell'ultimo esempio in cui, oltre alla conversione, avviene un troncamento.

Il C++ non converte automaticamente tipi non compatibili. Per esempio, l'istruzione

```
int* p = 50;
```

produce un errore perché a sinistra c'è un *puntatore a intero* mentre il valore a destra è di tipo *intero*, e puntatori e interi sono tipi diversi (non si può, per esempio, elevare al quadrato un puntatore).

Sappiamo già che quando non è possibile effettuare la conversione automatica si può utilizzare la conversione forzata, cioè il *casting*.

```
int* p = (int*) 50;          // p e (int*) 50 sono puntatori
```

Il compilatore funzionerà in maniera diversa a seconda che la conversione sia fra tipi base, fra tipi base e oggetti oppure fra oggetti di classi differenti. Quando si scrive un'istruzione come:

```
var_int = var_float;
```

si suppone che il compilatore converta, mediante un'opportuna funzione, il valore di var\_float, in virgola mobile, al formato int da assegnare a var\_int. Naturalmente esistono molte conversioni: da char a float, da float a double ecc, ciascuna con la propria funzione di conversione *implicita*. A volte la conversione la si vuole forzare in maniera esplicita con il *casting*. Per esempio, la conversione precedente verrebbe forzata scrivendo:

```
var_int = int(var_float); // esempio di casting stile C
```

Il *casting* è una *conversione forzata* ma utilizza la stessa funzione di conversione implicita.

Per convertire tipi definiti dal programmatore in tipi predefiniti non si possono utilizzare le funzioni conversione implicita ma bisogna definire delle specifiche funzioni di conversione, cioè degli *operatori di conversione*. L'argomento di questi operatori è l'argomento implicito *this*. Per convertire a un tipo *nometipo* si definisce un operatore:

```
operator nometipo();
```

tenendo presente che la funzione di conversione:

- deve essere un metodo della classe in cui convertire;
- non deve specificare un tipo di ritorno;
- non deve avere argomenti;
- il suo nome deve essere il tipo in cui convertire l'oggetto.

Per esempio, una funzione per convertire a double avrà questo prototipo:

```
operator double();
```

Per esempio, consideriamo due classi: **Cartesiane**, che rappresenta la posizione di un punto in un sistema cartesiano, e **Polari**, che rappresenta la posizione dello stesso punto in un sistema di coordinate polari (Raggio e Angolo) (Figura 14.1).

L'assegnamento:

**Cartesiane** = **Polari**;

dovrà implicare una conversione di tipo, quindi dobbiamo anche definire uno specifico operatore di conversione. Ricordiamo dalla geometria elementare che:

$$x = \text{Raggio} * \cos(\text{Angolo})$$

$$y = \text{Raggio} * \sin(\text{Angolo})$$

#### Esempio 14.10

```
#include <cmath>
class Cartesiane
{
private:
    double x;
    double y;
public:
    Cartesiane() {x = 0.0; y = 0.0;}
```

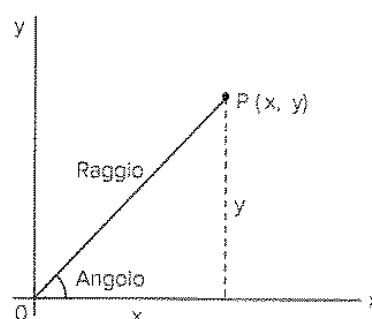


Figura 14.1

Coordinate ortogonali e polari di un punto.

```

Cartesiane(double ascissa, double ordinata) {x = ascissa; y = ordinata;}
void visualizza() {cout << " (" << x << ", " << y << " )";}
};

class Polari
{
private:
    double r;
    double a;
public:
    Polari() {r = 0.0; a = 0.0;}
    Polari(double raggio, double angolo) {r = raggio; a = angolo ;}
    void visualizza() { cout << " (" << r << ", " << a << " )";}
    operator Cartesiane()      // funzione di conversione coordinate
    {
        double x = r * cos(a);
        double y = r * sin(a);
        return Cartesiane(x, y); // invoca il costruttore di Cartesiane
    }
};

int main()
{
    Cartesiane cart;
    Polari pol1(100.0,0.01);
    cart = pol1;
    cout << "\nCoordinate polari\tCoordinate cartesiane\n";
    pol1.visualizza(); cout << '\t'; cart.visualizza(); cout << endl;
    Polari pol2(100.0,3.14);
    cart = pol2;
    pol2.visualizza(); cout << '\t'; cart.visualizza(); cout << endl;
    return 0;
}

```

### Esecuzione

Coordinate polari	Coordinate cartesiane
( 100 , 0.01 )	( 99.995 , 0.999983 )
( 100 , 3.14 )	( -99.9999 , 0.159265 )

La funzione operator Cartesiane() transforma l'oggetto di cui è membro nell'oggetto Cartesiane e restituisce questo oggetto, che main() assegna a Cart.

### 14.9 Un'applicazione del sovraccaricamento degli operatori

Consideriamo un tipo complesso per rappresentare i numeri complessi e le loro operazioni (addizione, sottrazione, moltiplicazione, divisione, uguaglianza) mediante sovraccaricamento degli operatori +, -, \*, / e ==.

**ESEMPIO**

Dati due numeri complessi

$$z_1 = a + bi \quad z_2 = c + di$$

le operazioni tipiche sono:

$$z_1 + z_2 = (a + c) + i(b + d)$$

$$z_1 - z_2 = (a - c) + i(b - d)$$

$$z_1 * z_2 = (ac - bd) + i(bc + ad)$$

$$z_1 / z_2 = \frac{(ac + bd)}{c^2 + d^2} + i \frac{(bc - ad)}{c^2 + d^2}$$

La dichiarazione della classe complesso va nel file complesso.h, mentre la sua definizione va nel file complesso.cpp.

**Esempio 14.20**

compleSSO.h

```
class complesso {
    float r, i;
public:
    complesso (float a = 0.0, float b = 0.0);
    float real();
    float imag();
    complesso operator + (const complesso& a);
    complesso operator - (const complesso& a);
    complesso operator * (const complesso& a);
    complesso operator / (const complesso& a);
    bool operator == (compleSSO a);
};
```

compleSSO.cpp

```
#include "compleSSO.h"
compleSSO::compleSSO(float a, float b){ r = a; i = b; }
float compleSSO::real() { return r; }
float compleSSO::imag() { return i; }
compleSSO compleSSO::operator +(const compleSSO& a) {
    compleSSO b(r + a.r, i + a.i);
    return b;
}
compleSSO compleSSO::operator -(const compleSSO& a) {
    compleSSO b(r - a.r, i - a.i);
    return b;
}
compleSSO compleSSO::operator *(const compleSSO& a) {
    compleSSO b(r * a.r - i*a.i, r * a.i + i * a.r);
    return b;
}
```

```

complesso complesso::operator /(const complesso& a) {
    complesso b;
    float denom = a.r * a.r + a.i * a.i;
    b.r = (r * a.r + i * a.i)/denom;
    b.i = (i * a.r - r * a.i)/denom;
    return b;
}
bool complesso::operator == (complesso a) {
    return (r == a.r) && (i == a.i);
}

```

Una volta definita la classe complesso, si desidera un programma che simuli una calcolatrice aritmetica per effettuare le operazioni aritmetiche sui numeri complessi letti da tastiera; il programma è nel file calcolatrice\_complexi.cpp.

#### calcolatrice\_complexi.cpp

```

// calcolatrice per l'aritmetica dei numeri complessi
// Il programma termina battendo control-c
#include "complesso.cpp"
int main()
{
    float ar, ai, br, bi;
    char opr;
    complesso r;
    for(;;) {
        cout << "Introduca il primo numero complesso (Reale Immaginario): ";
        cin >> ar; cin >> ai;
        cout << "introduca l'operatore (+, -, *, /): ";
        cin >> opr;
        cout << "introduca il secondo numero complesso (Reale Immaginario): ";
        cin >> br; cin >> bi;
        switch (opr) {
            case '+':
                r = complesso(ar, ai) + complesso(br, bi);
                break;
            case '-':
                r = complesso(ar, ai) - complesso(br, bi);
                break;
            case '*':
                r = complesso(ar, ai) * complesso(br, bi);
                break;
            case '/':
                r = complesso(ar, ai) / complesso(br, bi);
                break;
            default:
                cerr << "operatore non valido" << endl;
        }
    }
}

```

```

    }
    cout << "Il risultato è: ";
    cout << r.real() << ' ' << r.imag() << endl;
}
}

```

### Esecuzione di prova

Introduca il primo numero complesso (Reale Immaginario): 15 28

introduca l'operatore (+, -, \*, /): /

introduca il secondo numero complesso (Reale Immaginario): 4 4

Il risultato è: 6.375 1.625

Introduca il primo numero complesso (Reale Immaginario): ^C

### Spiegazione

Il sovraccaricamento degli operatori è una delle operazioni fondamentali nella OOP. Esso consente di associare nuove operatività a operatori già definiti. Per esempio, il significato dell'espressione  $a + b$  può essere fatto dipendere dai tipi delle variabili  $a$  e  $b$ ; oltre alla predefinita somma di interi o reali, l'operazione potrebbe acquisire il significato di somma di numeri complessi o addirittura di concatenamento di stringhe di caratteri.

Cerchiamo allora di riassumere per punti le sue caratteristiche:

1. il sovraccaricamento degli operatori fa uso della parola riservata `operator` seguita dall'operatore che si vuole sovraccaricare;
2. la Tabella 14.1 raccoglie gli operatori predefiniti che non si possono sovraccaricare;
3. quelli che non si possono sovraccaricare sono: `*`, `/`, `:`, `:`, `?`, `#`, `##`
4. gli operatori si possono sovraccaricare come funzioni ordinarie, amiche (`friend`) o membro;
5. non si possono modificare priorità, associatività e arità degli operatori predefiniti;
6. un operatore definito come funzione membro ha un parametro in meno di quanti ne avrebbe se fosse stato definito come funzione ordinaria o amica;

7. un operatore definito come funzione membro si può invocare sia utilizzando la notazione di funzione membro sia quella prefissa, infissa o suffissa. Per esempio, data la classe `T` che sovraccarica l'operatore binario `+`,

```

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

supponendo che  $a$  e  $b$  sono variabili di tipo `T`, l'operatore `T::operator+` si può invocare come:

`a.operator+(b)`

oppure come:

`a + b`

quest'ultimo è esattamente il modo in cui l'operatore `+` sarebbe stato invocato se fosse stato definito come funzione amica:

```

class T {
    // ...
public:
    friend T operator+(T,T);
    // ...
};
```

o come funzione ordinaria:

`T operator+(T,T)`

L'ambiguità si può evitare definendo un operatore per ciascuno di questi tre metodi;

8. le versioni sovraccaricate dei seguenti operatori:

= () [] →

devono essere definite come funzioni membro e mantengono le seguenti equivalenze:

$p \rightarrow x$  equivale a  $*(\mathbf{p}+1)$

$(\mathbf{*p}).x$  equivale a  $\mathbf{p}[1]$

9. le relazioni fra gli operatori sovraccaricati non si stabiliscono automaticamente; per esempio, se  $++$  è sovraccaricato nel tipo complesso per incrementare la parte reale di 1,0,  $a++$  non sarà automaticamente equivalente a:  $a+=1.0$  perché questa rela-

zione esiste a priori solo per gli operatori predefiniti.

10. gli operatori sovraccaricati non possono avere argomenti per default;

11. la relazione fra un operatore e la corrispondente chiamata di funzione si può riassumere così:

Operatore	Chiamata della funzione	
	... membro	... non membro
A <op> B	A.operator <op> B	operator <op> (A, B)
<op> B	B.operator <op>()	operator <op> (B)
A<op>	A.operator <op>(0)	operator <op>(A, 0)

### RISOLUZIONI ESERCIZI

- Conversione di tipi di dato
- Funzione operatore
- casting**
- Operatore binario

- Operatore unario
- Sovraccaricamento di new e delete
- operator

## Esercizi

**E1** Scrivere una classe Razionale per i numeri razionali, cioè numeri che possono essere rappresentati come coefficienti di due interi (1/2, 3/4, 4/2 ecc). Sovraccaricare i seguenti operatori in modo da farli applicare al tipo Razionale: ==, <, <=, >, >=, +, -, \*, /.

**E2** Data la classe Complesso introdotta in questo capitolo, sovraccaricare i seguenti operatori di modo in modo da farli applicare al tipo Complesso: =, +, -, \*, >> e <<. Si ricorda a riguardo che il prodotto di due numeri complessi è dato dalla seguente formula:

$$(a+b*i)*(c+d*i) == (a*c - b*d) + (a*d + b*c)*i$$

**E3** Scrivere una classe Ora\_giorno che consente di scrivere l'ora del giorno (in ore e minuti) su 24 ore. La classe deve sovraccaricare gli operatori ++ e -- per incrementare ore e minuti.

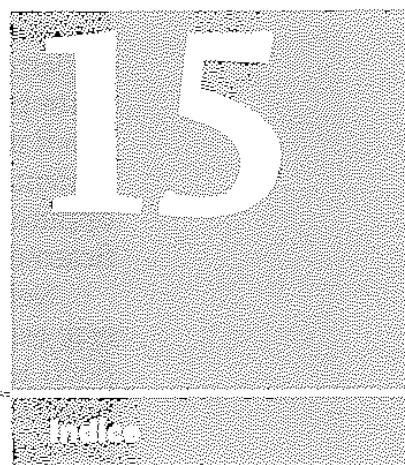
**E4** Data la classe Punto introdotta in questo capitolo, sovrascrivere l'operatore - per calcolare la distanza di due punti.

**E5** Sovraccaricare gli operatori aritmetici per applicarli a vettori o matrici.

**E6** Sovraccaricare tre operatori aritmetici per concatenare, separare e confrontare stringhe.

**Soluzioni degli esercizi sul sito web [www.mheducation.it](http://www.mheducation.it)**

# Eccezioni



- |             |                                    |             |                                    |
|-------------|------------------------------------|-------------|------------------------------------|
| <b>15.1</b> | Condizioni di errore nei programmi | <b>15.3</b> | Specifiche delle eccezioni         |
| <b>15.2</b> | Gestione delle eccezioni in C++    | <b>15.4</b> | Esempi di gestione delle eccezioni |

## Introduzione

Uno dei problemi più importanti nello sviluppo del software è la gestione delle condizioni d'errore. Indipendentemente da quanto sia buono e testato il software, errori possono sempre comparire per svariate ragioni (errori nascosti di programmazione, errori imprevisti del sistema operativo, indisponibilità delle risorse ecc.). Le *eccezioni* sono per l'appunto condizioni d'errore impreviste. Queste condizioni d'errore dovrebbero attivare delle proce-

dure di segnalamento dell'errore intercorso. Esempi tipici sono l'esaurimento del segmento di memoria, divisioni per zero ecc. C++ mette a disposizione un sistema di trattamento delle eccezioni. L'idea è quella di interrompere l'esecuzione del programma al presentarsi dell'eccezione, consentendo così l'attivazione di procedure definite dal programmatore per segnalare il problema e tentare di recuperare le condizioni di normale funzionamento.

## 15.1 Condizioni di errore nei programmi

La programmazione di classi e funzioni in sistemi software complessi è compito difficile e delicato, per questo è necessario gestire gli errori nel miglior modo possibile. La maggior parte dei programmatore si pone due problemi nella gestione degli errori: quali sono i problemi che si possono produrre in caso di errore e come gestirli. Poiché gli errori sono quasi inevitabili, bisogna pensare a un meccanismo che incorpori la gestione degli errori nella struttura stessa del codice sorgente.

Per migliorare la portabilità delle sue librerie, C++ include un *meccanismo di gestione delle eccezioni*. In questo capitolo lo esamineremo per capire come esso consenta di rilevare gli errori, gestirli, gestire le risorse e specificare le eccezioni.

Prevedere gli errori è sempre stato un problema per i progettisti di software. In parte perché non si capisce facilmente dove essi potrebbero prodursi, in parte perché poi non si capisce bene come gestirli. Infatti, una volta prodotto il malfunzionamento, il programma ha varie opzioni: terminare immediatamente, ignorare l'errore con la speranza che non succeda nulla di disastroso oppure attivare un segnale d'errore che dovrà, presumibilmente, essere verificato da altre istruzioni del programma.

Nella pratica i programmatore non si prendono queste brighe perché esse esigono molto lavoro e perché il codice aggiuntivo appesantisce e rende poco leggibile il software. È davvero difficile capire come prevedere ogni possibile condizione d'errore ogni volta che si chiama una certa funzione, e poi chiamare i distruttori per liberare la memoria e chiudere i file aperti prima di mettere in pausa il programma se qualcosa è andato storto.

La soluzione a questi problemi in C++ è chiamare il meccanismo del linguaggio che supporta la gestione degli errori. Quando si produce l'errore, C++ attiva automaticamente un blocco di codice, chiamato "gestore delle eccezioni", per rispondere in maniera appropriata all'errore che l'ha attivato. Questo comportamento generale si chiama *catching an exception*. Se il gestore delle eccezioni non c'è il programma viene semplicemente abortito.

Le condizioni d'errore vengono normalmente codificate per poter essere individuate e riconosciute in modo automatico. Per esempio, i sistemi operativi e le librerie di sistema documentano tutti i possibili codici d'errore, specialmente quelli legati a condizioni di terminazione di funzioni che possono lasciare connessioni aperte, file aperti, dati che rimangono sulla RAM senza essere salvati ecc. Se questi errori vengono gestiti immediatamente dopo che vengono prodotti allora forse possono essere trattati adeguatamente; ma se essi si ripercuotono sui processi successivi, allora possono produrre malfunzionamenti in questi ultimi che, a loro volta, attiveranno i propri meccanismi di gestione delle eccezioni, ma le azioni che verranno intraprese e le informazioni che verranno fornite sui guasti intercorsi potrebbero a quel punto non essere più corrette o significative.

## 15.2 Gestione delle eccezioni in C++

L'attivazione di un'eccezione indica una condizione anormale di funzionamento che non dovrebbe verificarsi in una normale esecuzione del programma. A essa deve essere associata un'urgente azione riparatrice. Se l'eccezione viene sollevata mentre è attivo il processo di gestione delle eccezioni, allora esso prende il controllo della situazione (cioè il flusso del programma), altrimenti probabilmente il programma si bloccherà e terminerà in maniera anomala.

Purtroppo l'inclusione di un tale meccanismo non è gratuita; si deve aggiungere nuovo codice che tende a rendere oscuro il codice originale e ne aumenta, ovviamente, la dimensione.

In C++, un'eccezione è un oggetto (di un tipo fondamentale del linguaggio oppure definito dal programmatore), che viene passato dal codice che ha prodotto il problema a quello che dovrebbe gestirlo.

Il meccanismo si serve di tre nuove parole riservate `try`, `throw` e `catch`:

- `try`, un blocco per rilevare le eccezioni;
- `catch`, un manipolatore per catturare le eccezioni dei blocchi `try`;
- `throw`, un'espressione per sollevare (`raise`) eccezioni.

I passi sono:

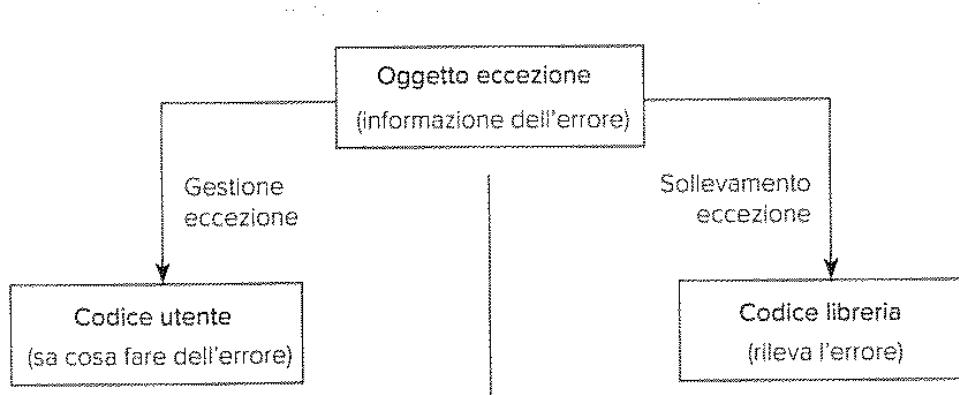
1. il programmatore prova (`try`) un'operazione per anticipare gli errori;
2. quando la procedura anticipa un errore lancia (`throw`) un'eccezione;
3. qualcuno interessato a quella condizione di errore (per eliminarlo o bypassarlo) anticiperà l'errore e catturerà (`catch`) l'eccezione sollevata.

Il meccanismo si completa con:

- una funzione `terminate()` che intrappoli le eccezioni non catturate;
- specifiche che discriminino quali eccezioni, se esistono, possono lanciare una funzione;
- una funzione `unexpected()` che intrappoli le violazioni delle specifiche delle eccezioni.

La filosofia del sistema di gestione delle eccezioni è semplice. Il codice che gestisce un problema non è lo stesso che lo rileva. Questo tipo di separazione costituisce un *firewall* fra le applicazioni e le librerie (Figura 15.1).

Quando un programma utente chiama in maniera non corretta una funzione oppure un oggetto di una classe, la libreria della classe crea un oggetto eccezione che contiene informazioni circa ciò che è successo. Essa solleva quindi un'eccezione, cosa che rende l'oggetto eccezione disponibile al codice utente attraverso il gestore delle eccezioni. Il codice utente che manipola l'eccezione può decidere cosa farne. Questo modo di gestire il problema offre diversi vantaggi. I programmi diventano più leggibili dato che il codice che gestisce l'errore è indipendente da quello che lo rileva. Esso uniforma il metodo di gestione e lo rende indipendente dalla particolare libreria coinvolta; il codice utente deve solo dettare le regole circa cosa fare dell'eccezione.



**Figura 15.1**

La gestione delle eccezioni costituisce un frangifuoco.

Il frangifuoco che agisce da ponte fra applicazione e librerie deve fare varie cose per gestire adeguatamente il flusso delle eccezioni, riservando e liberando memoria in modo dinamico.

Una delle ragioni più importanti per utilizzare le eccezioni è che le applicazioni non possono ignorarle. Quando il meccanismo solleva un'eccezione qualcuno deve raccoglierla, altrimenti essa diviene "non catturata" (*uncaught*) e il programma termina per omissione. Questo concetto potente è il cuore e la forza del meccanismo perché forza le applicazioni a gestire le eccezioni invece che ignorarle.

Questo meccanismo segue un *modello di terminazione*. Vale a dire che non si torna mai al punto in cui l'eccezione è stata sollevata. La gestione delle eccezioni non funziona come la gestione degli *interrupt*, che fa partire una routine di servizio dell'interruzione per poi ritornare a dove il programma è stato sospeso. Questa tecnica consuma tempo, è incline a loop infiniti ed è più complicata da implementare. Progettato per gestire eccezioni sincrone (cioè prodotte dal programma e non derivanti dall'esterno) con una sola linea di controllo, il meccanismo delle eccezioni implementa invece un cammino alternativo a una sola via per differenti punti del programma.

```
void f()
{
    // codice che produce il sollevamento di un'eccezione
    // ...
    throw i;
    // ...
}
int main()
{
    try {
        f(); // chiamata a f, preparata per qualunque errore
        // qui va il codice normale
    }
    catch(...)
    {
        // catturare qualunque eccezione sollevata per f()
        // fare altro
    }
    // codice normale del main()
}
```

f() è una semplice procedura. Quando s'incontra una condizione d'errore, si lancia un'eccezione mediante l'istruzione:

```
throw i;
```

L'operando dell'espressione `throw` è un oggetto. Normalmente gli oggetti lanciati contengono informazioni specifiche circa l'errore intercorso.

Nel `main()`, la chiamata a `f()` è inserita in un blocco `try`. Esso indica al compilatore l'eventualità di un'eccezione.

Il blocco `catch()` cattura un'eccezione del tipo indicato. Nel precedente esempio:

```
catch(...)
```

indica che cattura eccezioni di tutti i tipi, perché i punti di sospensione significano proprio "qualsiasi tipo di argomento". Il `catch` è come una procedura a un solo argomento.

Quindi il C++ può sollevare un'eccezione nel blocco `try` mediante l'espressione `throw`. L'eccezione sarà gestita invocando un apposito gestore da una lista che si trova alla fine del blocco `try`.

#### Esempio 15.1

```
int main () {
    try
    [
        throw 20;
    ]
    catch (int e)
    [
        cout << "Eccezione Nr. " << e << '\n';
    ]
    return 0;
}
```

#### Esecuzione

```
Eccezione Nr. 20
```

##### 15.2.1 Progetto delle eccezioni

La parola riservata `try` designa un blocco di istruzioni che rileva le eccezioni. All'interno dei blocchi `try`, normalmente si chiamano funzioni che possono sollevare eccezioni. La parola riservata `catch` designa un gestore delle catture con un argomento che rappresenta un tipo di eccezione. Questi gestori delle catture seguono immediatamente blocchi `try` oppure altri `catch` con argomenti diversi.

Per catturare eccezioni bisogna avere un blocco `try` e almeno un `catch`, altrimenti l'eccezione non viene catturata e il programma termina per omissione. I blocchi `try` sono importanti perché i gestori delle catture a loro associati determinano qual'è la parte del programma eccezione. Cosa fare con l'eccezione lanciata è scritto nel codice del blocco `catch`.

##### 15.2.2 Blocchi `try`

Un blocco `try` deve contenere le istruzioni che possono lanciare eccezioni e comincia con la parola riservata `try`. Dopo il blocco `try` c'è una sequenza di uno o più gestori `catch`. Quando un tipo di eccezione lanciata coincide con

l'argomento di un catch, il flusso di programma entra dentro il blocco di quel gestore catch. Se nessuna eccezione è lanciata dal blocco try, il controllo salta al gestore catch e prosegue all'istruzione successiva.

La sintassi del blocco try è:

```
try { codice del blocco try }
      ...
catch (argomenti_1) { codice del blocco catch }
      ...
catch (argomenti_n) { codice del blocco catch }
      ...
}
```

I blocchi try possono anche annidarsi:

```
void sub(int n)
{
    try [
        ...
        try [
            ...
            if (n==1) return ;
        ]
        catch (...) {...} // gestore catch interno
    ]
    catch (...) {...} // gestore catch esterno
    ...
}
```

Un'eccezione lanciata nel blocco try interno manda in esecuzione il gestore catch interno, se il suo argomento coincide con il tipo d'eccezione. Il gestore catch esterno gestisce le eccezioni lanciate dal blocco try esterno, sempre se il tipo d'eccezione coincide con il suo argomento, ma anche le eccezioni lanciate dal blocco try interno se il tipo d'eccezione coincide con il suo argomento (e non con quello del catch interno). Se il tipo dell'eccezione non coincide con l'argomento di alcun blocco catch, l'eccezione si propaga al chiamante di sub().

1. Quando si produce un'eccezione dopo il try si salta al primo gestore catch il cui parametro coincide con il tipo d'eccezione.
2. Dopo che il gestore catch ha terminato l'esecuzione delle sue istruzioni termina il blocco try e l'esecuzione prosegue con l'istruzione successiva. Non si ritorna mai al punto in cui s'è verificata l'eccezione.
3. Se non ci sono gestori per trattare l'eccezione, si abortisce il blocco try e si attiva l'eccezione.

**Esempio 15.2**

La funzione media() calcola la media di vettori di tipo double mediante la funzione avg() che lancia un'eccezione se la lunghezza del vettore è maggiore di 6 in maniera da non incorrere in errore di buffer overflow.

```
double avg(double n[], int l) {
    if (l > 6) throw 1;
    double somma = 0.0;
    for (int i = 0; i < l; ++i) somma += n[i];
    return somma / l;
}

int main () {
    double b[] = {1.2, 2.2, 3.3, 4.4, 5.5, 6.6};
    try { avg(b, 10); }
    catch (int i) { cout << "la media è " << avg(b, 6); }
return 0;
}
```

**Esecuzione**

la media è 3.86667

**15.2.3 Lancio di eccezioni**

L'istruzione throw lancia un'eccezione. Il formato di throw è:

1. `throw espressione`
2. `throw`

Il blocco più interno del try in cui l'eccezione è sollevata si utilizza per selezionare l'istruzione catch che la processerà. L'istruzione throw senza argomento si può utilizzare dentro un catch per rilanciare l'eccezione corrente. Normalmente si utilizza quando si desidera chiamare un ulteriore gestore catch oltre quello corrente per processare ulteriormente l'eccezione.

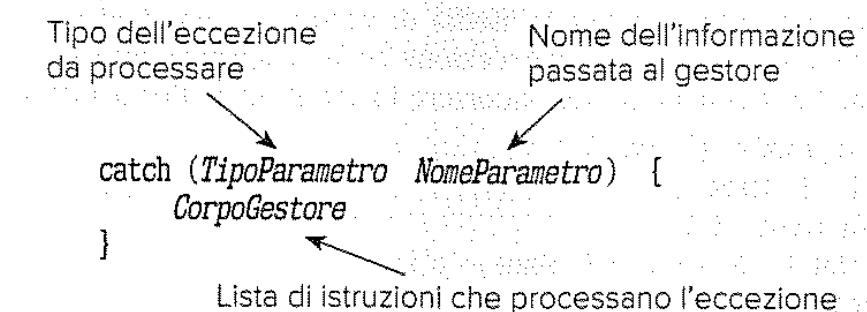
**15.2.4 Cattura di un'eccezione: catch**

Il gestore delle eccezioni in C++ consiste di tre parti: la parola riservata catch, la dichiarazione fra parentesi di un solo tipo od oggetto semplice, e l'istruzione composta che verrà eseguita se il gestore raccoglierà un'eccezione.

**Formati**

1. `catch (argomento) { // cattura un'eccezione corrispondente
 // all'argomento
 codice del blocco catch
}`
2. `catch (...) [ // cattura qualunque eccezione
 codice del blocco catch...]
}`

La prima forma del gestore catch è simile alla definizione di una funzione:



Al contrario del blocco try, il blocco catch si esegue solo in circostanze speciali. Il suo argomento può consistere di un tipo, un tipo seguito da un nome di argomento oppure una sintassi speciale:

```

catch (tipo)
catch (tipo nome_argomento)
catch (...)

```

Le prime due forme specificano un gestore di eccezioni che raccoglie solo un tipo d'eccezione coincidente. Così come le funzioni sovraccaricate, il gestore delle eccezioni si attiva solo se l'eccezione lanciata corrisponde con la dichiarazione del suo argomento.

La terza forma è una sintassi speciale che significa "qualunque eccezione". Si può utilizzare questa sintassi per scrivere gestori di eccezioni *catch-all* che raccolgono tutte le eccezioni lanciate e non ancora catturate.

Come già spiegato, quando un'istruzione all'interno del blocco try lancia un'eccezione, try prova ogni blocco catch nell'ordine in cui sono scritti fino a che ne incontra uno il cui parametro coincide con il tipo d'eccezione lanciata. Appena si incontra la coincidenza si eseguono le istruzioni del blocco catch corrispondente; dopodiché il blocco try termina e si prosegue con l'istruzione successiva.

Se non esistono gestori in grado di catturare l'eccezione, il blocco try viene abortito e l'eccezione non viene raccolta. Ovviamente, se non viene sollevata alcuna eccezione non si invocherà alcun gestore e tutte le istruzioni all'interno del blocco try saranno eseguite normalmente fino al suo termine.

Prendiamo, per esempio, la seguente funzione:

```

int f(int x)
{
    int a = fa(x);
    int b = fb(x);
    return a / b;
}

```

dove fa e fb sono funzioni arbitrarie. Se fb restituisce il valore 0, la funzione dovrebbe calcolare (e restituire) il risultato di una divisione per zero, cosa che provocherebbe un errore con terminazione del programma. Quindi bisogna intervenire sul codice per impedire che questo accada, sollevando un'ec-

cezione "divisione per zero" e prevedere azioni opportune per cercare di proseguire il programma. Una soluzione potrebbe essere:

```
int f(int x)
{
    int a = fa(x);
    int b = fb(x);
    if (b == 0)
        return INT_MAX;
    return a / b;
}
```

La costante `INT_MAX` è definita nel file header `limits.h`. Essa indica il maggior intero che si può contenere in una variabile di tipo `int`, che sarà il valore restituito nel caso `b` fosse uguale a 0; però questa soluzione non è corretta poiché il vero risultato di una divisione per zero è il valore "infinito" e non `INT_MAX`.

Una soluzione più corretta sarebbe:

```
if (b == 0)
{
    cout << "Overflow dentro la funzione f";
    exit(1);
}
```

però questa soluzione fa terminare il programma, cosa che potrebbe non essere davvero necessaria, perché magari la funzione `f` non è dopotutto così importante ecc.

La soluzione migliore è quella di consentire alla funzione `f` di generare un'eccezione qualora `b` dovesse diventare 0.

```
int f(int x)
{
    int a = fa(x);
    int b = fb(x);
    if(b == 0)
        throw errore_overflow ("Errore in f");
    return a/b;
}
```

Per esempio:

```
void g() {
    int i;
    while (i)
        try {
            cout << "?";
            if !(cin >> i) break;
            int r = f(i);
            cout << "Il risultato è " << r << endl;
        }
```

```

    }
    catch(errore_overflow)
        cout << "Il risultato è infinito." << endl;
    }
}

```

La funzione g legge l'intero dalla tastiera. L'immissione da tastiera termina quando si preme la combinazione dei caratteri che indica la fine del flusso (Ctrl-Z oppure Ctrl-D). Per ciascun numero letto si chiama la funzione f e si manda nel flusso di output il risultato di f. Se viene letto un numero che fa diventare b uguale a zero, si esegue la seguente istruzione:

```
throw errore_overflow ("Errore in f");
```

Poiché questa eccezione non è stata prodotta in un blocco try, non viene trattata nella funzione f. Ciò che accade è che f interrompe la sua esecuzione e l'eccezione viene inviata alla sua chiamante g. In g esiste ora un'eccezione prodotta nel punto in cui è stata chiamata f, cioè nell'istruzione:

```
int r = f(i);
```

questa istruzione viene abortita e, poiché risiede in un blocco try, il flusso del programma salta al primo gestore catch del blocco try il cui argomento coincide con il tipo errore\_overflow. In effetti ce n'è uno che stampa:

Il risultato è infinito.

Appena si incontra un gestore che cattura l'eccezione, questa viene eliminata. L'esecuzione prosegue quindi normalmente con la prima istruzione che viene dopo il blocco try.

#### Esempio 15.2

```

catch(char* messaggio)
{
    cerr << messaggio << endl;
    exit(1)
}
catch(...)
{
    cerr << " qui si raccoglie tutto " << endl;
    abort();
}

```

l'argomento con i punti sospensivi accoglie qualunque tipo di argomento.

### 15.3 Specifica delle eccezioni

Un interrogativo a questo punto è: Come si fa a conoscere i vari possibili tipi di eccezione? Ovviamamente per saperlo bisognerebbe leggere il codice, ma

questo non è sempre possibile con funzioni appartenenti a grandi programmi. Un altro metodo è leggere la documentazione disponibile sul software sperando che contenga informazioni sulle varie possibili eccezioni che esso può sollevare, ma purtroppo queste informazioni non sono sempre disponibili.

C++ offre una terza possibilità. Una dichiarazione di funzione può contenere una specifica circa quali eccezioni essa può generare. Questa *specifica delle eccezioni* garantisce che la funzione non lanci alcun altro tipo d'eccezione. I sistemi software ben progettati con gestione delle eccezioni devono definire quali funzioni lanciano eccezioni e quali no, e questo è implicito se si adotta la specifica delle eccezioni. Bisogna anche specificare cosa accade se una funzione lancia un'eccezione non prevista.

I formati per la specifica delle eccezioni sono:

```
tipo nomefunzione (argomento) throw (e1, e2, eN); // prototipo
tipo nomefunzione (argomento) throw (e1, e2, eN) // definizione
{
    corpo della funzione
}
e1, e2, eN
```

elenco dei nomi delle eccezioni che la funzione può lanciare, direttamente o indirettamente, incluse le eccezioni che possono essere derivate pubblicamente da questi nomi.

Se si lancia un'eccezione diversa da quelle indicate nell'elenco, il meccanismo chiama `unexpected()`. Si può inserire una specifica delle eccezioni in qualunque funzione C++, incluse le funzioni membro di classe e le funzioni template.

Se la lista è vuota la funzione non lancia eccezioni, come nel seguente esempio:

```
void g (argomenti) throw()
```

mentre se non viene specificata la lista delle eccezioni allora non ci sono vincoli e quindi si può lanciare qualunque eccezione, come nel seguente esempio:

```
void g (argomenti)
[...]
```

Riassumendo:

- una specifica di eccezione si può aggiungere alla fine della dichiarazione di una funzione

```
tipo ritorno f (argomenti) throw(T1, T2, T3,...)
```

da qui la funzione può generare solo eccezioni dei tipi specificati;

- una specifica di eccezione vuota significa che la funzione non genera alcuna eccezione

```
tipo ritorno f (argomenti) throw();
```

- senza specifica di eccezione la funzione può generare tutte eccezioni *tipo Ritorno f(argomenti);*
- se si genera un'eccezione di un tipo diverso da quelli specificati nella lista delle eccezioni viene chiamata la funzione `unexpected` la quale termina il programma.

#### Esempio 15.4

Il prototipo della seguente funzione `f()`, indica che `int` e `double` sono i tipi di eccezione che si potrebbe dover gestire se si invoca `f()`.

```
void f(char c) throw (int, double)
```

Se la funzione `f()` genera un'eccezione diversa dai tipi `int` o `double` verrà automaticamente invocata la funzione `unexpected`. Per esempio, supponiamo che `f()` sia:

```
void f(char c) throw (int, double) {
    if (isupper(c)) throw(1);
    else if (islower(c)) throw(1.0);
    else if (c == '.') throw(c);
}
```

Se la si invoca così:

```
f('a')
```

si genera un'eccezione di tipo `double`. Se la si invoca così:

```
f('A')
```

si genera un'eccezione di tipo `int`. Se la si invoca così:

```
f('.')
```

il programma termina. Questo perché viene lanciata un'eccezione di tipo `char`, che non è né di tipo `int` né di tipo `double`.

In questo esempio abbiamo visto che se l'eccezione non viene raccolta da uno specifico gestore, il meccanismo delle eccezioni chiama la funzione `unexpected()` (la quale chiama `terminate()` per terminare il programma).

#### 15.4 Esempi di gestione delle eccezioni

Per riepilogare vediamo alcuni esempi di gestione delle eccezioni in C++.

#### Esempio 15.5

Trattamento delle eccezioni in un programma per calcolare le radici di un'equazione di secondo grado.

```
#include <cmath>
enum errore {no_radici_reali, coefficiente_a_zero};
```

```

void radici (float a, float b, float c, float &r1, float &r2)
throw(errore)
{
    float discr;
    if(b*b < 4 * a * c) throw no_radici_reali;
    if(a==0) throw coefficiente_a_zero;
    discr = sqrt(b * b - 4 * a * c);
    r1 = (-b - discr) / (2 * a);
    r2 = (-b + discr) / (2 * a);
}
int main(int argc, char *argv[])
{
    float a, b, c, r1, r2;
    cout << "Introduca i coefficienti dell'equazione di 2° grado: ";
    cin >> a >> b >> c;
    try {
        radici (a, b, c, r1, r2);
        cout << "Radici reali " << r1 << " " << r2 << endl;
    }
    catch (errore e)
    {
        switch(e) {
        case no_radici_reali :
            cout << "Nessuna radice reale" << endl;
            break;
        case coefficiente_a_zero :
            cout << "Primo coefficiente zero" << endl;
        }
    }
    return 0;
}

```

## Esecuzione

```

$; Introduca i coefficienti dell'equazione di 2° grado: 0 2 4
Primo coefficiente zero
$; Introduca i coefficienti dell'equazione di 2° grado: 1 10 20
Radici reali -7.23607 -2.76393
$; Introduca i coefficienti dell'equazione di 2° grado: 10 1 20
Nessuna radice reale

```

Nell'esempio seguente applichiamo il meccanismo di gestione delle eccezioni a un programma che gestisce uno *stack*. Dichiariamo quindi una classe di interi *Pila* con un massimo di cinque elementi. Essa contiene due classi annidate: *Overflow* e *Underflow*, per gestire le condizioni d'errore:

- stack pieno: non si possono più inserire dati;
- stack vuoto: non si possono togliere dati.

Quando la gestione dello stack cade in uno di questi due casi viene lanciata la corrispondente eccezione.

#### Esempio 15.6

```
#define DIM_PILA 5
class Pila
{
public:
    class Overflow           // una classe eccezione
    {
        public:
            int val_overflow;
            Overflow (int i) : val_overflow(i) {}
    };
    class Underflow          // una classe eccezione
    {
        public:
            Underflow () {};
    };
    Pila () {cima = -1;}
    void mettere(int elem)
    {
        if (cima < (DIM_PILA-1)) lapila[++cima] = elem;
        else throw Overflow (elem);
    }
    int togliere()
    {
        if (cima > -1) return lapila[cima--];
        else throw Underflow ();
    }
private:
    int lapila[DIM_PILA];
    int cima;
};
int main()
{
    Pila miapila;
    int i = 5, j = 25;
    try           // Blocco try
    {
        miapila.mettere(i);
        miapila.mettere(j);
        miapila.mettere(1);
        miapila.mettere(12345);
        miapila.mettere(9999);
        // Pila piena con cinque numeri si forza un'eccezione
    }
}
```

```

    miapila.mettere(100); // lancia Pila::Overflow
}
// Gestori delle eccezioni
catch(Pila::Overflow& p)
{
    cout << "La pila ha debordato inserendo: "
    << p.val_overflow << endl;
}
catch(Pila::Underflow& p)
{
    cout << "La pila s'è svuotata." << endl;
}

```

## Esecuzione

La pila ha debordato inserendo: 100

Le eccezioni sono condizioni di errore impreviste. Normalmente queste condizioni abortiscono il programma con messaggi di errore forniti dal sistema operativo. In questo capitolo abbiamo visto che il C++ dispone di un meccanismo specifico per gestire le eccezioni (simile a quello del linguaggio Ada). Il

codice può sollevare un'eccezione mediante l'espressione `throw` all'interno di un blocco `try`. L'eccezione sarà raccolta da una clausola `catch` che conterrà le istruzioni per trattare opportunamente il caso senza far necessariamente terminare il programma.

- Cattura delle eccezioni
- `catch`
- Blocco `try`
- Specifiche delle eccezioni

- Eccezioni standard
- Lancio delle eccezioni
- `terminate()` e `unexpected`
- `throw`

## Esercizi

**Esercizio 1** Il seguente programma che implementa un algoritmo di ordinamento base non funziona bene. Collocare dichiarazioni nel codice del programma in modo da verificare se questo programma funziona correttamente. Scrivere il programma corretto.

```
void scambio(int x, int y)
{
    int aux = x;
    x = y;
    e = aux;
}

void ordinare (int v[], int n)
{
    int i, j;
    for (i=0; i<n; ++i)
        for (j=i; j<n; ++j)
            if (v[j] < v[j+1])
                scambio(v[j], v[j+1]);
}

int main()
{
    int z[12]={14,13,8,7,6,12,11,10,9,-5,1,5};
    ordinare (z, 12);
    for (int i=0; i<12; ++i)
        cout << z[i] << '\t';
    cout << "ordinato " << endl;
}
```

**Esercizio 2** Il seguente codice serve per definire e gestire eccezioni:

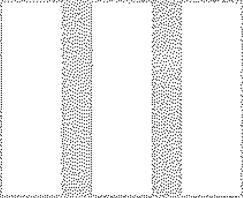
```
class stringa {
    char* s;
public:
    enum {minLong = 1, maxLong = 1000};
    stringa();
    stringa(int);
    ...
}
```

```
stringa::stringa(int lunghezza)
{
    if (lunghezza < minLong || lunghezza > maxLong)
        throw (lunghezza);
    s = new char [lunghezza];
    if (s == 0)
        throw ("Fuori memoria");
}
...
void f(int n)
{
    try{
        stringa cad (n);
    }
    catch (char* Msgerr)
    {
        cerr << Msgerr << endl;
        abort();
    }
    catch(int k)
    {
        cerr << "Fuori range: " << k << endl;
        if (stringa::maxLong);
    }
}
```

- Che succede se si eliminano le seguenti righe dalla funzione f e si invoca f con l'argomento 5.000?
- Che succede se si cambia il costruttore con il seguente codice e s'invoca f con l'argomento 5.000?

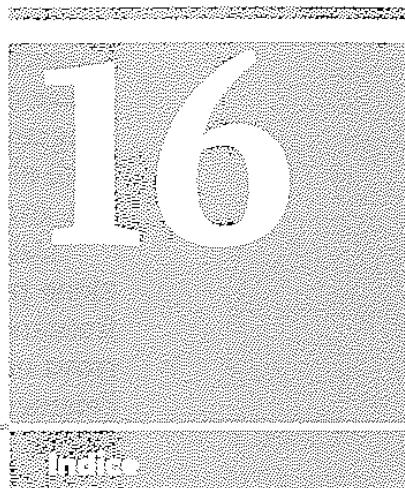
```
stringa::stringa (int n) throw (char*)
{
    ...
}
```

**PARTE**



# **Algoritmi e strutture dati astratte**

# Ordinamento e ricerca



- |             |   |             |                             |
|-------------|---|-------------|-----------------------------|
| <b>16.1</b> | Ricerca in vettori: ricerca sequenziale e binaria | <b>16.4</b> | Ordinamento per scambio     |
| <b>16.2</b> | Analisi degli algoritmi di ricerca                | <b>16.5</b> | Ordinamento per selezione   |
| <b>16.3</b> | Algoritmi di ordinamento elementari               | <b>16.6</b> | Ordinamento per inserimento |
|             |   | <b>16.7</b> | Ordinamento a bolle         |
|             |   | <b>16.8</b> | Ordinamento Shell           |

## Introduzione

L'*ordinamento* di elenchi, cataloghi e archivi contenuti in vettori è una delle funzionalità più richieste al software. Lo studio dei metodi di ordinamento è un compito molto interessante che ha impegnato le menti dei programmati dagli anni sessanta agli anni ottanta. Questo capitolo presenta alcuni degli algo-

ritmi più noti e utilizzati, anche in campo professionale, soffermandosi sulla loro implementazione utilizzando esclusivamente i concetti della programmazione strutturata, senza cioè utilizzare quelli della programmazione orientata agli oggetti che saranno invece oggetto della seconda parte del libro.

### 16.1 Ricerca in vettori: ricerca sequenziale e binaria

La ricerca di un elemento all'interno di un vettore, magari di grandi dimensioni, è uno dei compiti più richiesti al software, per questo è importante utilizzare tecniche di ricerca affidabili ed efficienti. In questa sezione si esamineranno due tecniche di ricerca: la *sequenziale* e la *binaria*.

#### 16.1.1 Ricerca sequenziale

La ricerca sequenziale cerca un elemento di un vettore utilizzando un valore **chiave**. In una ricerca sequenziale (a volte chiamata **ricerca lineare**), gli elementi di un vettore si attraversano in sequenza, uno dopo l'altro. La ricerca sequenziale è necessaria quando il vettore è ordinato in base al valore di un campo diverso da quello che contiene il valore cercato; per esempio, quando si cerca l'abbonato a cui corrisponde un certo numero di telefono, perché l'elenco è ordinato alfabeticamente per cognome e nome dell'abbonato e non per numero telefonico.

L'algoritmo di ricerca sequenziale confronta ogni elemento dell'array con la *chiave* di ricerca. Dato che l'array non è ordinato in base al campo chiave, il programma dovrà confrontare la chiave di ricerca mediamente con la metà degli elementi dell'array. Il metodo di ricerca lineare funzionerà quindi bene con array piccoli ed è necessario per vettori non ordinati.

L'algoritmo comincia nel primo elemento *vettore[0]* oppure in una posizione predeterminata (*inizio*) e attraversa i restanti elementi del vettore, confrontando ogni elemento con la chiave. L'attraversamento continua finché si trova un elemento con il valore chiave cercato o finché termina il vettore. Se l'elemento viene trovato se ne restituisce l'indice, altrimenti si restituisce il valore -1. L'algoritmo in C++ è il seguente:

```
int ricercaLineare (int vettore[], int dim, int chiave)
{
    for (int i = 0; i < dim; i++)
        if (vettore[i] == chiave) return i;
    return -1;
}
```

Se si desidera esaminare il vettore a partire da un elemento diverso dal primo:

```
int RicercaSequenziale (int v[], int dim, int n, int chiave)
{
    for (int i = n; i < dim; i++)
        if (v[i] == chiave) return i;
    return -1;
}
```

#### Esempio 16.1

Il programma applica la ricerca sequenziale contando il numero di occorrenze di una chiave in un vettore. Il programma principale introduce 10 interi in un array, poi chiede una chiave.

Il programma compie ripetute chiamate a *RicercaSeq* utilizzando un indice iniziale diverso, a partire dall'indice 0. A ogni chiamata di *RicercaSeq*, se si è trovata la chiave vengono incrementati sia il contatore delle occorrenze sia quello della posizione iniziale, in maniera tale che la successiva ricerca non riparta dall'inizio. Se non viene trovata la chiave il programma termina e visualizza il numero delle occorrenze trovate.

```
int RicercaSeq (int vettore[], int dim, int n, int chiave)
{
    for (int i = n; i < dim; i++)
        if (vettore[i] == chiave)
            return i;
    return -1;
}
int main ()
```

```

{
    int a[10];
    int pos, chiave, cont = 0;
    cout << "Introduca un vettore di 10 interi: ";
    for (pos = 0; pos < 10; pos++) cin >> a[pos];
    cout << "Introduca chiave da cercare: ";
    cin >> chiave;
    pos = 0;
    while ((pos = RicercaSeq(a, 10, pos, chiave)) != -1)
    {
        cont++;
        pos++;
    }
    cout << chiave << " si ripete " << cont
        << (cont != 1 ? "volte" : "volta") << " nel vettore." << endl;
}

```

### Esecuzione di prova

Introduca un vettore di 10 interi: 4 3 7 1 2 6 3 8 3 4

Introduca chiave da cercare: 3

3 si ripete 2 volte nel vettore.

#### 16.1.2 Ricerca binaria

La ricerca sequenziale si applica a qualsiasi vettore. Ma se il vettore è ordinato in base al campo che contiene il valore cercato allora la *ricerca binaria* è certamente migliore. Si inizia dal centro del vettore. Se l'elemento centrale non contiene la chiave cercata, allora si esaminerà l'elemento centrale del semivettore destro o di quello sinistro a seconda del fatto che la chiave dell'elemento esaminato era maggiore o minore di quella cercata. Il procedimento andrà avanti con semivettori sempre dimezzati fino a trovare, se c'è, l'elemento cercato. Per esempio, se si vuole cercare l'elemento 225 nel seguente vettore:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
13	44	75	100	120	275	325	510

Il punto centrale del vettore è l'elemento a[3] il cui valore 100 non coincide con 225; poiché 225 è maggiore di 100, la ricerca continua nel semivettore destro:

a[4]	a[5]	a[6]	a[7]
120	275	325	510

quest'ultimo ha un numero pari di elementi, quindi non c'è un solo elemento centrale ma ce ne sono due, a[5] e a[6]. Predeterminiamo arbitrariamente di prendere quello di indice minore, cioè a[5] nel nostro caso; esso contiene il valore 275, che è maggiore di 225, quindi la ricerca continua nel semivettore sinistro:

```
a[4]  a[5]
120   275
```

anche questo ha un numero pari di elementi, quindi prendiamo il primo che ha valore 120; 120 è minore di 225, quindi la ricerca continua nel semivettore destro che contiene un solo elemento, a[5], il cui valore 275 non è il valore cercato, quindi possiamo concludere che il vettore non contiene il valore chiave 225.

Il codice dell'algoritmo di ricerca binaria è:

```
int RicercaBinaria (TipoDato v[], int basso, int alto, TipoDato chiave)
{
    int centrale;
    TipoDato valoreCentrale;
    while (basso <= alto)
    {
        centrale = (basso + alto)/2; // indice elemento centrale
        valoreCentrale = v[centrale]; // valore dell'indice centrale
        if (chiave == valoreCentrale)
            return centrale; // trovato valore, restituisce posizione
        else if (chiave < valoreCentrale)
            alto = centrale - 1; // andare al semivettore basso
        else basso = centrale + 1; // andare al semivettore alto
    }
    return -1; // elemento non trovato
}
```

## 16.2 Analisi degli algoritmi di ricerca

Così come con le operazioni di ordinamento, anche per la ricerca è bene considerare la complessità degli algoritmi impiegati. Il grado di efficienza di una ricerca è essenziale per offrire prodotti realmente utilizzabili e migliori di quelli forniti dalla concorrenza (si pensi ai motori di ricerca su Internet).

La complessità della ricerca sequenziale varia da  $O(1)$ , quando la chiave è nel primo elemento del vettore, a  $O(n)$ , quando l'elemento cercato si trova alla fine del vettore. Per un vettore qualsiasi l'elemento cercato si trova mediamente dopo  $n/2$  confronti, valore che definisce il costo medio della ricerca sequenziale la cui complessità rimane quindi a  $O(n)$ .

Con la ricerca binaria, il caso migliore si presenta quando la chiave si trova nel punto centrale del vettore; si compie infatti un unico confronto e la complessità è  $O(1)$ . La complessità del caso peggiore corrisponde ai casi in cui l'elemento cercato non esiste o viene trovato all'ultimo confronto, e si può valutare considerando che nel caso peggiore il semivettore arriva a essere di un solo elemento. Ogni iterazione che fallisce deve continuare dimezzando la lunghezza del vettore. La dimensione dei semivettori è:

$n \quad n/2 \quad n/4 \quad n/8 \dots \quad 1$

e richiede  $m$  iterazioni, dove  $m$  è approssimativamente  $\log_2 n$ . Ogni iterazione richiede un'operazione di confronto, quindi:

**Tabella 16.1 Confronto tra le ricerche binaria e sequenziale**

Dimensione del vettore	Numero di confronti	
	Ricerca binaria	Ricerca sequenziale
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	1.000.000

Totale confronti  $\approx 1 + \log_2 n$

Come risultato, il caso peggiore della ricerca binaria è  $O(\log_2 n)$ .

La differenza tra i tempi degli algoritmi di ricerca sequenziale e binaria si fa spettacolare al crescere della dimensione del vettore. La Tabella 16.1 compara i metodi di ricerca sequenziale e ricerca binaria in base al numero dei confronti da effettuare nei rispettivi casi peggiori (complessità  $O(n)$  e  $O(\log n)$  rispettivamente).

### 16.3 Algoritmi di ordinamento elementari

Fin dagli anni sessanta, sono stati concepiti vari algoritmi di ordinamento elementari, i cui dettagli implementativi sono stati riportati in innumerevoli testi di programmazione e strutture dati. Probabilmente in questo campo il testo di riferimento più influente è stato il [Knuth, 1973],<sup>1</sup> soprattutto la seconda edizione pubblicata nel 1998 [Knuth, 1998].<sup>2</sup> Gli algoritmi si differenziano molto anche in base alla loro classe di complessità, ma una graduatoria di efficienza non è sempre facile da definirsi perché le loro prestazioni (tempi di ordinamento) possono anche dipendere dal particolare vettore da ordinare. Gli algoritmi di ordinamento classici sono:

- ordinamento per scambio;
- ordinamento per selezione;
- ordinamento per inserimento;
- ordinamento a bolle.

### 16.4 Ordinamento per scambio

Forse l'algoritmo di ordinamento più intuitivo è quello che confronta il primo elemento del vettore con i successivi effettuando uno scambio di posizione allorquando l'ordine dei due elementi confrontati non fosse corretto. Alla fine del ciclo ne parte un altro che ripete la stessa operazione a

<sup>1</sup> Knuth D.E., *The Art of Computer Programming*, vol. 3: *Sorting and Searching*, Addison-Wesley, 1973.

<sup>2</sup> Knuth D.E., *The Art of Computer Programming*, vol. 3: *Sorting and Searching*, 2<sup>a</sup> ed., Addison-Wesley, 1998.

partire dal secondo elemento e così via fino al completo ordinamento del vettore.

Per esempio, prendiamo un vettore di  $n = 4$  elementi, 8, 4, 6, 2 che deve essere ordinato in senso crescente e quindi produrre il vettore ordinato 2, 4, 6, 8. L'algoritmo compie  $n - 1$  iterazioni (3 nell'esempio), compiendo le seguenti operazioni.

#### Iterazione 1

L'elemento di indice 0 ( $a[0]$ ) si confronta con ogni elemento successivo ( $a[1]$ ,  $a[2]$  e  $a[3]$ ) scambiandosi di posto se quest'ultimo è più piccolo. Alla fine dell'iterazione nella posizione di indice 0 ci sarà l'elemento più piccolo del vettore.

#### Iterazione 2

Poiché l'elemento più piccolo è già a posto, si riapplica lo stesso procedimento al sottovettore rimanente (8, 6, 4). Cioè si confronta l'elemento di indice 1 con i successivi e così via, scambiando i posti se il successivo è minore dell'elemento di indice 1. Alla fine dell'iterazione al secondo posto nel vettore ci sarà il secondo elemento più piccolo.

#### Iterazione 3

Nel nostro vettore di quattro elementi la terza iterazione è l'ultima; a questo punto abbiamo due soli elementi da ordinare (8, 6) e quindi un solo confronto da effettuare.

#### Esempio 16.2

Il seguente programma ordina un vettore di 20 elementi e poi lo visualizza su schermo.

```
void Scambio (int &a, int &b)
{
    int aux = a;
    a = b;
    b = aux;
}
// ordinamento per Scambio
void OrdScambio (int a[], int n)
{
    for (int i = 0 ; i < n-1 ; i++)          // compie n-1 iterazioni
        for (int j = i+1 ; j < n ; j++)      // colloca in a[i] il
                                                // minore fra a[i+1]...a[n-1]
            if (a[i] > a[j])
                Scambio (a[i], a[j]) ; // scambia se a[i] > a[j]
}

void StampareVettore (int a[], int n)
{
```

```

for (int i = 0 ; i < n ; i++) cout << a[i] << " ";
cout << endl ;
}
int main()
{
    int vettore[20] = { 30, 35, 38, 58, 14, 15, 50, 27, 10, 20,
                       12, 85, 49, 65, 86, 60, 25, 90, 5, 16 };
    cout << "Vettore originale \n";
    StampareVettore(vettore, 20);
    OrdScambio(vettore, 20);
    cout << "Vettore ordinato \n";
    StampareVettore(vettore, 20);
}

```

### Esecuzione

```

Vettore originale
30 35 38 58 14 15 50 27 10 20 12 85 49 65 86 60 25 90 5 16
Vettore ordinato
5 10 12 14 15 16 20 25 27 30 35 38 49 50 58 60 65 85 86 90

```

## 16.5 Ordinamento per selezione

Anche l'ordinamento per selezione compie successive iterazioni, selezionando però l'elemento minore (o maggiore) del sottovettore non ordinato e mettendolo al suo primo posto. Supponendo sempre di dover ordinare il vettore in senso crescente, si parte selezionando l'elemento più piccolo e ponendolo al primo posto; si procede applicando lo stesso processo al sottovettore non ordinato e così via fino a che quest'ultimo si riduce a un solo elemento (il maggiore del vettore). Anche l'ordinamento per selezione compie  $n - 1$  iterazioni. Per esempio, prendiamo un vettore di  $n = 5$  elementi, 11, 9, 17, 5, 14:

Iterazione 0:	9 < 11 => selezione: 9 17 > 9 => niente 5 < 9 => selezione: 5 14 > 5 => niente scambio: 5, 9, 17, 11, 14
Iterazione 1:	17 > 9 => niente 11 > 9 => niente 14 > 9 => niente
Iterazione 2:	11 < 17 => selezione: 11 14 > 11 => niente scambio: 5, 9, 11, 17, 14
Iterazione 3:	14 < 17 => selezione: 14 scambio: 5, 9, 11, 14, 17 (Vettore ordinato)

Questa è la procedura di "ordinamento per selezione". Provarla al posto della procedura di "ordinamento per scambio" nell'esempio precedente.

```
// ordinamento per Selezione
void OrdSelezione(int a[], int n)
{
    int indicemin;
    for (int i = 0; i < n-1; i++)
    {
        indicemin = i; // posizione del minore
        for(int j = i + 1; j < n; j++)
            if(a[j] < a[indicemin]) indicemin = j; // nuova posizione del minore
        Scambio(a[indicemin],a[i]);
    }
}
```

L'algoritmo di selezione è semplice e richiede un numero di confronti che dipende solo dalla dimensione del vettore e non dalla distribuzione iniziale dei dati. La complessità dell'algoritmo si misura in base al numero di confronti. Nella prima iterazione si fanno  $n - 1$  confronti, nella seconda iterazione  $n - 2$  e così via. Matematicamente si può dire che all'iterazione  $i$ -esima il numero di confronti nel sottovettore  $A[i+1] \dots A[n-1]$  è

$$(n - 1) - (i + 1) + 1 = n - i - 1$$

quindi il numero totale di confronti è

$$C = \sum_{i=0}^{n-2} (n - 1) - i = (n - 1)^2 - \sum_{i=0}^{n-2} i = (n - 1)^2 - \frac{(n - 1) \cdot (n - 2)}{2} = \frac{1}{2} \cdot n \cdot (n - 1)$$

Cioè, per ordinare un vettore di  $n$  elementi, il numero di confronti si calcola sommando gli  $n - 1$  primi interi:

$$(n - 1) + (n - 2) + (n - 3) + \dots + 1 = \frac{n \cdot (n - 1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

La complessità dell'algoritmo è quindi *quadratica*, cioè  $O(n^2)$  (il numero di scambi è  $O(n)$ ). Non vi sono casi migliori né peggiori dato che il numero di iterazioni e la loro lunghezza non dipendono dal particolare vettore esaminato ma solo dalla sua lunghezza.

## 16.6 Ordinamento per inserimento

L'ordinamento per inserimento consiste nell'inserire un elemento alla volta nella posizione che gli spetta in un vettore già ordinato, partendo dal vettore vuoto. Per esempio, si osservi la Figura 16.1 in cui è mostrato il caso del vettore di interi 50 20 40 80 30.

L'algoritmo compie i seguenti passi:

1. il primo vettore costituito dal solo elemento  $A[0]$  si considera ordinato;

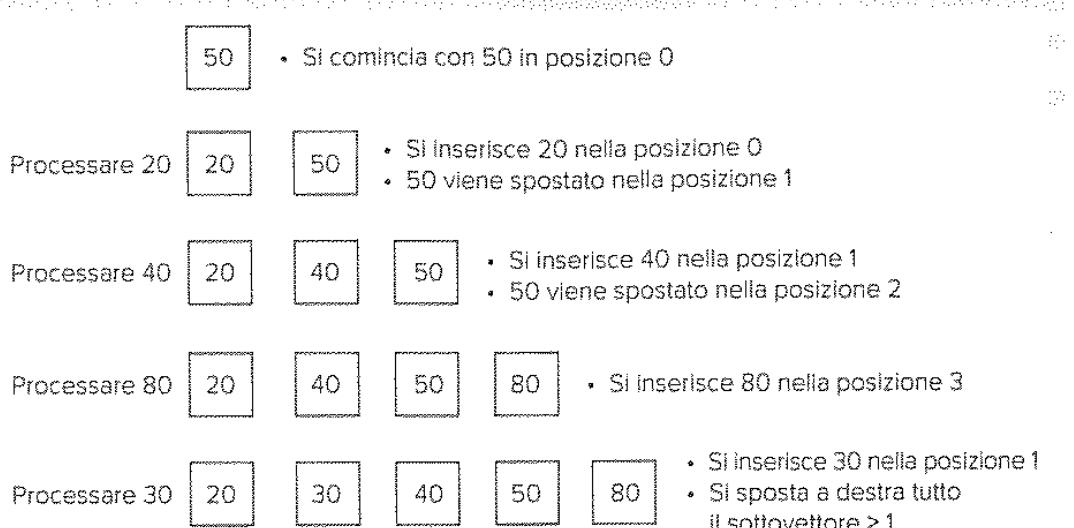


Figura 16.1

Metodo di ordinamento per inserimento: si inseriscono gli elementi uno per uno, da sinistra verso destra.

2. inizia un ciclo for per i che *sale* da 1 a n salvando A[i] in appoggio;
3. all'interno del for inizia un ciclo while per j che *scende* da i fino a un elemento minore o uguale ad appoggio (se non lo trova scende fino a 0); mentre scende, il ciclo while sposta gli elementi che attraversa di un posto verso destra (gli aumenta l'indice di 1);
4. il ciclo for inserisce appoggio nel posto liberatosi.

Questa è la procedura di "ordinamento per inserimento". Provarla nell'esempio precedente.

```
void OrdInserimento (int a[], int n)
{
    int j, appoggio;
    for (int i = 1; i < n; i++)
    {
        appoggio = a[i];
        j = i;
        while (j > 0 && appoggio < a[j-1])
        {
            a[j] = a[j-1];
            j--;
        }
        a[j] = appoggio;
    }
}
```

Questo metodo si può migliorare utilizzando una *ricerca binaria* per collocare correttamente a[i] (salvato in appoggio) nel sottovettore ordinato a[0], ..., a[i-1]. In questo caso lo spostamento dei dati verso destra deve avve-

nire dopo aver trovato la posizione dove inserire l'elemento (che in questa codifica è nella variabile sinistra).

```
void OrdInserimentoBinario(int a[], int n)
{
    int sinistra, destra, centro, appoggio;
    for(int i = 1; i < n; i++)
    {
        appoggio = a[i];
        sinistra = 0;
        destra = i - 1;
        while (sinistra <= destra)
        {
            centro = (sinistra + destra) / 2;
            if(a[centro] > a[i]) destra = centro - 1;
            else sinistra = centro + 1;
        }
        for(int j=i-1; j >= sinistra; j--) a[j + 1] = a[j];
        a[sinistra] = appoggio;
    }
}
```

Anche l'ordinamento per inserimento richiede  $n - 1$  iterazioni. L' $i$ -esima iterazione inserisce nel sottovettore  $a[0] \div a[i]$ , cosa che richiede la media di  $i/2$  confronti. Il numero totale (medio) di confronti sarà quindi:

$$\frac{1}{2} + \frac{2}{2} + \frac{3}{2} + \dots + \frac{n-2}{2} + \frac{n-1}{2} = \frac{n \cdot (n-1)}{4}$$

L'ordinamento per inserimento non utilizza scambi. La complessità dell'algoritmo è data dal numero di confronti ed è cioè  $O(n^2)$ . Il caso migliore capita quando il vettore originale è già ordinato; in questo caso ogni iterazione  $i$  inserisce in  $a[i]$ , il numero dei confronti è 1 e la complessità  $O(n)$ . Il caso peggiore si produce quando il vettore è inversamente ordinato, caso nel quale il numero totale di confronti sarà:

$$(n-1) + (n-2) + \dots + 3 + 2 + 1 = \frac{(n-1) \cdot n}{2}$$

Cioè, la complessità è  $O(n^2)$ .

I metodi di ordinamento per inserimento lineare e inserimento binario, si differenziano soltanto nel metodo della ricerca. Quando l'intervallo ha  $i$  elementi, si realizzano  $\log_2(i)$  confronti. Dunque, il numero di confronti è:

$$C = \sum_{i=1}^{n-1} \log_2 i = \int_1^{n-1} \log_2(x) dx \approx n \cdot \log_2(n)$$

Il metodo della *ricerca binaria* per collocare correttamente  $a[i]$  nel sottovettore ordinato  $a[0], \dots, a[i-1]$  riduce quindi la quantità totale di confronti da

$O(n^2)$  a  $O(n \log_2 n)$ . Però, anche se la posizione corretta si trova in  $O(n \log_2 n)$  passi, ognuno degli elementi  $a[j+1] \dots a[i-1]$  deve muoversi di una posizione e ciò richiede comunque  $O(n^2)$  sostituzioni.

## 16.7 Ordinamento a bolle

Nella tecnica del **BubbleSort** per ogni iterazione si confrontano elementi adiacenti e si scambiano i loro valori quando il primo elemento è maggiore del secondo. Alla fine di ogni iterazione, l'elemento maggiore ha "gorgogliato" fino alla cima del vettore. I passi dell'algoritmo sono:

- all'iterazione 0 si operano  $n - 1$  confronti di elementi adiacenti  
 $(A[0], A[1]), (A[1], A[2]), (A[2], A[3]), \dots, (A[n-2], A[n-1])$   
scambiando i valori di ogni coppia  $(A[i], A[i+1])$  se  $A[i+1] < A[i]$ ; al termine l'elemento maggiore del vettore sarà situato in  $A[n-1]$ .
- all'iterazione 1 si operano gli stessi confronti e scambi, terminando con il secondo maggiore in  $A[n-2]$ .
- Il processo termina all'iterazione  $n - 1$ , nella quale l'elemento più piccolo rimane in  $A[0]$ .

Un esempio illustrerà la tecnica. Sia dato il vettore  $a[]$ :

25 60 45 35 12 92 85 30

Alla prima iterazione si fanno i seguenti confronti:

$a[0]$ con $a[1]$	(25 con 60)	<i>no scambio</i>
$a[1]$ con $a[2]$	(60 con 45)	<i>scambio</i>
$a[2]$ con $a[3]$	(60 con 35)	<i>scambio</i>
$a[3]$ con $a[4]$	(60 con 12)	<i>scambio</i>
$a[4]$ con $a[5]$	(60 con 92)	<i>no scambio</i>
$a[5]$ con $a[6]$	(92 con 85)	<i>scambio</i>
$a[6]$ con $a[7]$	(92 con 30)	<i>scambio</i>

Dopo la prima iterazione il vettore diventa:

25 45 35 12 60 85 30 92

compiendo gli stessi confronti, dopo la seconda iterazione il vettore sarà:

25 35 12 45 60 30 85 92

continuando:

<i>Iterazione 0</i>	25 60 45 35 12 92 85 30
<i>Iterazione 1</i>	25 45 35 12 60 85 30 92
<i>Iterazione 2</i>	25 35 12 45 60 30 85 92
<i>Iterazione 3</i>	25 12 35 45 30 60 85 92
<i>Iterazione 4</i>	12 25 35 30 45 60 85 92
<i>Iterazione 5</i>	12 25 30 35 45 60 85 92
<i>Iterazione 6</i>	12 25 30 35 45 60 85 92
<i>Iterazione 7</i>	12 25 30 35 45 60 85 92

in questo esempio il vettore è ordinato già dopo 5 *iterazioni*, dunque le tre ultime hanno consumato tempo di esecuzione inutilmente. Il metodo a bolle ammette quindi qualche miglioramento per eliminare iterazioni inutili qualora ci si accorga che il vettore è già ordinato. Questa condizione è facile di scoprire poiché si produrrà quando l'ultima iterazione non ha prodotto alcuno scambio; questo fatto si può verificare tramite una *variabile flag* di tipo booleano che cambia valore quando si produce uno scambio.

L'algoritmo migliorato contempla due cicli annidati: il *ciclo esterno* controlla il numero delle iterazioni, e dato che all'inizio della prima iterazione ancora non si è prodotto alcuno scambio, la variabile continua è a false; il *ciclo interno* controlla ogni iterazione individualmente e, quando si produce uno scambio, mette a true il valore di continua. L'algoritmo termina o dopo l'ultima iterazione ( $n - 1$ ) oppure quando continua è false, cioè, non è stato operato alcuno scambio. Questa è la procedura di "ordinamento per BubbleSort"; provarla nell'esempio precedente.

```
void OrdBubbleSort(int a[], int n)
{
    bool continua = true;
    for (int i = 0; i < n-1 && continua; i++)
    {
        continua = false; //non s'è ancora fatto alcuno scambio
        for (int j = 0; j < n-i-1; j++)
            if (a[j] > a[j+1])
            {
                continua = true; // s'è fatto uno scambio
                Scambio(a[j], a[j+1]);
            }
    }
}
```

L'efficienza del BubbleSort dipende dalla versione utilizzata. In quella più semplice si fanno  $n - 1$  iterazioni e  $n - 1$  confronti per ogni iterazione. Pertanto, il numero di confronti è  $(n - 1) \cdot (n - 1) = n^2 - 2n + 1$  cioè, la complessità è  $O(n^2)$ . Con l'uso del flag continua si avrà un'efficienza diversa a seconda del vettore da ordinare. Nel migliore dei casi, si farà una sola iterazione (caso di un vettore già ordinato in ordine ascendente) e la complessità sarà quindi  $O(n)$ . Il caso peggiore coincide con l'altra versione ed è quindi di complessità  $O(n^2)$ . L'analisi del caso generale è complicata. Nel migliore dei casi, l'ordinamento a bolle può terminare in meno di  $n - 1$  iterazioni ma richiede, normalmente, molti più scambi dell'ordinamento per selezione.

## 16.8 Ordinamento Shell

Lo ShellSort è uno dei più vecchi algoritmi di ordinamento, ideato nel 1959 da Donald L. Shell. L'idea generale assomiglia a quella del BubbleSort, ma si scambiano coppie di elementi che stanno a distanza. Si parte da una distanza di  $n/2$ , poi si passa a  $n/4$  e così via fino alla distanza 1, cioè quella del Bubble-

Sort. Con opportune istruzioni di stampa l'esempio che stiamo utilizzando illustra i passaggi dell'algoritmo.

### Esempio 16.2

```

void Scambio (int &a, int &b)
{
    int aux = a;
    a = b;
    b = aux;
}

void StampareVettore (int a[], int n)
{
    for (int i = 0 ; i < n ; i++) cout << a[i] << " ";
    cout << endl ;
}

void OrdShell(int a[], int n)
{
    int j, dist = n/2;
    while (dist > 0)
    {
        cout << "distanza = " << dist << '\t';
        for (int i=dist; i < n; i++)
        {
            j = i - dist;
            while(j >= 0)
            {
                if (a[j] <= a[j+dist]) j = -1;
                else
                {
                    Scambio(a[j],a[j+dist]);
                    j -= dist;
                }
            }
        }
        dist = dist / 2;
        StampareVettore(a,n);
    }
}

int main()
{
    int vettore[20] = {30, 35, 38, 58, 14, 15, 16, 50, 27, 10, 20,
                      12, 85, 49, 65, 86, 60, 25, 90, 8, 16 };
    cout << "originale\t";
    StampareVettore(vettore, 20);
    OrdShell(vettore, 20);
}

```

```

cout << "ordinato\t";
StampareVettore(vettore, 20);
}

```

### Esecuzione

```

originale   30 35 38 58 14 15 50 27 10 20 12 85 49 65 86 60 25 90  5 16
distanza = 10 12 35 38 58 14 15 25 27  5 16 30 85 49 65 86 60 50 90 10 20
distanza = 5  12 25 27  5 14 15 35 38 10 16 30 50 49 58 20 60 85 90 65 86
distanza = 2  10  5 12 15 14 16 20 25 27 38 30 50 35 58 49 60 65 86 85 90
distanza = 1  5 10 12 14 15 16 20 25 27 30 35 38 49 50 58 60 65 85 86 90
ordinato    5 10 12 14 15 16 20 25 27 30 35 38 49 50 58 60 65 85 86 90

```

### ESERCIZI

Una delle applicazioni più frequenti in programmazione è l'ordinamento di una sequenza di elementi. In questo capitolo abbiamo visto gli algoritmi d'ordinamento di base, che sono:

- per scambio;
- per selezione;
- per inserimento.

Il più veloce algoritmo di ordinamento (di tipo "per scambio"), il *QuickSort*, lo

vedremo nel capitolo che tratta la ricorsione. Vi sono poi due metodi di base per la ricerca di elementi in array: **ricerca sequenziale** e **binaria**. La **ricerca sequenziale** (complessità  $O(n)$ ) si utilizza normalmente quando l'array non è ordinato. Se l'array è ordinato in base al campo sul quale s'effettua la ricerca, allora si utilizza la **ricerca binaria** (complessità  $O(\log_2 n)$ ).

### ESERCIZI DI RICORDO

- Algoritmi di ordinamento elementari
- Analisi degli algoritmi di ricerca
- Ricerca in liste: ricerca sequenziale e ricerca binaria
- Ordinamento a bolle
- Ordinamento per inserimento
- Ordinamento per scambio
- Ordinamento per selezione

## Esercizi

Eliminare tutti i numeri duplicati da un vettore. Per esempio, l'array:

4 7 11 4 9 5 11 7 3 5

deve diventare:

4 7 11 9 5 3

Un array contiene gli elementi indicati qui sotto. Utilizzando l'algoritmo di ricerca binaria, tracciare le tappe necessarie per trovare il numero 88.

8	13	17	26	44	56	88	97
---	----	----	----	----	----	----	----

Stessa ricerca per il numero 20.

Un array contiene i seguenti elementi:

3	13	7	26	44	23	98	57
---	----	---	----	----	----	----	----

Quante iterazioni saranno necessarie per ordinarlo in ordine ascendente con i metodi a bolle, per selezione e per inserimento?

**Esercizio** Scrivere un programma che legga 10 nomi e li metta in ordine alfabetico utilizzando il metodo di selezione.

**Esercizio** Ordinare il vettore

42 57 14 40 96 19 08 68

mediante i metodi: (a) bubble; (b) selezione; (c) inserimento.

**Esercizio** Data una sequenza di  $n$  numeri da ordinare:

1. quanti confronti e quanti scambi si richiederanno per ordinare la sequenza col metodo di selezione:

- se sono già ordinati?
- se sono in ordine inverso?

2. e con il metodo a bolle?

**Esercizio** Qual è la differenza tra ricerca e ordinamento?

**Esercizio** I primi due elementi del vettore qui sotto sono stati ordinati utilizzando un algoritmo di inserimento. Come sarà il vettore dopo quattro iterazioni in più?

3 13 8 25 45 23 98 58

**Esercizio** Qual è la differenza tra ordinamento per selezione e ordinamento per inserimento?

**Esercizio** Il vettore 47, 3, 21, 32, 56, 92 dopo due iterazioni di un algoritmo di ordinamento diventa: 3, 21, 47, 32, 56, 92. Che algoritmo di ordinamento si sta utilizzando (selezione, bubble o inserimento)? Giustificare la risposta.

**Esercizio** Utilizzando l'algoritmo di ricerca binaria, tracciare le tappe necessarie per trovare i numeri 88 e 20 nel seguente vettore:

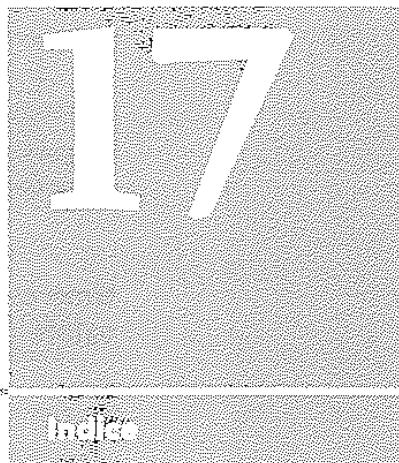
8 13 17 26 44 56 88 97

**Esercizio** Realizzare un programma che permetta l'introduzione di 12 numeri interi in un vettore e, tramite un menù, di eseguire a scelta uno dei seguenti metodi di ordinamento: selezione, scambio, inserimento o bubble. Il programma deve presentare su schermo i vettori ordinati in ordine crescente o decrescente a scelta dell'utente.

**Esercizio** Modificare l'algoritmo di ordinamento per selezione perché ordini un vettore di interi in ordine discendente.

**Esercizio** Modificare il metodo di ordinamento per selezione, perché invece di collocare l'elemento minore nel posto che gli corrisponde, lo faccia con l'elemento maggiore.

# Liste



## 17.1 Le liste

## 17.2 Operazioni con le liste semplici

## 17.3 Lista doppiamente concatenata

## 17.4 Liste circolari

### Introduzione

Questo capitolo riprende il concetto di assegnamento dinamico della memoria per illustrarne una prima fondamentale applicazione: le strutture dati dinamiche. Al contrario delle strutture dati statiche, che hanno dimensione fissa e vengono allocate in memoria durante la compilazione, quelle dinamiche hanno dimensione variabile e vengono allo-

cate durante l'esecuzione del programma.

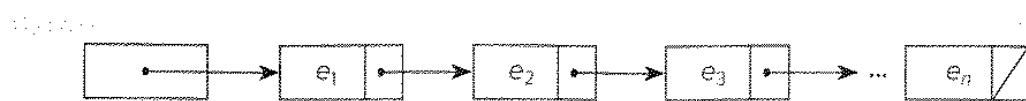
La struttura dati dinamica che si studierà in questo capitolo è la semplice **lista concatenata**, che è una catena di strutture (od oggetti) collegati, uno dopo l'altro, da *puntatori*. Le liste concatenate sono strutture molto flessibili e con numerose applicazioni nel mondo della programmazione.

### 17.1 Le liste

Finora abbiamo utilizzato strutture dati che vengono allocate in memoria o al momento della compilazione (variabili globali), oppure al momento dell'esecuzione di una funzione (variabili locali). Questa tecnica obbliga a fissare in anticipo lo spazio da occupare in memoria per ospitare la struttura dati. Per non incorrere in un errore di *buffer overflow* in fase di esecuzione, bisogna sovradimensionare gli array e metterli in condizione di ospitare il maggior numero di elementi prevedibili per quell'applicazione, risultando ciò in un considerevole spreco medio di memoria. Con la *gestione dinamica della memoria* si possono implementare strutture dati che crescono o diminuiscono con il fabbisogno reale del momento.

Una **lista concatenata** è una sequenza di elementi collegati l'uno all'altro da un puntatore. Gli elementi si compongono cioè di due parti: una che contiene l'informazione (comunque complessa essa sia) e un'altra costituita da un puntatore al successivo elemento della lista (Figura 17.1).

Gli elementi formano una sequenza dal primo all'ultimo. Il primo nodo  $e_1$  si collega a  $e_2$ ,  $e_2$  si collega a  $e_3$  e così via, fino ad arrivare a  $e_n$ . L'ul-



**Figura 17.1**  
Lista concatenata.

ultimo elemento non si collega ad alcuno, il suo puntatore assume quindi il valore 0.

#### 17.1.1 Classificazione delle liste concatenate

Le liste si possono suddividere in quattro categorie:

- *liste semplici*: ogni elemento contiene un unico puntatore che lo collega al nodo successivo; la lista è efficiente se percorsa in *avanti*;
- *liste doppiamente concatenate*: ogni elemento contiene due puntatori, uno all'elemento precedente e l'altro al successivo; la lista è efficiente attraversata in entrambi i sensi;
- *liste circolari semplici*: l'ultimo elemento si collega al primo in modo che la lista possa essere traversata in modo circolare;
- *liste circolari doppiamente concatenate*: l'ultimo elemento si collega al primo elemento e viceversa; questa lista si può attraversare in modo circolare in entrambi i sensi.

Per completezza bisogna dire che ognuno di questi quattro tipi di liste può essere implementato sia con i vettori sia con i puntatori. Ciò comporta differenze nel modo in cui si assegna la memoria, nel come si collegano gli elementi e nel come vi si accede. Con i vettori si fa riferimento all'assegnamento statico della memoria, mentre con i puntatori si può facilmente utilizzare l'assegnamento dinamico. Come già detto, quest'ultimo ha dei chiari vantaggi in termini di risparmio di memoria, anche se può pagare in termini di efficienza computazionale. In questo capitolo e nel successivo faremo riferimento all'implementazione tramite puntatori con memoria dinamica, lasciando come esercizio al lettore l'implementazione tramite vettori.

Ricapitolando, una lista concatenata è una sequenza di elementi costituiti da un campo che contiene l'informazione e un altro che contiene il puntatore al successivo elemento della lista. Il puntatore al primo elemento è detto **puntatore alla testa** della lista. Se la lista non contiene alcun elemento il puntatore alla testa è nullo.

In C++ si può definire un elemento, o "nodo" della lista, mediante i costrutti `struct` o `class`. Il tipo del nodo è quindi definito dal programmatore che deve pensare ad almeno due campi, uno che contiene l'informazione (di qualunque tipo), e uno di tipo *puntatore al tipo* (`struct` o `class`) che lui stesso sta *definendo* (Tabella 17.1). La definizione può utilizzare `struct`, ma se si usa la parola riservata `class` la definizione rimane simile, con l'aggiunta delle potenzialità della programmazione a oggetti. La differenza principale, come ricorderà il lettore, è che in una struttura i campi sono sempre pubblici

Tabella 17.1 Varie dichiarazioni di un nodo

<code>class Nodo { public:     int info;     Nodo* suc; } nodo;</code>	<code>class Nodo { public:     int info;     Nodo* suc; };</code>	<code>typedef ... Elemento; class Nodo { public:     Elemento info;     Nodo* suc; };</code>
Definizione stile C della struttura di un nodo	Definizione stile C++ della struttura di un nodo	Definizione della struttura di un nodo con generalizzazione della struttura informativa all'interno del nodo

mentre in una classe i membri sono privati, a meno che non siano preceduti dalla parola `public`. Poiché il campo "informazione" può essere di qualsiasi tipo (interi, double, caratteri, stringhe, strutture etc.), e perché questo tipo si possa cambiare con facilità, si utilizza di solito `typedef` per definire il nome di questo tipo dentro la struttura del nodo (esempio di destra nella Tabella 17.1), se si dovrà cambiare il tipo di informazione nei nodi, si dovrà soltanto cambiare l'istruzione `typedef`. Se bisogna riferirsi al tipo `elemento` in qualunque parte del programma si può utilizzare l'espressione `Nodo::elemento`.

Potremmo per esempio dichiarare e implementare una classe `Nodo` che contenga un'informazione di tipo `Tipo_elem` e un puntatore a `Nodo` con due attributi protetti:

- `info` di tipo `Tipo_elem`;
- `suc` che è di tipo puntatore alla propria classe `Nodo`;

e tre costruttori:

- il costruttore default;
- il costruttore che inizializza `info` al valore `x`, e `suc` a `NULL`;
- il costruttore che inizializza `info` al valore `x`, e `suc` a `s`.

I metodi per determinare o restituire il contenuto informativo e il puntatore al successivo sono:

- `Tipo_elem get();` // restituisce `info` elemento
- `void put(Tipo_elem info);` // determina `info elem`
- `Nodo* get_suc();` // restituisce successivo
- `void put_suc(Nodo *p);` // determina il nodo successivo

Infine si dichiara per default la funzione membro distruttore. In questo esempio `Tipo_elem` è il tipo intero, ma essendo definito in un `typedef`, può essere facilmente modificato per adattarlo alle necessità.

```
typedef int Tipo_elem;
class Nodo
{
protected:
    Tipo_elem info;
```

```

private Nodo* suc;
public:
    Nodo() {} // costruttore vuoto
    Nodo(Tipo_elem x); // costruttore
    Nodo(Tipo_elem x, Nodo* s); // costruttore
    ~Nodo() {} // distruttore
    Tipo_elem get(); // restituisce info elemento
    void put(Tipo_elem info); // determina info elem
    Nodo* get_suc(); // restituisce successivo
    void put_suc(Nodo *p); // stabilisce successivo
};

Nodo::Nodo(Tipo_elem x) {
    info = x; // costruttore che inizializza info a x e suc a NULL
    suc = NULL;
}

Nodo::Nodo(Tipo_elem x, Nodo* s) {
    info = x; // costruttore che inizializza info a x e suc a s
    suc = s;
}

Tipo_elem Nodo::get() {
    return info; // restituisce una copia del contenuto informativo
}

void Nodo::put(Tipo_elem x) {
    info = x; // determina il contenuto informativo del nodo
}

Nodo* Nodo::get_suc() {
    return suc; // restituisce il puntatore al nodo successivo
}

void Nodo::put_suc(Nodo *p) {
    suc = p; // determina il puntatore al nodo successore
}

```

## 17.2 Operazioni con le liste semplici

Le liste semplici sono quelle con un solo puntatore che contiene l'indirizzo dell'elemento successivo. Sulle liste semplici si possono fare svariate operazioni, ma le più comuni sono:

1. inserimento di un elemento in testa alla lista;
2. inserimento in coda;
3. inserimento nel punto determinato da un ordinamento;
4. estrazione di un elemento dalla testa della lista;
5. estrazione dalla coda;
6. estrazione di un elemento dato.

Ci sono molti modi per implementare queste operazioni, che variano sia per gli algoritmi utilizzati sia per il modo in cui si rappresenta l'elemento della

lista. Rappresentando l'elemento con la classe `Nodo` che abbiamo specificato, possiamo per comodità definire anche un nuovo tipo

```
typedef Nodo* lista;
```

per rappresentare la lista stessa; si intende che sarà di questo tipo il puntatore al primo elemento della lista (tutti gli altri saranno concatenati a seguire). Con questa scelta di comodo le sei operazioni fondamentali potrebbero essere implementate come segue.

#### Inserimento in testa

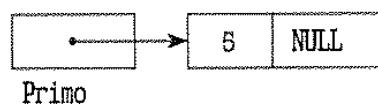
La procedura può essere implementata così:

```
void inserisci_in_testa(lista& inizio, Tipo_elem elem) {
    lista nuovo_nodo = new Nodo(elem, inizio);
    inizio = nuovo_nodo;
}
```

Passiamo il puntatore al primo elemento della lista come riferimento, in maniera che la procedura lo possa modificare; l'elemento da introdurre lo passiamo per valore poiché serve solo per conferire il contenuto informativo e non verrà poi modificato. Con la prima istruzione allochiamo in memoria dinamica il nuovo `Nodo` con il costruttore che copia `elem` nel campo `info` e setta a `inizio` il campo `suc`. Dopodiché modifichiamo il puntatore `inizio` mandandolo a puntare ciò che punta `nuovo_nodo`, ovvero il `Nodo` appena generato e inserito in testa alla `lista`. Se `inizio` era `NULL`, dopo quest'operazione la lista contiene un solo elemento; per esempio, le istruzioni:

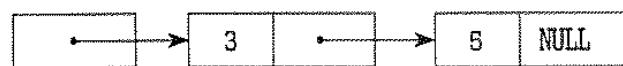
```
lista Primo = NULL;
inserisci_in_testa(Primo, 5);
```

producono la lista illustrata in figura



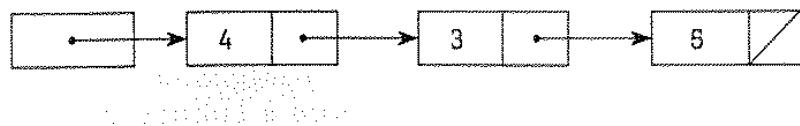
Se lanciamo di nuovo:

```
inserisci_in_testa(Primo, 3);
```



e ancora:

```
inserisci_in_testa(Primo, 4);
```



#### Inserimento in coda

La procedura può essere implementata così:

```

void inserisci_in_coda(lista& inizio, Tipo_elem elem) {
    lista p, q = inizio;
    while (q != NULL) {
        p = q;
        q = q->get_suc();
    }
    q = new Nodo(elem);
    if (inizio == 0) inizio->put_suc(q);
    else p->put_suc(q);
}

```

L'intestazione ha la stessa struttura della precedente, ma qui dobbiamo far partire un ciclo per raggiungere la fine della lista, dove poi si opera l'inserimento. Si utilizzano allo scopo due puntatori: *q* che parte per primo dopo essere stato mandato a puntare l'inizio della lista e *p* che lo riconcorre dentro il ciclo. Il ciclo termina quando *q* prende il valore NULL, cioè arriva al campo suc dell'ultimo elemento, e in quel momento *p* sta un passo indietro, puntando l'ultimo elemento. *q* non serve più e lo riutilizziamo per puntare il nuovo *Nodo* generato dinamicamente con il costruttore che inizializza il campo info a *elem* e il campo suc a NULL. A questo punto bisogna reindirizzare il campo suc dell'elemento puntato da *p* (che era ultimo ma adesso diventa penultimo) che adesso deve puntare l'elemento appena generato (che ora è l'ultimo) puntato da *q*; però questo va fatto se la lista originaria non era vuota, poiché in questo caso *p* non viene istanziato (e quindi dereferenziandolo si produrrebbe errore a run-time) e il puntatore *inizio* va reindirizzato all'elemento appena generato; questo caso è gestito dall'*if* con cui termina la procedura.

### Inserimento nel punto determinato da un ordinamento

La procedura può essere implementata così:

```

void inserisci_ordinatamente(lista& inizio, Tipo_elem elem) {
    lista r, p, q = inizio;
    while ((q != NULL) && (q->get() < elem)) {
        p = q;
        q = q->get_suc();
    }
    r = new Nodo(elem, q);
    if (q == inizio) inizio = r;
    else p->put_suc(r);
}

```

Qui il ciclo deve ovviamente terminare se si è raggiunta la fine della lista, ma questo accade solo se l'elemento da inserire è l'ultimo in base all'ordinamento vigente nella lista; altrimenti si deve fermare quando *q* punta un elemento maggiore o uguale a *elem*. A questo punto si genera dinamicamente il

nuovo **Nodo** con il costruttore che inizializza il campo **info** a **elem** e il campo **suc** a **q**, visto che poi dovrà puntare l'elemento che **q** sta puntando. L'inserimento si completa prendendo l'elemento puntato da **p** e settando a **r** il suo campo **suc**, ma bisogna controllare che non lo abbiamo inserito in testa alla lista (perché la lista era vuota o perché l'elemento che stiamo inserendo è il primo in base all'ordinamento vigente), poiché in questo caso non va rediretto il campo **suc** di **p** (che non è neppure istanziato) ma va rediretto il puntatore **inizio** a **r**, e questo controllo è gestito dall'ultimo **if**.

#### Estrazione di un elemento dalla testa della lista

La procedura può essere implementata così:

```
bool estrai_dalla_testa(lista& inizio, Tipo_elem& elem) {
    lista p = inizio;
    if (inizio == NULL) return false;
    elem = p->get();
    inizio = p->get_suc();
    delete p;
    return true;
}
```

Per le tre funzioni che estraggono elementi dalla lista cambiamo l'intestazione. Innanzitutto non sono più procedure ma funzioni che restituiscono il tipo **bool**, poiché abbiamo bisogno di sapere se l'elemento da estrarre c'era (e in tal caso le funzioni restituiscono **true**) o no (caso in cui restituiscono **false**). Inoltre anche il parametro **elem** cambia tipo: non è più di tipo **Tipo\_elem** ma di tipo **riferimento** al **Tipo\_elem**; questo perché abbiamo bisogno che le funzioni ci restituiscano la variabile che le passiamo modificata dai dati dell'elemento estratto; in effetti non sarebbe necessario ma nelle applicazioni è spesso utile poter poi fare qualcosa con i dati dell'elemento tolto dalla lista, quindi abbiamo bisogno che le funzioni ce li restituiscano.

In questa prima funzione l'operazione è semplice poiché essa non richiede un ciclo. Innanzitutto controlla che la lista non sia vuota, perché in questo caso la funzione termina e restituisce **false**. Superato questo controllo preleva i dati del primo elemento, manda **inizio** a puntare il secondo elemento e recupera la memoria occupata dall'elemento eliminato (**delete p**). Al termine restituisce **true** segnalando con ciò l'avvenuta rimozione.

#### Estrazione di un elemento dalla coda della lista

La procedura può essere implementata così:

```
bool estrai_dalla_coda(lista& inizio, Tipo_elem& elem) {
    lista p, q = inizio;
    if (inizio == NULL) return false;
    while (q->get_suc() != NULL) {
        p = q;
        q = q->get_suc();
    }
    elem = p->get();
    delete p;
    return true;
}
```

```

        q = q->get_suc();
    }
    e = q->get();
    if (q == inizio) inizio = NULL; // se si estra il primo elemento
    else p->put_suc(NULL);
    delete q;
    return true;
}

```

Qui abbiamo bisogno di un ciclo che arrivi al termine della lista. Anche qui si controlla che la lista non sia vuota, dopodiché parte il ciclo che termina quando q punta l'ultimo elemento della lista (il cui campo suc punta NULL). A questo punto la funzione valorizza la variabile passata al secondo argomento con i dati dell'elemento eliminando e setta a NULL il campo suc del penultimo elemento, tagliando così l'ultimo elemento dalla lista; però questo lo fa solo se l'ultimo elemento della lista presa in input non è anche il primo (cioè la lista ha un solo elemento), poiché in questo caso p non è istanziato e inizio va rediretto a NULL (ovvero la lista si vuota); questo controllo è gestito dall'ultimo if. Dopodiché recupera la memoria dell'elemento reciso e restituisce `true`.

#### Estrazione di un elemento dato

La procedura può essere implementata così:

```

bool estrai_elem_dato(lista& inizio, Tipo_elem& elem) {
    lista p, q = inizio;
    while ((q != NULL) && (q->get() != elem)) {
        p = q;
        q = q->get_suc();
    }
    if (q == NULL) return false;
    if (q == inizio) inizio = q->get_suc();
    else p->put_suc(q->get_suc());
    elem = q->get();
    delete q;
    return true;
}

```

Qui il ciclo termina quando trova l'elemento cercato, e se non lo trova arriva fino alla fine della lista; in questo caso q == NULL e la funzione restituisce `false`. Se l'elemento cercato è il primo (q == inizio) inizio viene rediretto a puntare il secondo elemento, altrimenti l'elemento trovato viene bypassato dalla lista mandando il campo suc del precedente a puntare quello puntato dall'elemento eliminando (`p->put_suc(q->get_suc())`). Al termine si recupera la memoria dell'elemento bypassato e si restituisce `true`.

Di seguito è presentato un esempio di applicazione che utilizza queste funzioni e la definizione della classe `Nodo`.

**Esempio 17.1**

```

int main() {
    char r;
    lista scritto = 0;
    Tipo_elem e;
    do {
        cout << endl;
        cout << "i: inserisci in testa " << "c: inserisci in coda "
            << "o: inserisci ordinatamente\n";
        cout << "t: estrai dalla testa " << "e: estrai dalla coda "
            << "f: estrai un particolare elemento\n";
        cout << "s: stampa la lista ";
        cin >> r;
        switch (r) {
            case 'i': cin >> e; inserisci_in_testa(scritto, e); break;
            case 'c': cin >> e; inserisci_in_coda(scritto, e); break;
            case 'o': cin >> e; inserisci_ordinatamente(scritto, e); break;
            case 't': estrai_dalla_testa(scritto, e); cout << "Ho tolto " << e; break;
            case 'e': estrai_dalla_coda(scritto, e); cout << "Ho tolto " << e; break;
            case 'f': cout << "Da togliere? ";
                        cin >> e; estrai_elem_dato(scritto, e); break;
            case 's': stampaLista(scritto); break;
        }
    }
    while (r=='i' || r=='c' || r=='o' || r=='t' || r=='e' || r=='f' || r=='s');
}

```

**Esecuzione di prova**

i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente  
 t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento  
 s: stampa la lista i

**3**

i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente  
 t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento  
 s: stampa la lista c

**5**

i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente  
 t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento  
 s: stampa la lista o

**4**

i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente  
 t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento  
 s: stampa la lista s

**3 4 5**

i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente

```

t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento
s: stampa la lista i
1
i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente
t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento
s: stampa la lista o
2
i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente
t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento
s: stampa la lista s
1 2 3 4 5
i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente
t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento
s: stampa la lista f
Da togliere? 3
i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente
t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento
s: stampa la lista s
1 2 4 5
i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente
t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento
s: stampa la lista e
Ho tolto 5
i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente
t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento
s: stampa la lista t
Ho tolto 1
i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente
t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento
s: stampa la lista s
2 4
i: inserisci in testa c: inserisci in coda o: inserisci ordinatamente
t: estrai dalla testa e: estrai dalla coda f: estrai un particolare elemento
s: stampa la lista w

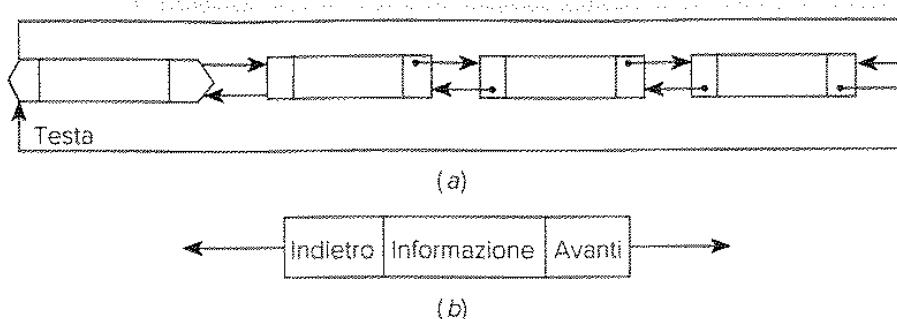
```

### 17.3 Lista doppiamente concatenata

Esistono numerose applicazioni nelle quali è conveniente l'uso di una **lista doppiamente concatenata**. In tale lista, ogni elemento contiene due puntatori: uno punta all'elemento successivo e l'altro al precedente (Figura 17.2).

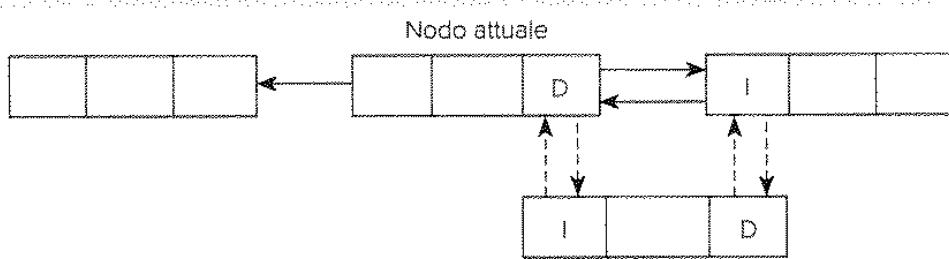
C'è un'operazione di *inserimento* e di *rimozione* per ogni direzione. La Figura 17.3 mostra l'inserimento di un nodo  $p$  a destra del nodo attuale. Debbono essere riassegnati quattro puntatori (due vecchi e due nuovi).

Per rimuovere un nodo da una lista doppiamente concatenata basta cambiare due puntatori (Figura 17.4).



**Figura 17.2** *Diagramma di una lista doppiamente concatenata.*

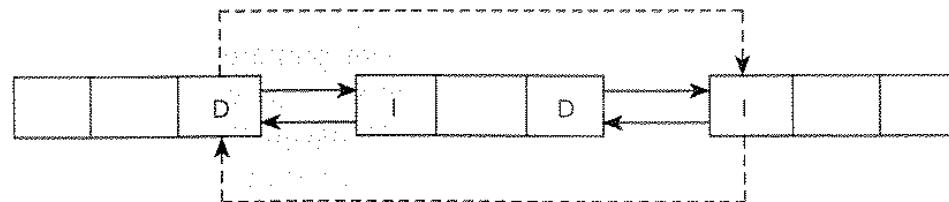
**Lista doppiamente concatenata. (a) Lista con tre nodi; (b) nodo.**



**Figura 17.3** *Inserzione di un nodo in una lista doppiamente concatenata.*

Una lista doppiamente concatenata (con informazione di tipo int) richiede due puntatori, il valore del campo dato e, se il nodo è di tipo class, un costruttore con cui costruire i nuovi elementi. Questo costruttore funzionerà come costruttore default.

```
class Elemento
{
    public :
        Elemento *avanti, *indietro;
        int dato; // dato di tipo intero
        Elemento (Elemento *f = 0, Elemento *b = 0, int d = 0)
            : avanti(f), indietro(b), dato(d) {}
};
```



**Figura 17.4** *Rimozione di un nodo in una lista doppiamente concatenata.*

Ovviamente il nodo può anche essere di tipo struct:

```
struct Nodo
{
    typedef int Elemento;
    Elemento dato;
    Nodo *avanti;
    Nodo *indietro;
};
```

#### Inserimento in testa

Il procedimento si può riassumere in questo algoritmo:

- allocare dinamicamente un nuovo nodo, assegnarne l'indirizzo a un puntatore nuovo e copiarvi dentro l'elemento e che si vuole inserire;
- assegnare il campo suc del nuovo nodo al puntatore di testa p, e il campo prec del primo nodo, se esiste, al nodo nuovo. Se la lista è vuota non fare alcunché;
- far puntare p al nodo nuovo.

#### Inserimento di una determinata posizione

L'inserimento in una determinata posizione richiede le seguenti tappe:

- cercare la posizione dove bisogna inserire il nodo;
- allocare dinamicamente memoria al nuovo nodo puntato dal puntatore nuovo, e copiarvi il contenuto informativo e;
- far puntare il campo suc del nuovo nodo al nodo pos che va dopo la posizione del nuovo nodo ptrnodo (oppure a NULL in caso non vi sia alcun nodo dopo la nuova posizione). L'attributo prec del nodo successivo a quello che occupa la posizione del nuovo nodo che è pos, deve puntare a nuovo se esiste. In caso non esista non fare nulla;
- far puntare l'attributo suc del puntatore prec al nuovo nodo. L'attributo prec del nuovo nodo, farlo puntare a prec.

#### Rimozione di un elemento dato

L'algoritmo per rimuovere un nodo che contiene un dato è simile all'algoritmo di cancellazione per una lista semplice. Ora l'indirizzo del nodo anteriore si trova nel puntatore prec del nodo da cancellare. I passi da seguire sono:

- ricerca del nodo che contiene il dato. Si deve avere l'indirizzo del nodo da rimuovere e l'indirizzo dell'antecedente (prec);
- l'attributo suc del nodo anteriore (prec) deve puntare all'attributo suc del nodo da rimuovere, pos (nel caso in cui non sia il nodo primo della lista). In caso sia il primo della lista l'attributo p della lista deve puntare all'attributo suc del nodo da rimuovere pos;
- l'attributo prec del nodo seguente da cancellare deve puntare all'attributo prec del nodo da rimuovere, nel caso in cui non sia il nodo ultimo. Se il puntatore da rimuovere è l'ultimo non fare nulla;
- da ultimo si libera la memoria occupata dal nodo rimosso pos.

## 17.4 Liste circolari

Nelle liste lineari semplici o nelle doppie vi è sempre un primo nodo e un ultimo nodo che ha l'attributo di riferimento a nullo (NULL). Una lista circolare per propria natura non ha inizio né fine. Tuttavia, è comodo stabilire un nodo a partire dal quale si acceda alla lista, così da poter accedere ai suoi nodi, inserire, cancellare ecc.

### Inserimento di un elemento in una lista circolare

L'algoritmo impiegato per aggiungere o inserire un elemento in una lista circolare varia con la posizione in cui si vuole inserire l'elemento nella lista circolare. In ogni caso bisogna seguire i seguenti passi:

- allocare memoria al nuovo nodo nuovo e riempirlo di informazione da e;
- se la lista è vuota far puntare il suc del nuovo nodo al nodo stesso, e far puntare il puntatore di lista al nuovo nodo;
- se la lista non è vuota si deve decidere dove collocare il nuovo nodo, conservando l'indirizzo del nodo antecedente prec. Collegare l'attributo suc del nuovo nodo con l'attributo suc del nodo antecedente ant. Collegare l'attributo suc del nodo antecedente prec con il nuovo nodo. Se si pretende che il nuovo nodo nuovo già inserito sia il primo della lista circolare, muovere il puntatore della lista circolare al nuovo nodo. Altrimenti non fare nulla.

### Rimozione da una lista circolare semplicemente concatenata

L'algoritmo per rimuovere un nodo di una lista circolare è il seguente:

- cercare il nodo ptrnodo che contiene il dato conservando un puntatore all'antecedente prec;
- far puntare il campo suc del nodo antecedente prec dove punta il campo suc del nodo da cancellare. Se la lista conteneva un solo nodo si mette a NULL il puntatore p della lista;
- se il nodo da rimuovere è quello puntato dal puntatore d'accesso alla lista, p, e la lista contiene più di un nodo, si modifica p per mandarlo dove punta il campo suc del nodo puntato da p (se la lista rimanesse vuota fare prendere a p il valore NULL);
- da ultimo si libera la memoria occupata dal nodo eliminato.

### Concetti

In questo capitolo abbiamo introdotto il concetto di *lista concatenata*: sequenza di elementi nella quale ognuno punta al successivo. Ogni nodo della lista contiene due parti: informazione e puntatore. Le operazioni fondamentali su una lista sono la visita, l'aggiunta di elementi (in testa, in coda o in un punto preordinato)

e la cancellazione di elementi (dalla testa, dalla coda o la cancellazione di un particolare elemento).

In una *lista doppialmente concatenata* ogni nodo ha anche un puntatore al suo precedente. In una *lista concatenata circolarmente* il puntatore dell'ultimo nodo punta al primo elemento della lista.

- Rimozione di elementi da una lista
- Struttura di una lista
- Fondamenti teorici
- La classe *Pila* implementata tramite liste
- Lista doppiamente concatenata
- Operazioni con le liste
- Attraversamento di una lista

## Esercizi

**E1** Scrivere una funzione che calcoli il numero di nodi di una lista concatenata.

**E2** Scrivere una funzione che elimini da una lista l' $n$ -esimo elemento.

**E3** Scrivere il codice che mostra se una lista concatenata è vuota.

**E4** Scrivere il frammento di codice che crea la lista concatenata con i dati 1, 2 ... 20.

**E5** Scrivere una funzione che conti il numero di volte che una determinata chiave si ripete in una lista sequenziale.

**E6** Scrivere un algoritmo che legga una lista di interi in input, crei una lista concatenata con essi e stampi il risultato della stessa.

**E7** Scrivere un algoritmo che accetti una lista concatenata, l'attraversi e restituisca il dato del nodo con il valore minore.

**E8** Scrivere un programma che scambi due nodi di una lista concatenata. I nodi si identificano per numeri e si passano come parametri. Per esempio, per scambiare i nodi 5 e 8 si deve invocare `scambio(5, 8)`. Se lo scambio si realizza con successo, si restituisce vero; se si produce un errore, come un numero di nodo non valido, si restituisce falso.

**E9** Scrivere una funzione membro di una classe lista semplicemente concatenata che restituisca vero «true» se è vuota e falso «false» altrimenti; scrivere un'altra funzione membro che crei una lista vuota.

**E10** Scrivere una funzione membro che restituisca il numero di nodi di una lista semplicemente concatenata.

**E11** Scrivere una funzione membro di una classe lista semplicemente concatenata che cancelli il primo nodo.

**E12** Scrivere una funzione membro di una classe lista semplicemente concatenata che mostri gli elementi nell'ordine in cui si trovano.

**E13** Scrivere una funzione membro di una classe lista semplicemente concatenata che restituisca il primo elemento della classe lista, se esiste.

**E14** Scrivere una funzione membro di una classe lista doppiamente concatenata che aggiunga un elemento come primo nodo della lista.

**E15** Scrivere una funzione membro di una classe lista doppiamente concatenata che restituisca il primo elemento della lista.

**E16** Scrivere una funzione membro di una classe lista doppiamente concatenata che elimini il primo elemento della lista, se esiste.

**E17** Scrivere una funzione membro della classe lista doppiamente concatenata che decida se è vuota.

**E18** Scrivere un programma che legga un file e costruisca una lista concatenata. Una volta costruita, essa si visualizza sullo schermo. Si può utilizzare qualunque struttura dati, sempre che abbia un campo

chiave e dati. Due casi: una lista di CD di musica oppure un'agenda di numeri di telefono.

**Esercizio 1** Supponiamo che si dispone di una lista concatenata nella quale i dati di ogni elemento contengono un intero. Scrivere una funzione per determinare se la lista è ordinata.

**Esercizio 2** Scrivere un programma che legga una lista di studenti da un file e crei una lista concatenata. Ogni entrata della lista concatenata deve avere il nome dello studente, un puntatore al seguente studente e un puntatore a una lista concatenata di qualificazioni. Per esempio, cinque qualificazioni per ogni studente.

**Esercizio 3** Scrivere un programma che crei un array di liste concatenate.

**Esercizio 4** Un vettore disperso è un vettore che ha diversi elementi che sono zeri. Scrivere un programma che permetta di rappresentare tramite liste un vettore disperso. Poi realizzare le operazioni:

- somma di due vettori dispersi;
- prodotto scalare di due vettori dispersi.

**Esercizio 5** Data una lista doppiamente concatenata di numeri interi, scrivere il programma necessario per ordinare la detta lista in ordine crescente.

**Esercizio 6** Scrivere una funzione che prenda una lista concatenata di interi e inverta l'ordine dei suoi nodi.

**Esercizio 7** Si dispone di una lista doppiamente concatenata ordinata con chiavi ripetute. Progettare una funzione di inserzione di una chiave nella lista in modo tale che se la chiave già si trova nella lista si inserisca alla fine di tutte quelle che hanno la stessa chiave.

**Esercizio 8** In una lista semplicemente concatenata  $L$  si trovano nomi di persone ordinati alfabeticamente. Partendo dalla detta lista  $L$  creare una lista doppiamente concatenata  $L'$  di tale forma che il puntatore di inizio della lista punti alla posizione centrale. Si presuppone che la posizione centrale sia il nodo che occupa la posizione  $n/2$ , essendo  $n$  il numero di nodi della lista.

**Esercizio 9** Scrivere un programma che legga un testo di lunghezza indeterminata e produca come risultato la lista di tutte le parole differenti contenute nel testo così come la loro frequenza di apparizione.

**Esercizio 10** Utilizzare una lista concatenata per controllare una lista di passeggeri di una linea aerea. Il programma principale deve essere controllato per menu e permettere all'utente di visualizzare i dati di un passeggero determinato, visualizzare la lista completa, creare una lista, inserire un nodo, cancellare un nodo e sostituire i dati personali di un determinato passeggero.

**Esercizio 11** Disegnare una funzione che inserisca nodi in una lista concatenata, in una posizione determinata. Ripetere l'operazione per il caso di una lista doppiamente concatenata.

**Esercizio 12** Costruire un programma che gestisca una lista doppiamente concatenata che deve rimanere ordinata durante l'esecuzione del programma.

**Esercizio 13** Scrivere il codice di un costruttore di copia di una classe lista semplicemente concatenata.

**Esercizio 14** Scrivere il codice di un distruttore che elimini tutti i nodi di un oggetto della classe lista semplicemente concatenata.

**Esercizio 15** Scrivere una funzione membro della classe lista semplicemente concatenata, che inserisca in una lista concatenata ordinata in modo crescente un elemento  $x$ .

**Esercizio 16** Scrivere una funzione membro di una lista semplicemente concatenata, che elimini tutte le apparizioni di un elemento  $x$  che riceva come parametro.

**Esercizio 17** Scrivere un metodo della classe lista semplicemente concatenata che elimini un elemento  $x$  che riceva come parametro di una lista concatenata ordinata.

**Esercizio 18** Scrivere una funzione membro di una classe lista semplicemente concatenata che ordini la lista muovendo soltanto puntatori.

**Esercizio 19** Scrivere una funzione che non sia membro della classe lista semplicemente concatenata, che riceva come parametro due oggetti di tipo lista che contengono due

liste ordinate in modo crescente e le mischi in un'altra lista che riceva come parametro.

Scrivere il codice di un costruttore di copia di una classe lista doppiamente concatenata.

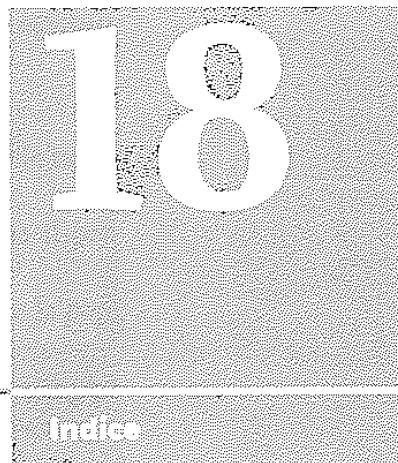
Scrivere una funzione membro di una classe lista doppiamente concatenata che inserisca un elemento  $x$  nella lista, rimanendo essa ordinata.

Scrivere una funzione membro di una classe lista doppiamente concatenata che cancelli un elemento  $x$  dalla lista.

Scrivere una funzione che non sia membro della classe lista doppiamente concatenata che copi una lista doppia in un'altra.

Scrivere una classe lista semplicemente concatenata circolare che permetta di decidere se la lista è vuota, di vedere qual è il primo della lista, di aggiungere un elemento come primo della lista, di cancellare il primo elemento della lista e infine di cancellare la prima apparizione di un elemento  $x$  nella lista.

# Pile e code



**18.1** Concetto e gestione di una pila

**18.2** Concetto e gestione di una coda

[Indice](#)

## Introduzione

In questo capitolo si studiano in dettaglio le strutture dati pila (stack) e coda (queue) che sono probabilmente le strutture dati dinamiche più utilizzate. Esse immagazzinano e recuperano i loro elementi secondo

un ordine ben preciso: le pile, con strategia **LIFO** (*Last In First Out*, ultimo in ingresso, primo in uscita), e le code con strategia **FIFO** (*First In First Out*, primo in ingresso, primo in uscita).

### 18.1 Concetto e gestione di una pila

Una **pila** (*stack*) è una lista alla quale si può accedere solo da un estremo. Gli elementi della pila si aggiungono o si rimuovono solo dalla sua parte "superiore" (il termine si usa per analogia con una pila di piatti, una pila di pratiche nell'ufficio di un impiegato ecc.).

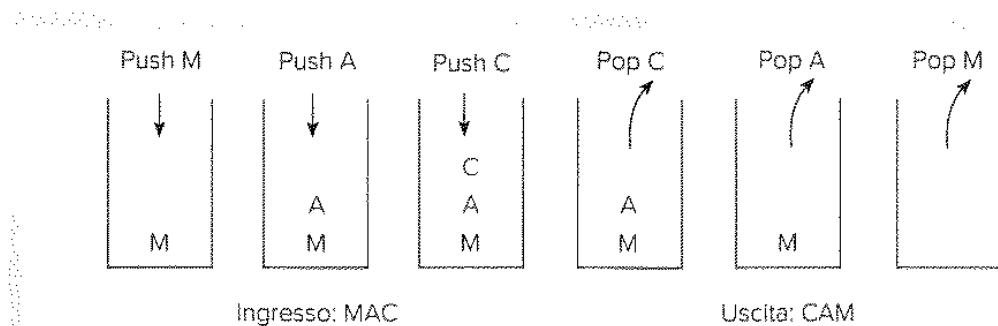
La pila è gestita con strategia *Last In First Out* (LIFO), cioè la sequenza di estrazione di elementi dallo stack è esattamente l'inversa di quella di immessione, cosicché il primo elemento a essere immesso sarà anche l'ultimo a essere estratto.

Le operazioni usuali nella pila sono *Inserzione* e *Rimozione*. L'operazione **Inserire** (push) aggiunge un elemento alla cima della pila e l'operazione **Rimuovere** (pop) elimina o toglie un elemento dalla pila. La Figura 18.1 mostra una sequenza di operazioni push e pop in uno stack.

L'operazione *push* mette un elemento nella cima della pila e *pop* lo elimina dalla cima.

La pila si può implementare tramite array, nel qual caso la sua dimensione o lunghezza è fissa, o tramite puntatori e liste concatenate, nel qual caso si utilizza memoria dinamica e non vi è alcuna limitazione alla sua dimensione.

Una pila può essere *vuota* o *piena* (solo nel caso di dimensione massima fissa). Se un programma tenta di rimuovere un elemento da una pila vuota, si produrrà un errore, detto di **underflow**. Invece, se un programma tenta



**Figura 18.1**  
Inserire e prelevare elementi dalla pila.

di inserire un elemento in una pila piena, si produce un errore chiamato **overflow**. Per evitare queste situazioni si progettano funzioni che verificano se la pila è piena o vuota.

1973 - Implementare in C base Pilha com um array

Come anticipato, una pila si può implementare tramite array (dimensione massima statica) o tramite liste (dimensione dinamica), tramite una classe oppure utilizzando i template.

Se si utilizza un array per contenere gli elementi della pila, una prima considerazione da fare è che la dimensione della pila non può oltrepassare il numero di elementi dell'array; la condizione *pila piena* sarà significativa per il progetto. Servirà quindi un indice/puntatore alla cima della pila e bisognerà definire l'insieme di operazioni sulla pila.

Normalmente la *base* della pila è nella posizione 0 dell'array, quindi si inizia a lavorarci definendo una *pila vuota*. Si introducono poi elementi nell'array a partire dalla posizione 0, alla posizione 1, alla 2 e così via. Basta ogni volta incrementare di 1 il puntatore alla sommità della pila. Gli algoritmi *push* e *pop* utilizzando l'indice dell'array come puntatore alla sommità della pila sono:

*push*

1. Verificare che la pila non è piena.
  2. Incrementare di 1 il puntatore della pila.
  3. Mettere l'elemento nella posizione del puntatore della pila.

DOD

1. Verificare che la pila non è vuota.
  2. Leggere l'elemento della posizione del puntatore della pila.
  3. Decrementare di 1 il puntatore della pila.

Se l'array ha massimo DimPila elementi, l'indice della pila sarà compreso fra 0 e DimPila-1; *in una pila piena* il puntatore della pila punta a DimPila-1 e *in una pila vuota* il puntatore della pila punta a -1, poiché 0 sarà l'indice del primo

elemento. Per dichiarare una pila bisogna definire il tipo del dato e aggiungere le altre operazioni ammesse, ovvero bisogna definire:

1. il tipo dei dati della pila (`Tipo_elem`, tramite `typedef`);
2. funzione `Pilapiena`: verifica che la pila non sia piena (prima di effettuare la `push`); restituisce `true` se la pila è piena e `false` in caso contrario; comportamento: se la pila è piena dà messaggio di errore e termina il programma;
3. funzione `Pilavuota`: verifica che la pila non sia vuota (prima di effettuare la `pop`); restituisce `true` se la pila è vuota e `false` in caso contrario; comportamento: se la pila è vuota dà messaggio di errore e termina il programma;
4. funzione `Svuota`: svuota la pila, lasciandola senza elementi e disponibile per altri compiti;
5. `Top`; restituisce il valore situato in cima alla pila, ma non decrementa il puntatore della pila, che rimane quindi intatta.

```
#define MaxDimPila 100;
class Pila
{
private:
    Tipo_elem vettorepila[MaxDimPila];
    int cima;
public:
    Pila() : cima(-1) {};
    void Push(Tipo_elem& elemento);
    Tipo_elem Pop();
    void Svuota();
    Tipo_elem Top() const;
    int Pilavuota() const;
    int Pilapiena() const;
};
```

#### Esercizio 18.1

*Scrivere un programma che manipoli la classe `Pila` definita prima introducendo un dato di tipo intero (25).*

```
Pila P;
P.Push(25);
cout << P.Top () << endl;
if (!P.Pilavuota())
    aux = P.Pop();
cout << aux << endl;
P.Svuota ();
```

Le operazioni della pila definite nella classe `Pila` sono: `Push`, `Pop` e `Top`. Le operazioni `Push` e `Pop` inseriscono ed eliminano un elemento nella/dalla pila; l'operazione `Top` accede alla cima della pila senza rimuoverne l'elemento.

L'operazione Push incrementa la cima di 1 e immette il nuovo elemento sulla pila. Il tentativo di aggiungere un elemento a una pila piena produce un messaggio di errore "Stack Overflow" e termina il programma.

```
void Pila::Push(const Tipo_elem elemento)
{
    if (cima == MaxDimPila-1)
    {
        cerr << "Stack Overflow" << endl;
        exit(1);
    }
    cima++;
    vettorepila[cima] = elemento;
}
```

L'operazione Pop elimina l'elemento cima dalla pila copiandone prima il valore in una variabile locale ausiliare e, poi, decrementandone la cima di 1. La variabile aux si restituisce nell'esecuzione dell'operazione Pop. Il tentativo di togliere un elemento da una pila vuota produce un messaggio di errore "Stack Underflow" e termina il programma.

```
Tipo_elem Pila::Pop(void)
{
    Tipo_elem Aux;
    if (cima == -1)
    {
        cerr << "Stack Underflow" << endl;
        exit(1);
    }
    aux = vettorepila[cima];
    cima--;
    return aux;
}
```

Si deve proteggere l'integrità della pila, perciò la classe Pila deve fornire operazioni che ne verifichino lo stato: Pilavuota o Pilapiena. Inoltre, si deve definire un'operazione Svuota che ripristini la condizione iniziale della pila determinata dal costruttore (cima della pila a -1).

La funzione Pilavuota verifica se la cima della pila è -1. In quel caso, la pila è vuota e si restituisce true; in caso contrario, si restituisce false.

```
int Pila::Pilavuota() const
{
    return cima == -1;
}
```

La funzione Pilapiena verifica se la cima è MaxDimPila-1. In quel caso, la pila è piena e si restituisce true; in caso contrario, si restituisce false.

```
int Pila::Pilapiena() const
```

```
[ return cima == MaxDimPila-1;
}
```

Da ultimo, Svuota reinizializza la cima al suo valore iniziale con la pila vuota (-1).

```
void Pila::Svuota()
{
    cima = -1;
}
```

### Esempio 10.2

#### *Classe pila implementata con un array.*

Si definisce la costante `MaxDimPila=100` che sarà il massimo numero di elementi che potrà contenere la pila. Si definisce in un `typedef` il tipo di dati che `Pila` contiene (in questo caso saranno interi). `Pila` è una classe i cui attributi sono la cima che punta sempre l'ultimo elemento aggiunto alla pila e un array `A` i cui indici varieranno tra 0 e `MaxDimPila-1`. Tutte le funzioni membro della classe sono pubbliche, eccetto `Piena` che è privata.

- `Svuota`: crea la pila vuota mettendo la cima nel valore -1.
- `Pila`: è il costruttore della `Pila`, coincide con `Vuota`.
- `vuota`: decide se la pila è vuota (cima vale -1).
- `piena`: nell'implementazione di una pila con array, bisogna dichiararla come privata per prevenire possibili errori; in questo caso la pila sarà piena quando la cima punta al valore `MaxDimPila-1`.
- `Aggiunge`: aggiunge un elemento alla pila; perciò verifica in primo luogo che la pila non sia piena, e, in caso affermativo, incrementa la cima di un'unità, per poi mettere l'elemento nella posizione `cima` dell'array `A`.
- `Primo`: verifica che la pila non sia vuota, e restituisce l'elemento dell'array `A` nella posizione puntata da `cima`.
- `Cancella`: elimina l'ultimo elemento messo nella pila; prima verifica che la pila non sia vuota, quindi diminuisce la cima di un'unità.
- `Pop`: estrae il primo elemento dalla pila e lo cancella; può essere implementata direttamente, oppure chiamando le primitive `Primo` e poi `Cancella`.
- `Push`: coincide con `Aggiunge`.

```
typedef int Tipo_elem;
class Pila
{
protected:           // questa è l'implementazione interna
    Tipo_elem A[MaxDimPila];
    int cima;
    void Aggiunge(const Tipo_elem elemento);
    bool Piena() { return cima == MaxDimPila-1; };
    bool Vuota() { return cima == -1; };
}
```

```
public:          // questa è l'interfaccia pubblica
    Pila() { cima = -1; }           // genera la pila
    ~Pila() {};
    void Svuota() { cima = -1; }    // svuota la pila
    void Push(Tipo_elem elemento) { Aggiunge(elemento); }
    Tipo_elem Pop();             // preleva l'elemento
    Tipo_elem Top();              // vede l'elemento
    void Cancella();             // cancella l'elemento
};

void Pila::Aggiunge(Tipo_elem elemento) {
    if (Piena())
    {
        cout << "Stack Overflow";
        exit(1);
    }
    cima++;
    A[cima] = elemento;
}

Tipo_elem Pila::Pop() {
    Tipo_elem Aux;
    if (Vuota())
    {
        cout << "Stack Underflow";
        exit(1);
    }
    Aux = A[cima];
    cima--;
    return Aux;
}

Tipo_elem Pila::Top() {
    if (Vuota())
    {
        cout << "Stack Underflow";
        exit(1);
    }
    return A[cima];
}

void Pila::Cancella() {
    if (Vuota())
    {
        cout << "Stack Underflow";
        exit(1);
    }
    cima--;
}
```

```

int main(int argc, char *argv[]) {
    Pila P;
    P.Push(5);
    P.Push(6);
    cout << P.Pop() << endl;
    cout << P.Pop() << endl;
    return 0;
}

```

### Esecuzione

```

6
5

```

#### 18.1.2 Implementare la classe pila con liste dinamiche

Per implementare una pila con puntatori basta usare una lista semplicemente concatenata; ma piena, la pila sarà vuota se il puntatore di testa, cioè la cima della pila, è NULL. La pila si dichiara come una classe con un attributo protetto che è un puntatore a nodo.

Gli algoritmi *push* e *pop* sono:

- *push*: aggiungere un nuovo nodo, con il dato che si vuole inserire, come primo elemento della lista;
- *pop*: verificare se la lista non è vuota; estrarre il valore del primo nodo; cancellare il primo nodo.

### Esempio 18.3

*Classe Pila implementata con puntatori.*

Si definisce la classe *Nodo* già implementata nel Capitolo 17 e utilizzata nelle liste semplicemente concatenate. La classe *Pila* ha come attributo protetto un puntatore alla classe *Nodo*. Le funzioni membro della classe *Pila* sono:

- costruttore *Pila*: crea la pila vuota mettendo l'attributo *p* a NULL;
- costruttore di copia *Pila* utilizzato per la trasmissione di parametri per valore da una pila a una funzione;
- distruttore *-Pila*: recupera la memoria di tutti i nodi della lista;
- *Svuota*: crea la pila vuota mettendo l'attributo *p* a NULL;
- *vuota*: decide se la pila è vuota, valuta cioè se l'attributo *p* vale NULL;
- *Aggiunge*: aggiunge un elemento alla pila; per fare ciò aggiunge un nuovo nodo che contenga come informazione l'elemento che si vuole aggiungere e lo inserisce in testa;
- *Primo*: verifica che la lista non sia vuota e restituisce l'informazione del primo nodo della lista;
- *Cancella*: elimina l'ultimo elemento immesso nella pila; prima verifica che la lista non sia vuota, quindi rimuove la testa della lista;

- Pop: estrae il primo elemento della pila e lo cancella; può essere implementata direttamente, oppure chiamando le primitive Primo e, poi, Cancellare;
- Push: coincide con Aggiunge.

```

typedef int Tipo_elem;

class Nodo {
    protected:
        Tipo_elem e;
        Nodo *Seg;
    public:
        Nodo()
        ~Nodo()
        Nodo (Tipo_elem x){ e = x; Seg = NULL;}      // costruttore
        Tipo_elem get(){ return e;}                     // Ottiene elemento
        void put(Tipo_elem x){ e = x;}                 // Assegna elemento
        Nodo * get_suc(){ return Seg;}                 // Ottiene successivo
        void put_suc( Nodo *p){ Seg = p;}              // Assegna successivo
    };
class Pila
{
    protected:
        Nodo *p;
    public:
        Pila(){p = NULL;}                            // costruttore
        Pila (const Pila&);                         // costruttore di copia
        ~Pila();
        void Svuota(){ p = NULL;}
        bool vuota(){ return !p;}
        Tipo_elem Primo(){ if (p) return p->get();}
        void Aggiunge( Tipo_elem e);
        void Push(Tipo_elem e){ Aggiunge(e);}
        void Cancellare();
        Tipo_elem Pop();
};

void Pila::Aggiunge(Tipo_elem e) {
    Nodo *aux;
    aux = new Nodo(e);
    if( p )
        aux->put_suc(p);
    p = aux;
}

void Pila::Cancellare() {
    Nodo *paux;
    if(p)
    [

```

```

    paux = p;
    p = p->get_suc();
    delete paux;
}

}

Pila::Pila (const Pila &p2) {
    Nodo* a = p2.p, *af, *aux;
    p = NULL;
    Aggiunge(-1); // aggiunge Nodo Testa
    af = p;
    while ( a != NULL)
    {
        aux = new Nodo(a->get());
        af->put_suc(aux);
        af = aux;
        a = a->get_suc();
    }
    Cancella(); // cancella nodo Testa
}
Pila::~Pila() {
    Nodo *paux;
    while(p != 0)
    {
        paux = p;
        p = p->get_suc();
        delete paux;
    }
}
Tipo_elem Pila::Pop() {
    Tipo_elem e;
    e = Primo();
    Cancella();
    return e;
}
int main(int argc, char *argv[])
{
    Pila p;
    p.Svuota();
    p.Aggiunge(5);
    p.Aggiunge(6);
    while (!p.vuota())
    {
        cout << p.Primo() << endl;
        p.Cancella();
    }
}

```

```
    return 0;
}
```

### Esecuzione

```
6  
5
```

## 18.2 Concetto e gestione di una coda

La **coda** è una struttura dati gestita con politica *First In First Out* (FIFO). Pensata come un vettore, permette di accedere ai dati da ciascuno dei suoi due estremi (Figura 18.2). Gli elementi si immettono in coda e si rimuovono dalla testa nello stesso ordine in cui sono stati immessi.

Le *azioni* permesse in una coda sono:

- creazione di una coda vuota;
- verifica che una coda sia vuota;
- aggiunta di un dato alla fine della coda;
- eliminazione di un dato dalla testa della coda.

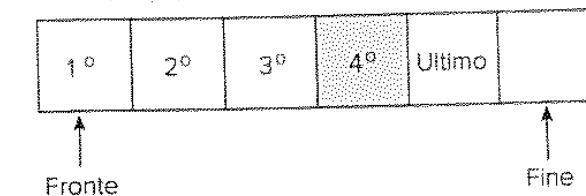
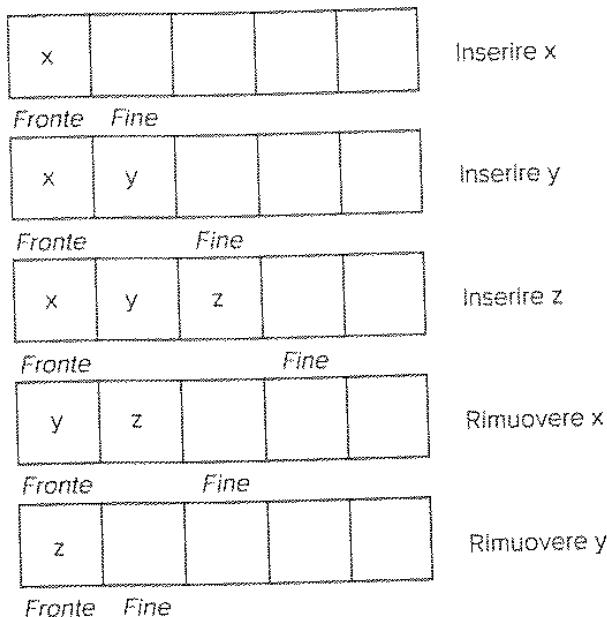


Figura 18.2

Una coda.

### 13.2.1 La classe coda implementata con un array

Come le pile, le code possono essere implementate tramite vettori o liste dinamiche. In questa sezione considereremo l'implementazione tramite un vettore. Pensiamo a una classe Coda che contiene un array (vettoreQ) per immagazzinare elementi di un tipo generico `Tipo_elem`, e due indici, fronte e fine; il primo punta la testa della coda e l'altro la prima cella vuota dell'array che segue la fine della coda. Quando si inserisce in coda, si verifica prima se fine punta una posizione valida; se "sì", si aggiunge l'elemento in coda e si incrementa fine di 1. Quando si rimuove dalla coda, si fa una prova per vedere se la coda è vuota e, se così non è, si recupera l'elemento dalla posizione puntata da fronte che viene incrementato di 1.

La massima dimensione del vettore è determinata dalla costante `MaxDimQ`.

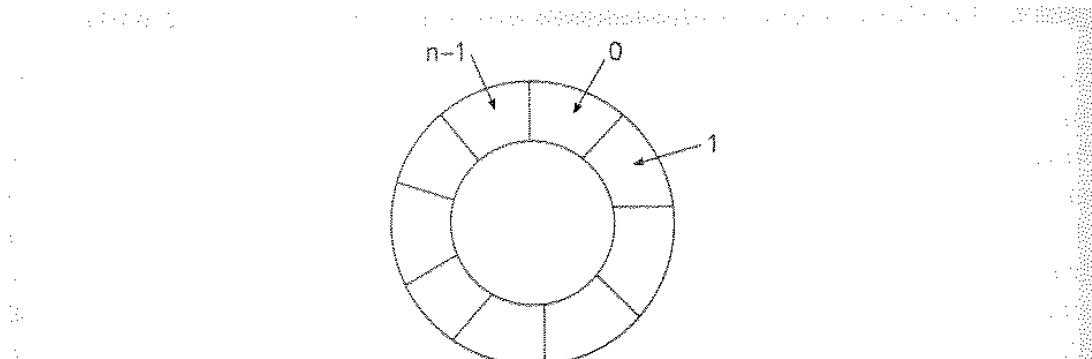
L'indice `conto` controlla il numero di elementi della coda e serve per determinare se la coda è vuota o piena.

Le operazioni tipiche della coda sono: `InserireQ`, `RimuovereQ`, `Qvuota`, `Qpiena` e `FronteQ`. `InserireQ` prende un elemento di tipo `Tipo_elem` e lo inserisce in fine. `RimuovereQ` elimina e restituisce l'elemento da fronte. Il metodo `FronteQ` restituisce il valore dell'elemento in fronte. L'operazione `Qvuota` verifica se la coda è vuota prima di eliminare un elemento, mentre `Qpiena` verifica se la coda è piena prima di inserire un nuovo elemento. Se le precondizioni per `InserireQ` e `RimuovereQ` non sono soddisfatte il programma deve stampare un messaggio di errore e terminare.

```
#define MaxDimQ 100;
class Coda
{
    private:
        int fronte, fine, conto;
        Tipo_elem vettoreQ[MaxDimQ];
    public:
        Coda();
        void InserireQ(const Tipo_elem& elemento);
        Tipo_elem RimuovereQ(void);
        void CancellareCoda(void);
        Tipo_elem FronteQ();
        int LunghezzaQ();
        int Qvuota();
        int Qpiena();
};
}
```

La dichiarazione e l'implementazione della coda si possono mettere in un header file "codarray.h".

Questo procedimento funziona bene fino alla prima volta che l'indice fine raggiunge la fine dell'array perché non ci sarà più posto per aggiungere elementi. Ma se durante questo tempo sono occorse eliminazioni, ci sarà spazio vuoto all'inizio dell'array, quindi si potrebbero spostare verso il fronte tutti gli elementi di tanti posti quante sono state le rimozioni, ma questo



**Figura 18.3**  
Un array circolare.

richiede un tempo di computazione proporzionale al numero dei dati in coda. Il modo migliore per gestire una coda tramite array è quello di utilizzare un tipo speciale di array che unisca l'ultimo elemento con il primo; questo array *circolare* (Figura 18.3) permette di implementare la coda senza necessità di spostare alcun elemento.

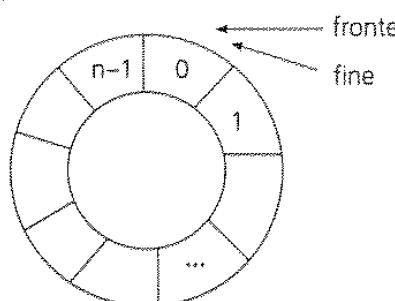
Una coda vuota si rappresenta per la condizione fronte=fine (Figura 18.4).

Dopo le operazioni che li coinvolgono, sia fronte sia fine si incrementano in senso orario. La variabile cont mantiene il numero di elementi nella coda, e se cont=MaxDimQ la coda è piena. Sia fronte sia fine si incrementano circolarmente all'infinito, quindi per associargli il loro corrispondente indice dell'array si utilizza l'operazione "modulo":

muovere fine avanti:     $\text{fine} = (\text{fine} + 1) \% \text{MaxDimQ}$   
 muovere testa avanti:     $\text{fronte} = (\text{fronte} + 1) \% \text{MaxDimQ}$

Gli algoritmi per la gestione di code in array circolari includono almeno i seguenti compiti (Figura 18.5).

- Creazione di una coda vuota:  $\text{fronte} = \text{fine} = 0$ .
- Verifica se una coda è vuota:  $\text{fronte} == \text{fine} ?$ .



**Figura 18.4**  
Una coda vuota.

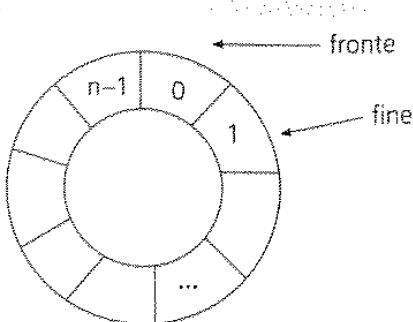


Figura 18.5

Una coda che contiene un elemento.

- Verifica se una coda è piena:  $(\text{fine} + 1) \% \text{MaxDimQ} == \text{fronte}$  ?
- Aggiunge un elemento alla coda: se la coda non è piena, aggiungere un elemento nella posizione fine e mettere  $\text{fine} = (\text{fine} + 1) \% \text{MaxDimQ}$ .
- Elimina un elemento da una coda: se la coda non è vuota, eliminarlo dalla posizione fronte e mettere  $\text{fronte} = (\text{fronte} + 1) \% \text{MaxDimQ}$ .

Prima di cominciare il processo di inserimento, fine punta la seguente posizione disponibile nel vettore. Il nuovo elemento si colloca nella sua posizione e cont si incrementa di 1.

```
VettoreQ[fine] = elemento;
cont++;
```

l'indice fine deve essere aggiornato tramite l'operatore resto (%).

```
void Coda::InserireQ (const Tipo_elem& elemento) {
    if (conto == MaxDimQ)
    {
        cerr << "Queue Overflow" << endl;
        exit(1);
    }
    cont++;
    vettoreQ[fine] = elemento;
    fine = (fine + 1)% MaxDimQ;
}
```

L'operazione RimuovereQ cancella o elimina un elemento dalla testa della coda riferita dall'indice fronte. L'eliminazione inizia copiando il valore dell'elemento da estrarre in una variabile temporale e decrementando conto:

```
elemento = vettoreQ[fronte];
cont--;
```

Nel modello circolare fronte si deve riposizionare nel successivo elemento del vettore utilizzando l'operatore resto (%).

```
fronte = (fronte + 1)% MaxDimQ;
```

Il codice sorgente è:

```

Typo_elem Coda::RimuovereQ (void) {
    Typo_elem aux;
    if (conto == 0)
    [
        cerr << "Queue Underflow" << endl;
        exit(1);
    ]
    aux = vettoreQ[fronte] ;
    cont--;
    fronte = (fronte + 1) % MaxDimQ;
    return aux ;
}

```

#### Esempio 18.4

*Implementazione della classe Coda con un array circolare.*

La classe Coda contiene gli attributi protetti fronte, fine e l'array A dichiarato con una lunghezza massima. Tutte le funzioni membro saranno pubbliche, eccetto il metodo piena che sarà privato perché non serve all'utente della classe. Bisogna tenere in conto che fronte punterà sempre la posizione precedente quella dove si trova il primo elemento della coda e fine punterà sempre la posizione dell'ultimo della coda.

```

#define MaxDimQ 100

typedef int Typo_elem;
class Coda
{
protected:
    int fronte, fine;
    Typo_elem A[MaxDimQ];
    bool piena() { return (fronte == (fine+1) % MaxDimQ); };
    bool vuota() { return (fronte == fine); };
public:
    Coda() : fronte(0), fine(0) {};
    ~Coda() {};
    void Svuota(){ fronte = 0; fine = 0; }
    void Inserisce(Typo_elem e);
    void Cancella();
    Typo_elem Preleva();
};

void Coda::Inserisce(Typo_elem e) {
    if (piena())
    [

```

```

cout << "Queue Overflow";
exit(1);
}
fine = (fine + 1) % MaxDimQ;
A[fine] = e;
}
Tipo_elem Coda::Preleva() {
    if (vuota())
    {
        cout << "Elemento fronte di una coda vuota";
        exit(1);
    }
    fronte = (fronte + 1) % MaxDimQ;
    return (A[(fronte) % MaxDimQ]);
}

void Coda::Cancella() {
    if (vuota())
    {
        cout << "Eliminazione da una coda vuota";
        exit(1);
    }
    fronte = (fronte + 1) % MaxDimQ;
}

int main(int argc, char *argv[])
{
    Coda c;
    c.Svuota();
    c.Inserisce(5); c.Inserisce(6); c.Inserisce(7);
    cout << c.Preleva() << ' ' << c.Preleva() << ' ' << c.Preleva() << endl;
    c.Cancella();
    return 0;
}

```

## Esecuzione

```

5 6 7
Eliminazione da una coda vuota

```

### 13.2.2 La classe coda implementata con una lista concatenata

Se si implementa la coda utilizzando variabili dinamiche, la memoria utilizzata è limitata al numero di elementi attualmente in coda, ma se ne consuma un po' di più per la concatenazione tra gli elementi della coda. Si utilizzano due puntatori per accedere alla coda, fronte e fine, che sono gli estremi da dove gli elementi escono e si inseriscono rispettivamente.

**CONCETTO**

In questo capitolo abbiamo introdotto le più importanti strutture dati dinamiche, le pile e le code.

Una *pila* è una struttura dati gestita con strategia LIFO (*Last In First Out*, ultimo a entrare, primo a uscire) nella quale i dati si inseriscono e rimuovono dallo stesso estremo, che si denuncia *cima* della pila. Le più importanti operazioni su una pila sono: Push (aggiunge un elemento nella cima della pila) e Pop (elimina un elemento dalla cima della pila). Ogni operazione di estrazione di dati

deve assicurarsi previamente che vi sia almeno un elemento della pila, altrimenti la pila finirà in *underflow* e si produrrà un errore.

Una *coda* è una struttura dati gestita con strategia FIFO (*First In First Out*, primo a entrare, primo a uscire) nella quale i dati si inseriscono in *coda* e si eliminano dalla *testa*. Le operazioni base in una coda sono: inserire, rimuovere, fronteCoda e fineCoda.

Pile e Code si possono implementare sia utilizzando liste dinamiche sia array.

**CONCETTO**

- Classe Coda
- Classe Pila
- Coda

- FIFO
- LIFO
- Pila

## Esercizi

**Esercizio 1** Descrivere due possibili applicazioni delle pile in programmi di computer.

**Esercizio 2** Qual è l'output di questo frammento di codice (tipo di dato = int):

```
Pila p;
int x = 5, y = 3;
p.Inserire(8);
p.Inserire(9);
p.Inserire(y);
x = p.Rimuovere();
p.Inserire(18);
x = p.Rimuovere();
p.Inserire(22);
while (!p.Pilavuota())
{
    y = p.Rimuovere();
    cout << y << endl;
}
cout << x << endl;
```

**Esercizio 3** Scrivere i seguenti algoritmi di una pila per dati di tipo intero:

- |              |                         |
|--------------|-------------------------|
| • creare     | • pila_piena            |
| • inserire   | • cima_pila             |
| • rimuovere  | • lunghezza_pila        |
| • pila_vuota | • liberare_memoria_pila |

**Esercizio 4** Siano P1 e P2 due pile vuote di interi. Disegnare uno schema che rappresenti le pile dopo le seguenti operazioni:

```
P1. Inserire (3);
P1. Inserire (5);
P2. Inserire (7);
P1. Inserire (9);
P2. Inserire (11);
P2. Inserire (13);
while (!P1.pilavuota ())
{
    P1. rimuovere (x);
    P2. rimuovere (x);
}
```

**Esercizio 1** Scrivere una funzione

```
void CancellarePila (Pila& P);
```

che pulisca (cancelli) una pila. Perché è importante che P si passi per riferimento?

**Esercizio 2** Descrivere due applicazioni delle code in un programma informatico.

**Esercizio 3** Leggere 10 interi di un array e metterli in una pila. Stampare il vettore originale e, poi, stampare la pila estraendo gli elementi.

**Esercizio 4** Scrivere un programma che permetta di determinare se una parola o frase è un palindromo. *Nota:* Una parola è palindroma se la lettura in entrambi i sensi produce lo stesso risultato; esempio: natan. Nel caso di una frase si omettono gli spazi. Esempio (in spagnolo): "Dabale arroz a la zorra el abad".

**Esercizio 5** Scrivere una funzione copiarePila che copi il contenuto di una pila in un'altra. La funzione deve avere due argomenti di tipo pila, uno per la pila sorgente e un altro per la pila destinazione.

**Esercizio 6** Scrivere un programma che inverta il contenuto di una pila.

**Esercizio 7** Scrivere una funzione che verifichi se i contenuti di due pile sono identici.

**Esercizio 8** Scrivere un programma che crei una pila a partire da una coda.

**Esercizio 9** Scrivere un programma che comprima una stringa di caratteri, cancellando tutti i caratteri spazio in bianco.

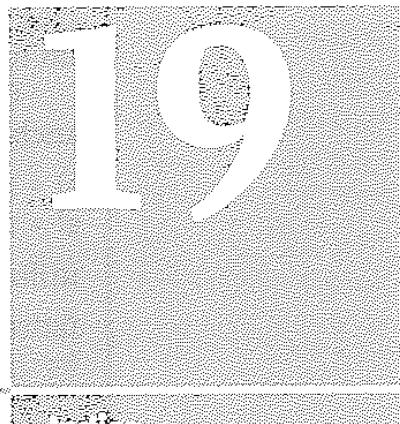
**Esercizio 10** Data una coda di interi, scrivere un programma che elimini tutti gli interi negativi senza cambiare gli altri elementi della coda.

**Esercizio 11** Scrivere un programma che inverta il contenuto di una coda.

**Esercizio 12** Scrivere un programma che verifichi i contenuti di due code e restituisca vero se sono identiche e falso in caso contrario.

**Esercizio 13** Scrivere un codice per creare una lista concatenata che descriva una coda di tre macchine. Per ogni macchina si debbono immagazzinare numero e anno di immatricolazione, marca e modello.

# Ricorsione



**19.1** Funzioni ricorsive

**19.2** Confronto fra ricorsione e iterazione

**19.3** Soluzione di problemi attraverso la ricorsione

**19.4** *QuickSort*

## Introduzione

In questo capitolo introdurremo il concetto di "ricorsione". Una funzione si dice "ricorsiva" se chiama sé stessa. Si può utilizzare la ricorsione come alternativa all'iterazione, ma normalmente essa è meno efficiente

a causa delle operazioni ausiliarie compiute per chiamare la funzione. Ciò nonostante, la ricorsione viene usata perché spesso semplifica la concezione degli algoritmi risolutivi dei problemi.

### 19.1 Funzioni ricorsive

Per la soluzione di alcuni problemi è comodo consentire a una funzione di chiamare sé stessa. Quando una funzione chiama sé stessa, sia direttamente sia per il tramite di altre funzioni, essa viene detta *ricorsiva*. La ricorsione è stata un argomento importante e molto studiato nella storia della programmazione.

In matematica esistono numerose funzioni che hanno un intrinseco carattere ricorsivo.

#### Esempio 19.1

Il fattoriale di un intero negativo  $n$ , scritto  $n!$ , è il prodotto:

$$n! = n * (n - 1) * (n - 2) * \dots * 1$$

per esempio:

$$5! = 5 * 4 * 3 * 2 * 1 = 5 * 4! = 120$$
$$(5-1)!$$

cosicché il fattoriale di  $n$  è:

$$n! = n * (n - 1)!$$

cioè la funzione ha un'intrinseca natura ricorsiva (si calcola il fattoriale di un numero calcolando il fattoriale di un altro numero).

Ovviamente il fattoriale di un numero intero  $n \geq 0$  si può calcolare anche in maniera iterativa tenendo presente che:

$$\begin{aligned} n! &= 1 && \text{se } n = 0 \\ n! &= n * (n - 1)! && \text{se } n > 0 \end{aligned}$$

utilizzando per esempio un semplicissimo ciclo for:

```
fattoriale = 1;
for (int contatore = n; contatore >= 1; contatore --)
    fattoriale *= contatore;
```

L'algoritmo che implementa la funzione in maniera ricorsiva deve contemplare una condizione di uscita, la condizione cioè che, se verificata, interromperà la sequenza delle chiamate ricorsive. Così come la condizione d'uscita di un ciclo iterativo, anche la condizione d'uscita di una ricorsione deve essere prima o poi verificata, pena la non terminazione della ricorsione. Per esempio, nel caso del fattoriale, la condizione d'uscita sarà il raggiungimento del passo in cui si calcola il fattoriale del numero 0, cioè  $0! = 1$ . L'algoritmo ricorsivo sarà:

$$\begin{aligned} n! &= 1 && \text{se } n = 0 \text{ (condizione di terminazione)} \\ n! &= n * (n - 1)! && \text{se } n > 0 \text{ (passo della ricorsione)} \end{aligned}$$

L'implementazione ricorsiva della funzione fattoriale sarà:

```
int fattoriale(int n)
{
    if (n == 0) return 1;           // termine della ricorsione
    else return n * fattoriale(n - 1); // passo della ricorsione
}
```

#### Esempio 19.2

Definizione ricorsiva del prodotto di numeri naturali.

1. *Soluzione iterativa*  $a * b = a + a + a + \dots + a$   
 $b$  volte
2. *Soluzione ricorsiva*  $a * b = a$  se  $b = 1$   
 $a * b = a * (b - 1) + a$  se  $b > 1$

#### Esempio 19.3

Implementare la serie di **Fibonacci**: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

La serie di Fibonacci inizia con 0 e 1, e ogni successivo è la somma dei precedenti due:

$$\begin{aligned} 0 + 1 &= 1 \\ 1 + 1 &= 2 \\ 2 + 1 &= 3 \\ 3 + 2 &= 5 \\ 5 + 3 &= 8 \\ \dots \end{aligned}$$

quindi le condizioni di terminazione e il passo saranno:

```
fibonacci(0) = 0
fibonacci(1) = 1
...
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)
```

cioè:

fibonacci(n) = n	<i>se n = 0 oppure n = 1</i>
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)	<i>se n &gt; = 2</i>

Si osservi che la definizione ricorsiva della serie di Fibonacci si differenzia dai due esempi precedenti perché il passo richiede due chiamate ricorsive invece che una. La serie potrebbe essere generata anche da un algoritmo iterativo:

```
int fibonacci(int n)
{ if (n <= 1) return(n);
  fibinf = 0;
  fibsup = 1;
  for (int i = 2; i <= n; i++){
    x = fibinf;
    fibinf = fibsup;
    fibsup = x + fibinf;
  }
  return (fibsup);
}
```

L'equivalente ricorsivo, però, sembra più intuitivo.

#### Esempio 13.4

```
unsigned long fibonacci(unsigned n) {
  if (n == 0 || n == 1) return n;
  else return fibonacci(n - 1) + fibonacci(n - 2);
}

int main()
{
  unsigned num;
  cout << "Introduca un intero : ";
  cin >> num;
  cout << "Fibonacci(" << num << ") = " << fibonacci(num) << endl;
  return 0 ;
}
```

#### Esecuzione di prova

Introduca un intero : 22

Fibonacci(22) = 17711

La ricorsione indiretta avviene quando la funzione chiama sé stessa non direttamente ma tramite una concatenazione di chiamate ad altre funzioni che termina con una chiamata alla prima.

#### Esempio 19.5

```
void A(char);
void B(char);

void A(char c) {
    if (c > 'A') B(c);
    putchar(c);
}

void B(char c) {
    A(--c);
}

int main()
{
    A('Z');
    cout << endl;
    return 0;
}
```

#### Esecuzione

ABCDEFIGHJKLMNOPQRSTUVWXYZ

Il programma principale chiama la funzione ricorsiva `A()` con argomento '`Z`'. La funzione `A` esamina il suo parametro `c`, dopodiché se il valore in esso contenuto è lessicograficamente maggiore di '`A`' essa chiama la funzione `B()`, che, a sua volta, chiama di ritorno `A()` passandole come argomento il valore ASCII immediatamente precedente quello contenuto nel parametro `c`. Ciò farà sì che `A()` torni a esaminare l'argomento `c` chiamando, eventualmente, ancora `B()` fino a quando `c` conterrà il valore '`A`'. A questo punto la ricorsione terminerà con la chiamata `putchar()` che sarà eseguita ventisei volte visualizzando tutto l'alfabeto ASCII, carattere per carattere.

## 19.2 Confronto fra ricorsione e iterazione

Sia la ricorsione sia l'iterazione implicano la ripetizione e si basano su una struttura di controllo, un test che se verificato terminerà la ripetizione. Lo svantaggio della ricorsione sta nel fatto che a ogni ripetizione essa invoca una o più funzioni (sé stessa se la ricorsione è diretta), ed è quindi duplice perché implica ritardi e maggiore occupazione di memoria. Lo svantaggio in termini di tempo di computazione sta nel fatto che la chiamata di una funzione richiede essa stessa un tempo d'esecuzione, indipendentemente da cosa farà poi la funzione, mentre lo svantaggio in termini di occupazione di memoria sta nel fatto che a ogni chiamata di funzione bisogna memorizzare nello

"stack" una serie di registri e parametri, i più importanti dei quali sono il registro del processore che contiene l'indirizzo dell'istruzione che dovrà essere eseguita quando la funzione terminerà la sua esecuzione, gli argomenti della funzione stessa e le sue variabili locali. Poiché questi spazi di memoria nello stack saranno rilasciati solo al termine della funzione, una ricorsione di chiamate provoca una crescita continua dello stack che comincerà a svuotarsi come un "pila", cioè in ordine inverso rispetto a quello di riempimento, solo raggiungimento della condizione di terminazione della ricorsione.

Qualunque problema che si può risolvere ricorsivamente lo si può risolvere anche mediante un algoritmo iterativo; cioè per ogni funzione ricorsiva se ne può trovare un'altra che fa la stessa cosa attraverso un ciclo e senza richiamare sé stessa. La ricorsione spesso produce soluzioni semplici e più facili a un'immediata comprensione, ma la corrispondente soluzione iterativa sarà normalmente più efficiente, sia in termini di occupazione di spazio di memoria sia in termini di tempo di computazione.

L'unico vantaggio della ricorsione sta nel fatto che alcuni problemi e funzioni hanno un'implementazione ricorsiva intuitiva e naturale, cosa che semplifica il lavoro del programmatore, sia nella scrittura del programma sia nella sua "manutenzione".

In conclusione possiamo consigliare la soluzione ricorsiva solo quando la soluzione iterativa è difficile da trovare o l'applicazione non richiede particolari prestazioni in termini di tempo e di occupazione di memoria.

#### Esempio 19.6

Per confronto, proponiamo la versione iterativa della funzione "fattoriale".

```
int fattoriale(int n) {
    int fatt = 1;
    for (int i = 2; i <= n; i++) fatt *= i;
    return fatt;
}
```

Se l'iterazione sia la ricorsione possono non terminare mai, è sufficiente che la condizione di controllo non venga mai falsificata.

#### Esempio 19.7

Si calcoli la somma dei primi  $n$  interi positivi. Una funzione iterativa è:

```
int Sommatoria(int n);
{
    int somma = 0;
    for (int i = 1; i <= n; i = i + 1)
        somma = somma + i;
    return somma;
}
```

La versione ricorsiva della funzione Sommatoria si basa sul fatto che:

$$\text{Sommatoria}(N) = \begin{cases} 1 & \text{se } N = 1 \\ N + \text{Sommatoria}(N-1) & \text{altrimenti} \end{cases}$$

quindi tale versione ricorsiva sarà:

```
int Sommatoria(int n)
{
    if (n == 1) return 1; // condizione di uscita, tappo della ricorsione
    else return n + Sommatoria(n - 1); // passo della ricorsione
}
```

#### Esempio 19.8

*Scrivere una funzione mcd() che calcoli il Massimo Comune Divisore di due numeri interi e un programma che ne faccia uso.*

Il Massimo Comune Divisore di due numeri naturali è il maggiore intero che li divide entrambi. L'algoritmo  $mcd(m, n)$  deve essere tale per cui:

- $mcd(m, n)$  sia  $n$  se  $n \leq m$  e  $n$  è divisore di  $m$ ;
- $mcd(m, n)$  sia  $mcd(n, m)$  se  $m < n$ ;
- $mcd(m, n)$  sia  $mcd(n, resto di m diviso per n)$  altrimenti.

Vale a dire che il MCD è già  $n$  se  $n$  è un divisore di  $m$ ; se  $m$  è più piccolo di  $n$  allora bisogna invertire gli argomenti, altrimenti il MCD si ottiene calcolando il MCD di  $n$  e del resto della divisione intera di  $m$  per  $n$ . L'algoritmo ricorsivo che calcola il MCD potrebbe essere quindi:

```
int mcd(int a, int b)
{
    if (b == 0) return a; // tappo
    else return mcd(b, a%b); // passo
}
```

Un'altra implementazione ricorsiva è la seguente:

```
int mcd(int m, int n) {
    if (n <= m && m%n==0) return n;
    else if (m < n) return mcd(n, m);
    else return mcd(n, m%n);
}
```

#### Esempio 19.9

*Scrivere una funzione ricorsiva che calcoli la somma dei quadrati dei primi  $N$  numeri interi.*

La funzione Sommaquadrati implementata ricorsivamente seguirebbe la seguente definizione:

$$\text{Sommatoria}(N) = \begin{cases} 1 & \text{se } N = 1 \\ N^2 + \text{Somma}(N-1) & \text{altrimenti} \end{cases}$$

```
int Sommaquadrati(int n)
{
    if (n == 1) return 1; // tappo
    else return n*n + Sommaquadrati(n - 1); // passo
}
```

Attenzione! Cosa succede se la funzione viene chiamata passandogli il valore 0 o un numero negativo?

#### Esempio 19.10

*Scrivere una funzione ricorsiva che calcoli il prodotto di due numeri naturali utilizzando l'operazione somma.*

Il prodotto di due numeri naturali  $a$  e  $b$  si può definire ricorsivamente così:

Prodotto( $a, b$ ) = 0 se  $b=0$ ;  
Prodotto( $a, b$ ) =  $a + \text{Prodotto}(a, b-1)$  se  $b>0$ .

Anche questa funzione, come quella dell'esempio precedente, produrrà una ricorsione infinita se chiama con un numero negativo al secondo membro.

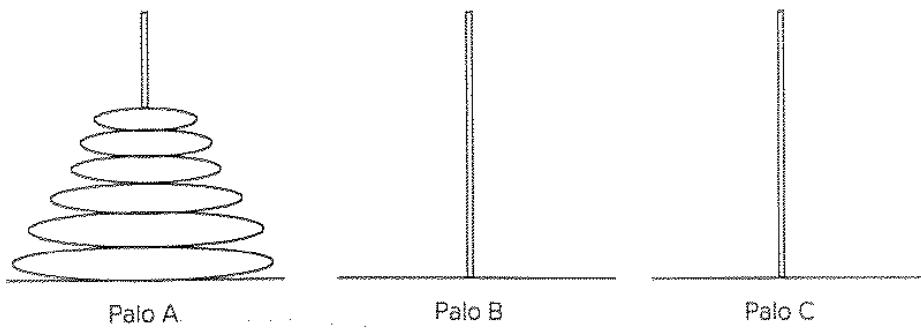
```
int Prodotto(int a, int b)
{
    if (b == 0) return 0;
    else return a + Prodotto(a, b - 1);
}
```

### 19.3 Soluzione di problemi attraverso la ricorsione

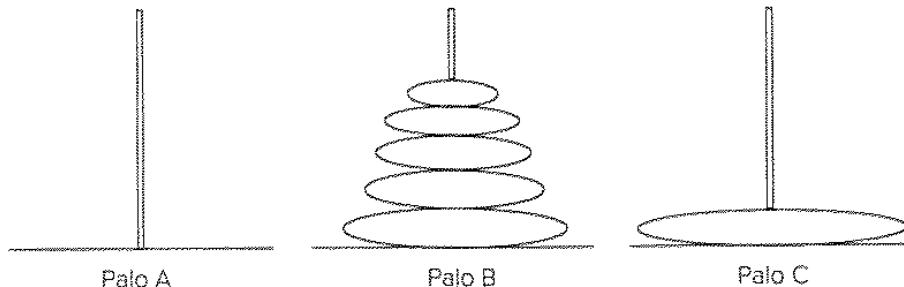
In questa sezione vediamo alcuni classici esempi di problemi che si risolvono in maniera semplice ed elegante tramite la ricorsione.

#### 19.3.1 Torre di Hanoi

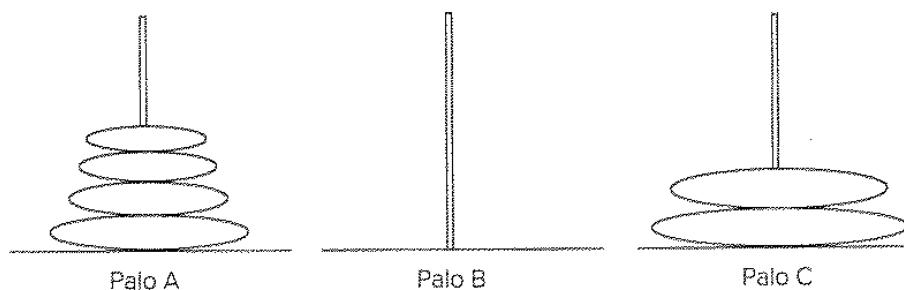
Si tratta di un gioco orientale ispirato dalla struttura del tempio di Brahma. Ci sono tre pali, A, B e C, su cui sono impilati  $n$  dischi di diverso diametro. I dischi si possono spostare da un palo all'altro con la regola che ogni disco impilato deve essere di diametro immediatamente inferiore a quello che gli sta sotto.



Illustriamo il problema con tre pali che contengono sei dischi sul palo A, che devono essere trasferiti al palo C. Così, per esempio, si possono spostare cinque dischi in una volta dal palo A al palo B, e il disco più grande al palo C.



Ora il problema è quello di passare i cinque dischi dal palo B al palo C e si utilizza un metodo simile al precedente, passare i quattro dischi superiori dal palo B al palo A e il disco maggiore dal palo B al palo C. Il processo continua in modo ricorsivo fin quando, finalmente, non rimane alcun disco nel palo B, che è la condizione di terminazione.



Questo problema è chiaramente ricorsivo.

#### Progetto dell'algoritmo

L'esempio precedente dei sei dischi si può generalizzare a un algoritmo con  $n$  dischi e tre pali. La funzione di Hanoi dichiara i tre pali come oggetti di tipo char. Nella lista dei parametri, l'ordine delle variabili è:

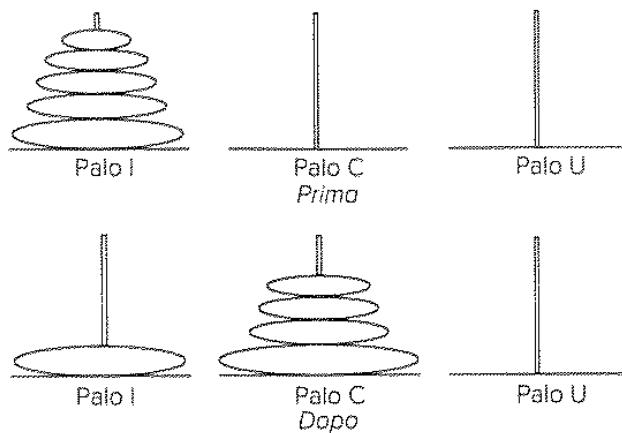
paloiniziale      palocentrale      palofinale

il che implica che si stanno muovendo dischi dal palo iniziale al finale utilizzando il palo centrale come ausilio. Se  $n = 1$  si ha la condizione di terminazione, giacché si può spostare l'unico disco dal palo iniziale al palo finale. L'algoritmo sarà il seguente:

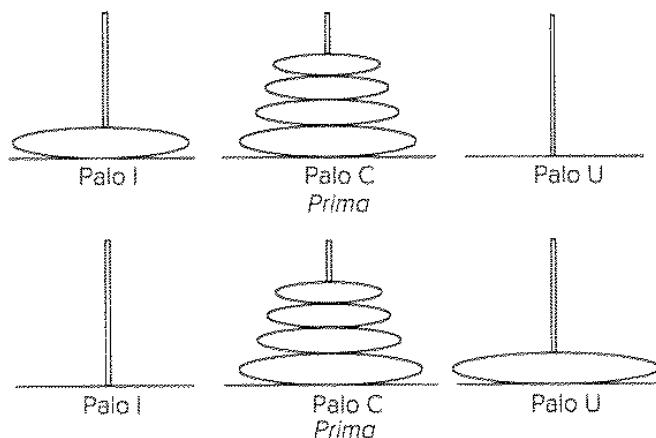
1. **Se  $n$  è 1**
  - 1.1 Muovere il disco 1 da paloiniziale a palofinale.
2. **Sennò**
  - 1.2 Muovere  $n-1$  dischi da paloiniziale fino al palocentrale attraverso palofinale.
  - 1.3 Muovere il disco  $n$  da paloiniziale a palofinale.
  - 1.4 Muovere  $n-1$  dischi da palocentrale a palofinale attraverso paloiniziale.

Vale a dire, se  $n$  è 1, si raggiunge la condizione di terminazione dell'algoritmo. Se  $n$  è maggiore di 1, le tappe ricorsive 1.2, 1.3 e 1.4 sono tre problemi più piccoli, uno dei quali è la condizione di uscita.

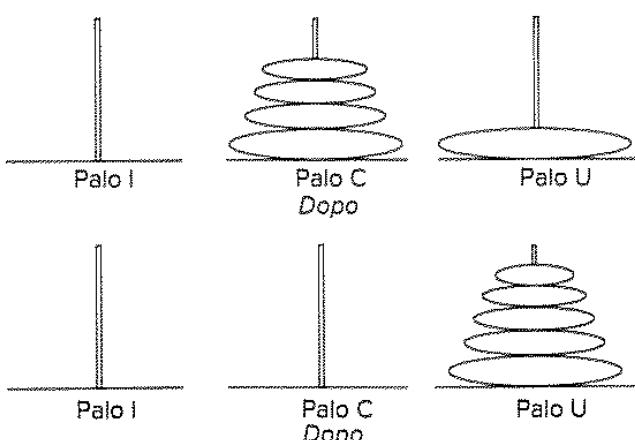
*Tappa 1:* muovere  $n - 1$  dischi dal palo iniziale (I).



*Tappa 2:* muovere un disco da I a U.



*Tappa 3:* muovere  $n - 1$  dischi da C a U.



La prima tappa dell'algoritmo muove  $n - 1$  dischi dal palo iniziale a quello centrale utilizzando il palo finale. Di conseguenza, l'ordine dei parametri nella chiamata alla funzione ricorsiva è *paloiniziale*, *palofinale* e *palocentrale*.

```
// utilizza palofinale come appoggio ausiliario
Hanoi(n - 1, paloiniziale, palofinale, palocentrale);
```

La seconda tappa muove semplicemente il disco maggiore dal palo iniziale al palo finale:

```
cout << "muove " << palocentrale << "a " << palofinale << endl;
```

La terza tappa dell'algoritmo muove  $n - 1$  dischi dal palo centrale a quello finale utilizzando il palo iniziale come appoggio temporaneo. Di conseguenza, l'ordine dei parametri nella chiamata alla funzione ricorsiva è: palocentrale, paloiniziale e palofinale.

```
// utilizza paloiniziale come appoggio ausiliario
Hanoi( n- 1, palocentrale, paloiniziale, palofinale);
```

#### Implementazione della torre Hanoi

L'implementazione dell'algoritmo si basa sui nomi dei tre pali passati come parametri alla funzione. Il programma comincia chiedendo all'utente che introduca il numero  $N$  dei dischi. Si chiama la funzione ricorsiva Hanoi per ottenere una serie di movimenti che trasferirà gli  $N$  dischi da paloiniziale a palofinale. L'algoritmo richiede  $2^N - 1$  movimenti.

```
void Hanoi(char paloiniziale, char palofinale, char palocentrale, int n)
{
    if ( n == 1)
        cout << "Muove disco 1 dal palo " << paloiniziale
            << " al palo " << palofinale << endl;
    else
    [
        Hanoi(paloiniziale, palocentrale, palofinale, n - 1);
        cout << " Muove disco" << n << " dal palo " <<
            << paloiniziale << " al palo " << palofinale << endl;
        Hanoi(palocentrale, palofinale, paloiniziale, n - 1);
    ]
}
```

Per muovere tre dischi dal palo A al palo C mediante il palo B si può chiamare la funzione così:

```
Hanoi('A', 'C', 'B', 3);
```

che stamperà in output:

```
Muove disco 1 dal palo A al palo C
Muove disco 2 dal palo A al palo B
Muove disco 1 dal palo C al palo B
Muove disco 3 dal palo A al palo C
Muove disco 1 dal palo B al palo A
Muove disco 2 dal palo B al palo C
Muove disco 1 dal palo A al palo C
```

### Considerazioni sull'efficienza della funzione Hanoi

La funzione Hanoi risolve il problema della torre per qualunque numero di dischi. Con tre dischi il problema si risolve con 7 (cioè  $2^3 - 1$ ) chiamate alla funzione Hanoi. In generale, come abbiamo detto,  $n$  dischi richiedono  $2^n - 1$  chiamate. Ogni chiamata di funzione richiede l'allocazione di spazio di memoria nello stack, quindi l'occupazione di memoria è esponenziale con la dimensione del problema. Provate a stimare il massimo numero di dischi che sarà in grado di gestire questa funzione sul vostro computer.

#### 10.3.2 Ricerca binaria ricorsiva

Anche la ricerca binaria di un elemento all'interno di un array ordinato rispetto a un campo chiave può essere svolta da un algoritmo ricorsivo. Data la chiave cercata, si comincia la ricerca dalla posizione centrale del vettore. Se essa contiene l'elemento cercato la ricerca finisce, altrimenti si richiama la funzione di ricerca per applicarla al sottovettore inferiore (dal primo elemento a quello centrale) o a quello superiore (dall'elemento centrale all'ultimo), a seconda del risultato del confronto fra la chiave cercata e l'elemento in posizione centrale. Il procedimento continua in maniera ricorsiva.

1. Se chiave < A[centrale], l'elemento cercato non può che trovarsi eventualmente nel sottovettore che va dall'estremo inferiore all'elemento central - 1.
2. Se chiave > A[centrale], l'elemento cercato non può che trovarsi eventualmente nel sottovettore superiore, dall'elemento central + 1 all'estremo superiore.
3. Il procedimento ricorsivo ricerca in sottovettori ogni volta dimezzati fino a uno dei due eventi: successo se l'elemento centrale è quello cercato, fallimento se il limite superiore del vettore diviene più piccolo del limite inferiore, cioè inferiore > superiore (e l'algoritmo restituirà convenzionalmente -1).

RicercaBinaria(inferiore, superiore, chiave) // ricerca binaria ricorsiva

```
{ restituisci -1 if inferiore > superiore // elemento non trovato
  { restituisci centrale if A[centrale] == chiave // elemento trovato
    restituisci RicercaBinaria(centrale + 1, superiore, chiave) if A[centrale] < chiave
    restituisci RicercaBinaria(inferiore, centrale - 1, chiave) if A[centrale] > chiave
```

La funzione C++ sarà:

```
int RicercaBinaria(int inferiore, int superiore, int chiave, int A[])
{
int i;
if (inferiore > superiore) return -1      // elemento non trovato
else
{ i = (inferiore + superiore)/2;
  if (A[i]==chiave) return i;           // elemento trovato
  else if (A[i]<chiave)
    return RicercaBinaria(i+1, superiore, chiave);
  else return RicercaBinaria(inferiore, i-1, chiave);
}
```

## 19.4 QuickSort

Il metodo di ordinamento *QuickSort* fu scoperto da Tony Hoare nel 1962 ed è il più efficiente fra gli algoritmi noti. Si basa sulla divisione del vettore di  $n$  elementi in tre partizioni: la *sinistra*, la *centrale* che contiene un solo elemento denominato *pivot* e la *destra*. Supponendo di dover ordinare in senso crescente, tutti gli elementi della partizione sinistra dovranno essere minori del più piccolo elemento della parte destra, e questo risultato sarà ottenuto confrontando gli elementi con il pivot. Applicando ricorsivamente l'algoritmo sia alla partizione sinistra sia a quella destra si arriverà a ordinare tutto il vettore. L'elemento pivot può essere scelto a caso all'interno del vettore. Statisticamente sarebbe buona cosa che esso fosse la metà della somma dei valori minimo e massimo all'interno del vettore. Per esempio, se il vettore contiene elementi che vanno da 1 a 10, allora si potrebbe prendere come pivot un elemento che ha valore 5 oppure 6. Scelto il pivot, esso sarà utilizzato per generare le due partizioni; la sinistra con gli elementi minori del pivot, la destra con gli elementi maggiori. Dopodiché si applicherà l'algoritmo ricorsivamente alle due partizioni fino a che la partizione da ordinare contiene un solo elemento.

### Esempio 19.11

(Pivot primo elemento del vettore)

1. Vettore iniziale

5	2	1	9	3	8	7
---	---	---	---	---	---	---

Pivot scelto

5
---

Partizione sinistra (elementi minori di 5)

2	1	3
---	---	---

Partizione destra (elementi maggiori di 5)

9	8	7
---	---	---

2. Partizione sinistra

2      1      3

↑  
pivot

partizione sinistra 1  
partizione destra 3

Partizione sinistra

sinistra      pivot      destra

1	2	3
---	---	---

3. Partizione destra

9      8      7

↑  
pivot

partizione sinistra 8    7  
partizione destra

Partizione destra

sinistra      pivot      destra

7	8	9
---	---	---

4. Vettore finale ordinato

partizione sinistra

1      2      3

pivot

5

partizione destra

7      8      9

L'efficienza del *QuickSort* è legata alla scelta del pivot. Scelto il pivot si passa alla seconda fase che è la generazione delle due partizioni. Per questo si scambia subito il pivot con l'ultimo elemento.

#### Esempio 19.12

Vettore iniziale: 8 1 4 9 6 3 5 2 7 0

Pivot (elemento centrale) 6

Scambio del pivot con l'ultimo elemento 0

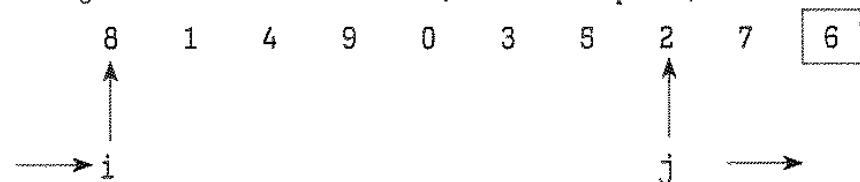
Lo scambio produce il vettore:

8 1 4 9 0 3 5 2 7 6

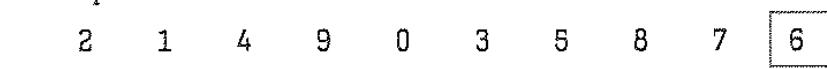
Per generare le due partizioni:

1. si scorre il vettore da sinistra a destra con un contatore  $i$  che viene inizializzato all'indice minore (inferiore) cercando un elemento maggiore del pivot;
2. si scorre il vettore da destra verso sinistra con un altro contatore  $j$  che viene inizializzato alla posizione più alta (superiore-1) cercando un elemento minore del pivot.

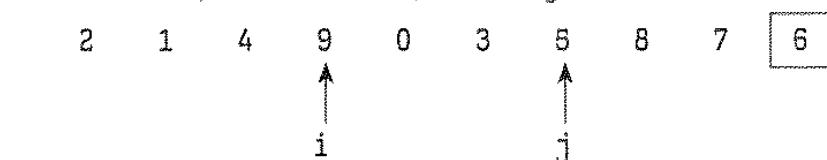
Il contatore  $i$  seleziona l'elemento 8 (maggiore del pivot), mentre il contatore  $j$  si ferma all'elemento 2 (minore del pivot).



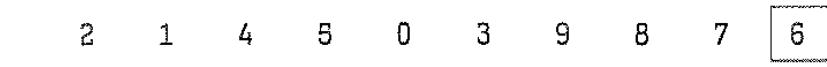
A questo punto si scambiano 8 e 2 perché vadano a far parte delle corrette partizioni.



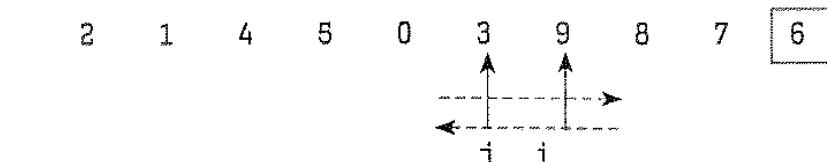
Continuando,  $i$  si ferma al 9, mentre  $j$  si ferma al 5.



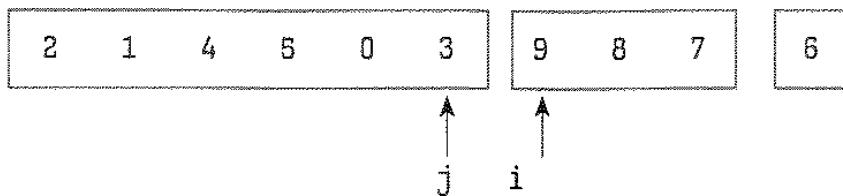
Le coppie continuano a essere scambiate fintanto che  $i$  e  $j$  non si incrociano. Nel nostro caso si scambiano 9 e 5.



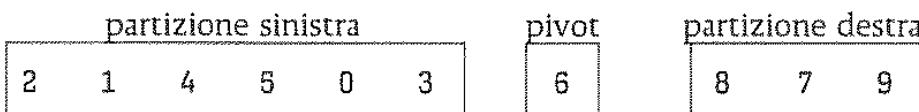
Continuando,  $i$  arriva all'elemento 9 e  $j$  scende fino al 3.



Ora i e j si incrociano e non si effettua alcuno scambio perché i due elementi stanno già dalla parte corretta. Il vettore originale è stato diviso in due partizioni, la sinistra e la destra.



A questo punto si scambia l'elemento in posizione i con il pivot che sta all'ultimo posto, in maniera da ritornare alla sequenza iniziale.



#### 19.4.1 Il QuickSort in C++

Il primo compito è selezionare il pivot. Poiché cercare il valore mediano è troppo dispendioso, si prende l'elemento centrale del vettore o uno dei suoi adiacenti (Figura 19.1).

Riassumiamo l'algoritmo *quicksort*:

*Selezionare l'elemento centrale del vettore a[0:n-1] come pivot.*

*Dividere gli elementi restanti in partizione sinistra (elementi minori del pivot) e destra (elementi maggiori o uguali al pivot).*

*Ordinare la partizione sinistra utilizzando quicksort ricorsivamente.*

*Ordinare la partizione destra utilizzando quicksort ricorsivamente.*

*La soluzione è il concatenamento della partizione sinistra, con il pivot e con la partizione destra.*

Per implementare quicksort utilizziamo un template di funzione. Supponiamo di dover ordinare il vettore in senso crescente.

#### Esempio 19.10

```
template <typename T> void sort(T v[], int n);
template <typename T> void quicksort(T v[], int s, int d) ;
template <typename T> void sort(T v[], int n) {
    quicksort(v, 0, n-1);
}

template <typename T> void quicksort(T v[], int s, int d) {
    int i = s;
    int j = d;
    T aux, pivot = v[(s + d)/2];
    while (i <= j) {
        while (v[i] < pivot) i++;
        while (v[j] > pivot) j--;
        if (i < j) {
            aux = v[i];
            v[i] = v[j];
            v[j] = aux;
            i++;
            j--;
        }
    }
}
```

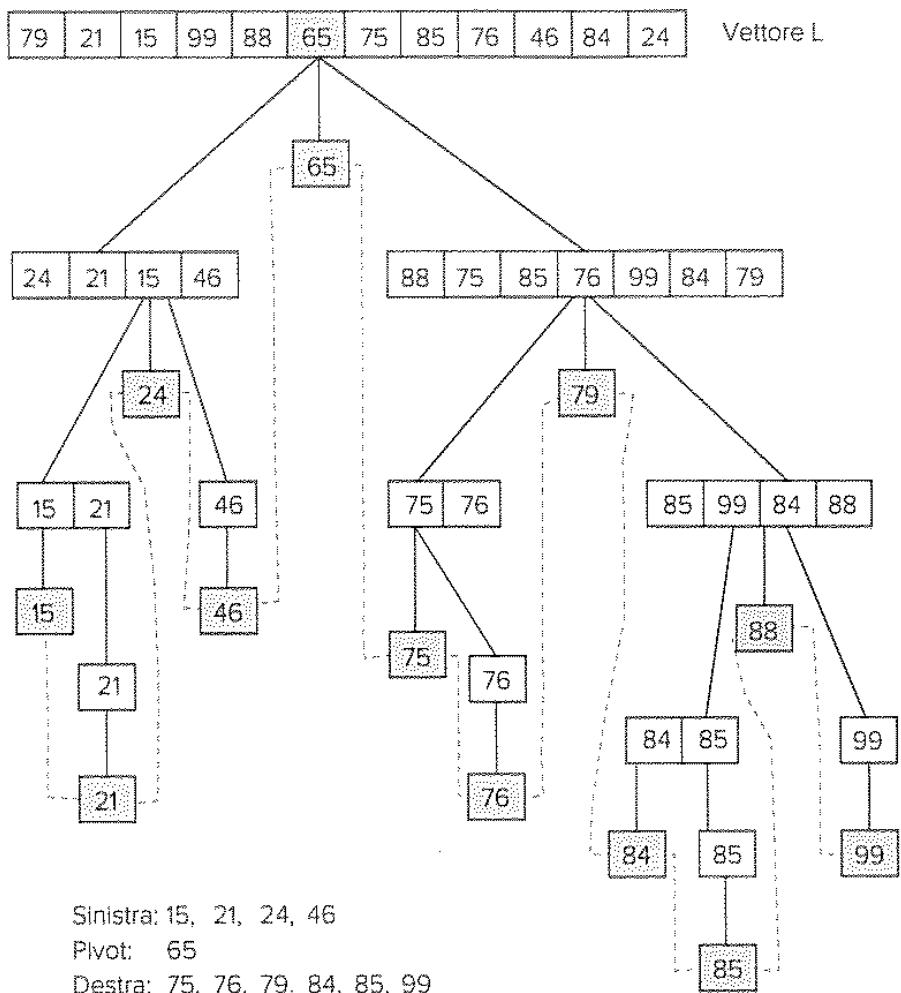


Figura 19.1 Algoritmo di ordinamento quicksort.

```

if (i <= j) {
    aux = v[i];
    v[i] = v[j];
    v[j] = aux;
    i++;
    j--;
}
if (s < j) quicksort(v, s, j);
if (i < d) quicksort(v, i, d);
}

int main() {
    double vettore[7] = {3.14, 3.02, 4.12, 5.34, 2.13, 2.31, 3.24};
    sort(vettore, 7);
}

```

```

    for (int i=0; i<7; i++) cout << vettore[i] << " ";
    return 0;
}

```

### Esecuzione

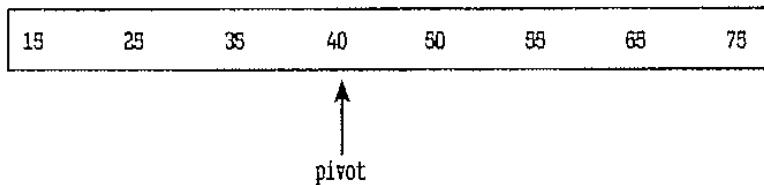
```
2.13 2.31 3.02 3.14 3.24 4.12 5.34
```

#### 19.4.2 Analisi del QuickSort

L'analisi completa dell'efficienza del *QuickSort* è difficile e non rientra negli scopi di un testo introduttivo sui fondamenti della programmazione in C++. Un modo intuitivo per illustrare la complessità dell'algoritmo è quello di considerare il numero di confronti compiuti in circostanze ideali. Supponiamo che il numero di elementi del vettore sia una potenza di due,  $n = 2^k$ , quindi  $k = \log_2 n$ . Se il pivot è l'elemento centrale del vettore, *QuickSort* divide l'array in due partizioni della stessa lunghezza, o quasi. La prima scansione opera  $n - 1$  confronti e genera due partizioni lunghe pressappoco  $n/2$ . Nel passo successivo il processamento di ogni partizione richiede approssimativamente  $n/2$  confronti, e quindi i confronti totali di questo passo sono  $2(n/2) = n$ . Il passo successivo richiede il processamento di quattro partizioni dimezzate compiendo quindi  $4(n/4)$  confronti e così via. L'ordinamento terminerà dopo  $k$  passi, quando le partizioni saranno lunghe 1. Il numero totale di confronti sarà approssimativamente:

$$n + 2(n/2) + 4(n/4) + \dots + n(n/n) = n + n + \dots + n = n * k = n * \log_2 n$$

e quindi la complessità stimata del *QuickSort* è  $O(n \log_2 n)$ . Questo caso ideale corrisponde a un vettore già ordinato, perché il pivot sarà proprio nel centro di ogni partizione.



Il caso peggiore è quando il pivot è sempre l'elemento più piccolo della sua partizione. Al primo passo si compiono  $n$  confronti e la partizione grande contiene  $n - 1$  elementi. Al passo successivo la partizione maggiore richiede  $n - 1$  confronti e produce una sottopartizione di  $n - 2$  elementi ecc. Il numero totale di confronti sarà:

$$n + n - 1 + n - 2 + \dots + = n(n + 1)/2 - 1$$

e la complessità risulta essere  $O(n^2)$ .

Comunque in generale la complessità del *QuickSort* è  $O(n \log_2 n)$  risultando probabilmente il più efficiente algoritmo conosciuto. La Tabella 19.1 confronta la complessità degli algoritmi di ordinamento presentati in questo libro.

In conclusione si suggerisce di utilizzare l'inserimento diretto e la selezione per vettori piccoli, e il *QuickSort* per vettori grandi.

**Tabella 19.1** Confronto sulla complessità degli algoritmi di ordinamento

Metodo	Complessità
BubbleSort	$n^2$
Inserimento	$n^2$
Selezione	$n^2$
Shell	$n \log_2 n$
QuickSort	$n \log_2 n$

**CONSIGLI**

In questo capitolo abbiamo introdotto la *ricorsione*, la capacità cioè di una funzione di richiamare se stessa, direttamente o per il tramite di altre. La ricorsione può essere sempre rimpiazzata da un algoritmo iterativo, e quasi sempre ciò converrebbe in termini di occupazione di memoria e di tempo d'esecuzione. Però essa a volte semplifica e rende eleganti gli algoritmi, facilitandone la comprensione e la manutenzione. L'al-

goritmo ricorsivo deve distinguere due casi (mediante un'opportuna istruzione condizionale *if*): il "passo" della ricorsione e il "tappo", cioè l'istruzione di terminazione; vi possono essere diverse chiamate ricorsive e diversi tappi (quindi l'istruzione condizionale potrebbe diventare uno *switch*), ma l'importante è che l'algoritmo prima o poi termini correttamente.

**CONSIGLI**

- Complessità
- Ricorsione *versus* iterazione

Soluzioni degli esercizi sul sito web [www.mheducation.it](http://www.mheducation.it)

## Esercizi

**Esercizio 1** Si converta la seguente funzione iterativa in ricorsiva; la funzione calcola il valore approssimato di e, base dei logaritmi naturali, sommando la serie:

$$1 + 1/1! + 1/2! + \dots + 1/n!$$

fino a che i termini da aggiungere non saranno più significativi con l'approssimazione voluta:

```
float loge()
{
    float enl, delta, fact;
    int n;
```

```
enl = 1.0;
n = 1;
fact = 1.0;
delta = 1.0;
do
{
    n++;
    fact *= n;
    delta = 1.0 / fact;
}
while (enl != enl + delta);
return enl;
```

**Esercizio 1** Spiegare perché la seguente funzione può restituire un valore sbagliato:

```
long fattoriale(long n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return n * fattoriale (--n);
}
```

**Esercizio 2** Qual è la sequenza numerica generata dalla seguente funzione ricorsiva?

```
long f(int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return 3 * f(n - 2) + 2 * f(n - 1);
}
```

**Esercizio 3** Scrivere una funzione ricorsiva `int lungstring(char S[])` che calcoli la lunghezza di una stringa.

*Suggerimento:* condizione di terminazione `S[] == 0` (stringa vuota).

**Esercizio 4** Pensare la funzione ricorsiva per produrre la sequenza delle permutazioni di una lista di numeri.

**Esercizio 5** Scrivere una funzione ricorsiva che calcoli la funzione di Ackermann:

$$\begin{aligned} A(m, n) &= n + 1 && \text{se } m = 0 \\ A(m, n) &= A(m - 1, 1) && \text{se } n = 0 \\ A(m, n) &= A(m - 1, A(m, n - 1)) && \text{se } m > 0 \text{ e } n > 0 \end{aligned}$$

**Esercizio 6** Qual è la sequenza numerica generata dalla seguente funzione ricorsiva?

```
int f(int n)
{
    if (n == 0)
        return 1;
    else if (n == 1)
        return 2;
    else
        return 2*f(n - 2) + f(n - 1);
}
```

**Esercizio 7** Il maggiore di un array di  $n$  interi si può calcolare ricorsivamente. Definire la funzione:

```
int max(int x, int y);
```

che restituisce il maggiore di due interi  $x$  e  $y$ . Definire la funzione:

```
int maxarray(int a[], int n);
```

che utilizza la ricorsione per trovare il maggiore dell'array di interi  $a[]$ .

*tappo:*  $n = 1$   
*passo:*  $\maxarray = \max(a[0] \dots a[n-2], a[n-1])$

**Esercizio 8** Scrivere una funzione ricorsiva che calcoli il numero di combinazioni di  $m$  elementi in classe  $n$ .

$$C(m, n) = \frac{m!}{n!(m-n)!}$$

**Esercizio 9** Scrivere una funzione ricorsiva che restituisca `true` se e solo se le viene passata come argomento una stringa contenente una frase palindroma.

**Esercizio 10** Quello delle "otto regine" è un classico problema che consiste nel porre otto regine su una normale scacchiera in maniera che nessuna sia sotto scacco di un'altra. Scrivere un programma che contenga una funzione ricorsiva per risolvere questo problema.

*Suggerimento:* proposta per enumerare le caselle di una scacchiera.

**Esercizio 11** la somma dei primi  $n$  numeri interi è data da:

$$1 + 2 + 3 + \dots + n = n(n+1)/2$$

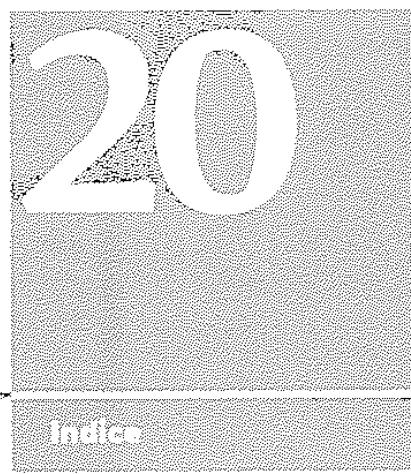
Inizializzare l'array `a[]` con i primi 50 interi. La media degli elementi del vettore è 25.5. Verificarlo con la funzione ricorsiva `media(float a[], int n)`.

**Esercizio 12** Scrivere una funzione ricorsiva che calcoli la funzione potenza  $a^n$  (con  $n$  positivo).

**Esercizio 13** Scrivere una funzione ricorsiva per il calcolo del quoziente intero di  $n$  per  $m$ , dove  $m$  e  $n$  sono interi positivi.

**Esercizio 14** Scrivere un programma che decida se un numero naturale è pari o dispari mediante una funzione ricorsiva indiretta.

**Esercizio 15** Scrivere una funzione ricorsiva che sommi le cifre di un numero naturale.



- |  |  |
|--|--|
| <b>20.1</b> Gli alberi                     | <b>20.4</b> Visita di un albero          |
| <b>20.2</b> Alberi binari                  | <b>20.5</b> Albero binario<br>di ricerca |
| <b>20.3</b> Struttura di un albero binario |  |

## Introduzione

In questo capitolo introduciamo gli "alberi", strutture dati molto importanti in informatica e nella scienza della programmazione. A differenza degli array e delle liste, gli alberi sono strutture *non lineari*. Essi sono molto utilizzati, per esempio per rappresentare formule algebriche,

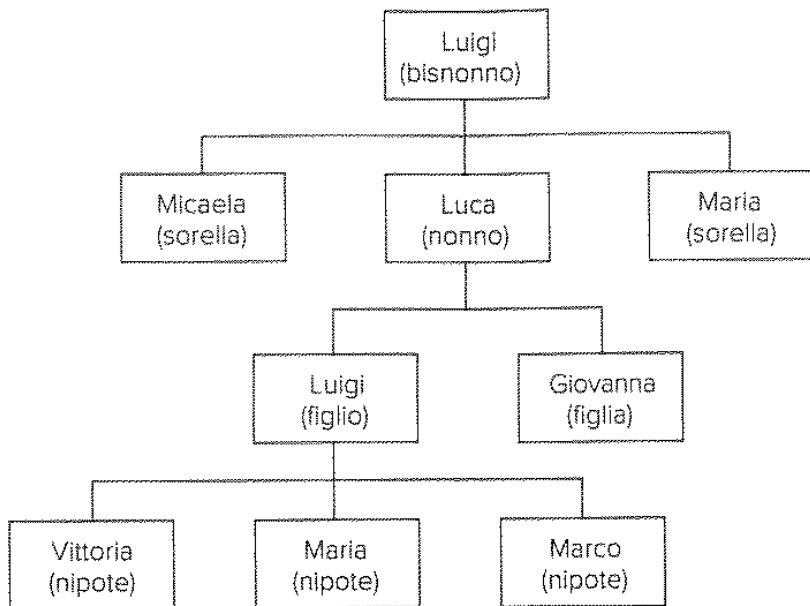
aumentare la velocità dell'estrazione di dati da archivi, nell'intelligenza artificiale e nella crittografia. I "file system" dei sistemi operativi sono organizzati ad albero, e alberi vengono pure utilizzati nella progettazione di compilatori, elaboratori di testo e algoritmi di ricerca.

### 20.1 Gli alberi

Si dice "grafo diretto" un insieme di nodi (che contengono informazioni, come nelle liste) tutti legati "a due a due" da archi direzionali (puntatori). Un *albero* è un grafo diretto in cui ogni nodo può avere un solo arco entrante e un qualunque numero di archi uscenti. Se un nodo non ha archi uscenti si dice "foglia". Se un nodo non ha archi entranti si dice "radice". Poiché in un albero non ci sono nodi con due o più archi entranti, per ogni albero vi deve essere una e una sola radice.

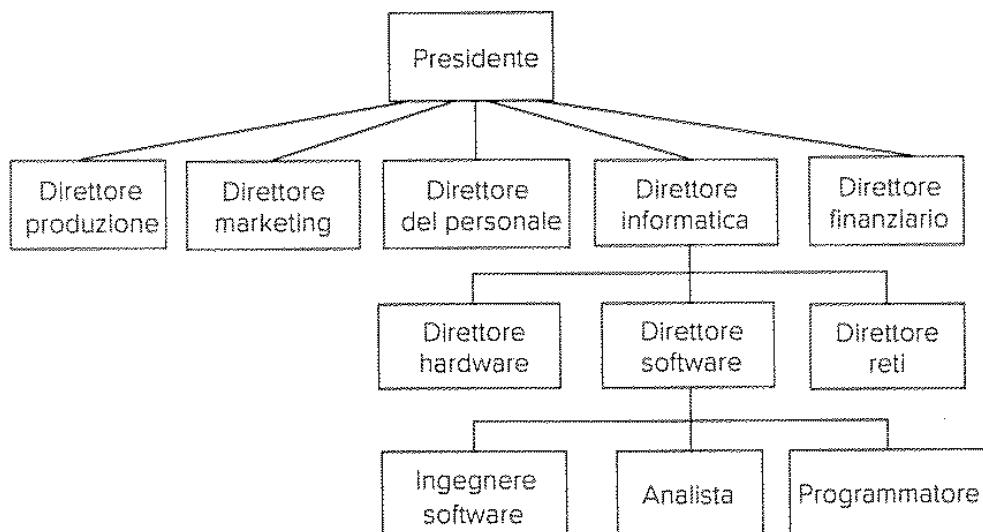
Probabilmente l'esempio più rappresentativo del concetto di albero è l'"albero genealogico" (Figura 20.1 E Figura 20.2).

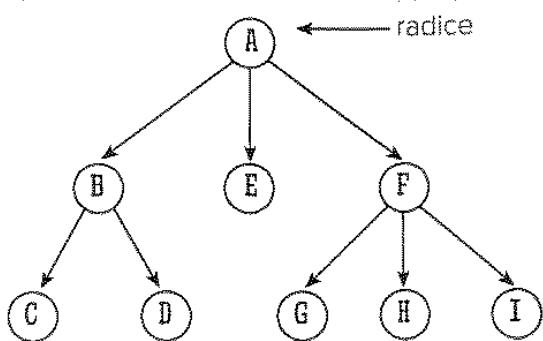
Per analogia con l'albero genealogico, il nodo da cui un arco parte si dice *padre*, un nodo a cui questo arriva si dice *figlio*. Per esempio, in Figura 20.3 il nodo *B* è il padre dei figli *C* e *D*. Due nodi con lo stesso padre sono detti fratelli. Da ogni nodo non-foglia di un albero si dirama un sottoalbero, quindi si intuisce la natura ricorsiva di questa struttura dati. Dato un nodo, i nodi che appartengono al suo sottoalbero si dicono suoi *descendenti*. Dato un qualunque nodo, i nodi che si trovano nel *cammino* dalla radice a esso sono i suoi *ascendenti*. Per esempio, *B* e *A* sono ascendenti di *C*.



Il livello di un nodo è la sua distanza dalla radice; la radice ha livello 0, i suoi figli livello 1, i suoi nipoti livello 2 e così via. I fratelli hanno lo stesso livello, ma non tutti i nodi dello stesso livello sono fratelli.

Nella Figura 20.4, il cammino dalla radice alla foglia I si rappresenta con la sequenza dei nomi dei nodi AFI. La profondità di un albero è la lunghezza del suo cammino più lungo aumentata di uno (per definizione la profondità di un albero vuoto è 0), ovvero è il livello della sua foglia più profonda. La Figura 20.4 contiene nodi su tre livelli: 0, 1 e 2, e ha profondità 3.



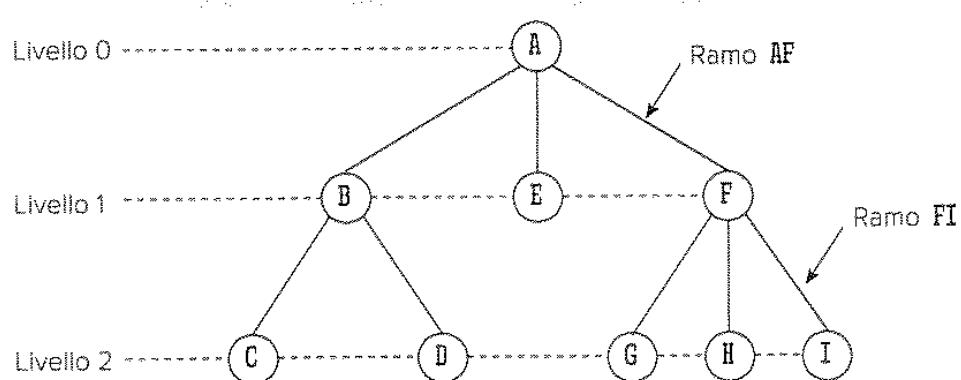
**Figura 20.3****Albero.**

Il livello di un nodo è la sua distanza dalla radice. La profondità di un albero è la lunghezza del suo cammino più lungo aumentata di uno.

Come già osservato, il concetto di sottoalbero mette in luce la struttura intrinsecamente *ricorsiva* dell'albero. Un albero è un insieme di nodi tale per cui:

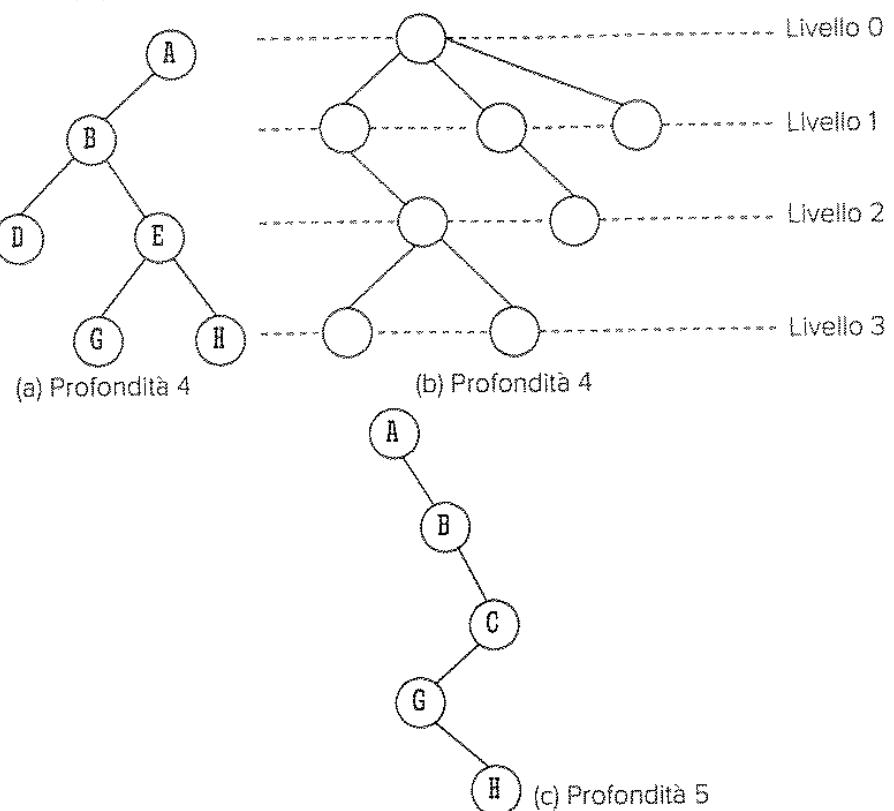
1. o è vuoto;
2. oppure ha un nodo denominato *radice* da cui discendono zero o più sottoalberi che sono essi stessi alberi (Figura 20.5).

Un albero di profondità  $h$  è *equilibrato* quando, dato un numero massimo di  $k$  figli per ciascun nodo, ogni nodo di livello  $l < h - 1$  ha esattamente  $k$  figli. Un albero di profondità  $h$  è *perfettamente equilibrato* quando ciascun nodo di livello  $l < h$  ha esattamente  $k$  figli (Figura 20.6).

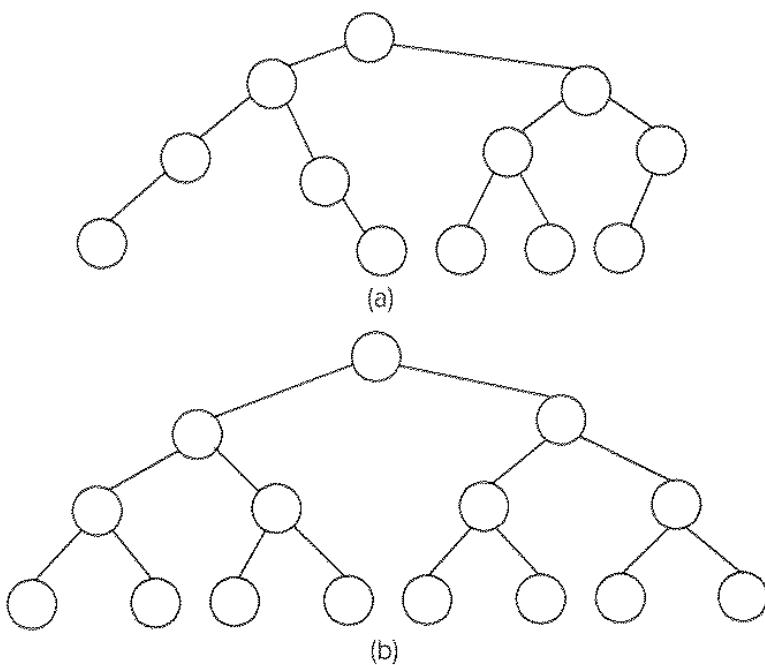


Padri:	A, B, F	Foglie:	C, D, E, G, H, I
Figli:	B, E, F, C, D, G, H, I	Nodi Interni:	B, F
Fratelli:	{B, E, F}, {C, D}, {G, H, I}		

**Figura 20.4****Terminologia degli alberi.**



**Figura 20.5**  
Alberi di varie profondità.



**Figura 20.6**  
(a) Un albero equilibrato. (b) Un albero perfettamente equilibrato.

## 20.2 Alberi binari

Un albero è *binario* se ogni nodo non ha più di due figli (sottoalberi), un sinistro e un destro (Figura 20.7.)

A ogni livello  $n$ , un albero binario può contenere da 1 a  $2^n$  nodi.

Nella Figura 20.8 (a) l'albero A contiene 8 nodi e ha Profondità 3, mentre b contiene 5 nodi e ha Profondità di 4. Questo secondo caso è un albero "degenerato" in una lista semplicemente collegata, perché c'è solo un nodo foglia (E) e ogni nodo non-foglia ha solo un figlio.

### 20.2.1 Equilibrio

Il livello di un nodo determina l'efficienza con la quale esso può essere localizzato. Una caratteristica importante di un albero binario è l'*equilibrio*. Si

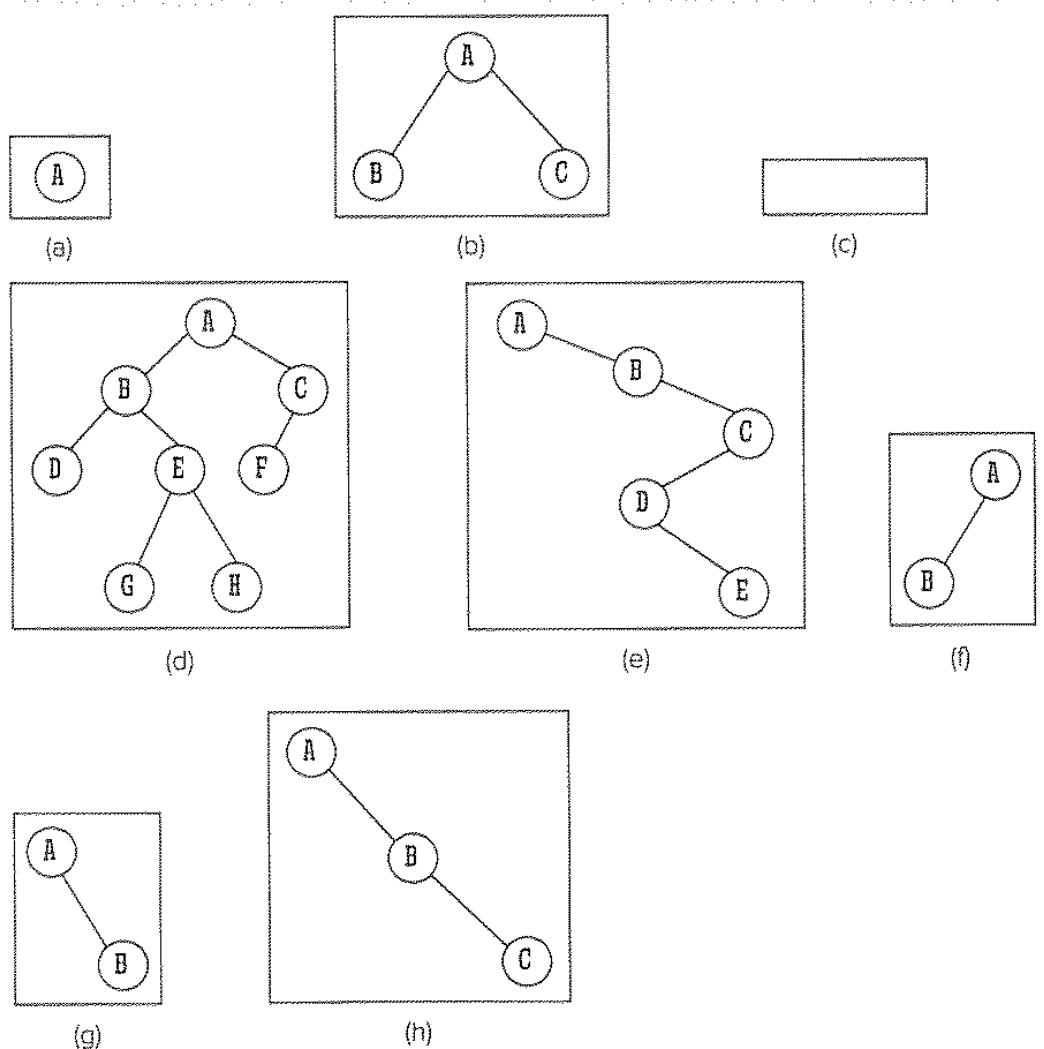
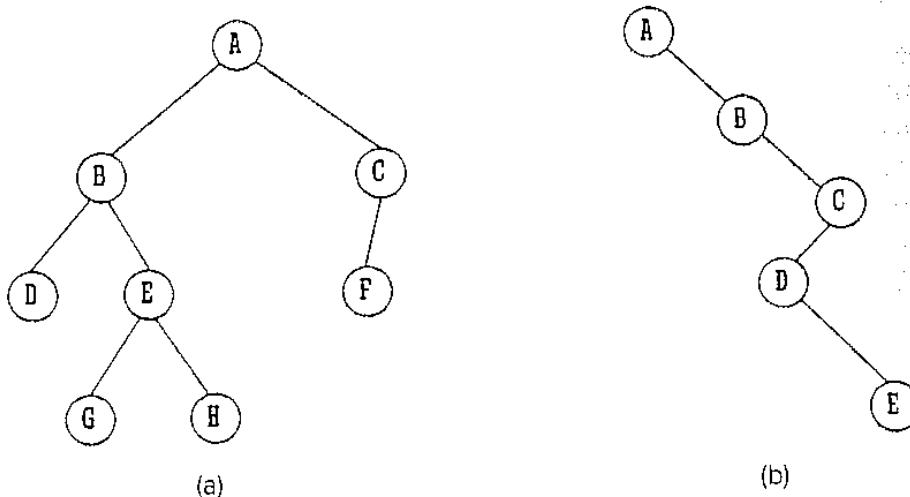


Figura 20.7

Alberi binari.



**Figura 20.8**  
Alberi binari: (a) profondità 4; (b) profondità 5

definisce *fattore di equilibrio* di un albero binario la differenza di profondità fra i suoi due sottoalberi. Denominate rispettivamente  $H_s$  e  $H_d$  le altezze dei sottoalberi sinistro e destro dell'albero B, il suo fattore di equilibrio è:

$$B = H_D - H_S$$

Utilizzando questa formula agli otto alberi della Figura 20.7 si ha (a) 0, (b) 0, (c) 0, (d) -1, (e) 4, (f) -1, (g) 1, (h) 2.

Un albero si dice *equilibrato* se sia lui sia i suoi sottoalberi hanno equilibrio 0. Poiché ciò capita di rado, si utilizza una definizione alternativa: un albero si dice *equilibrato* se sia lui sia i suoi sottoalberi hanno equilibrio 1 oppure 0 oppure -1.

Un albero binario si dice *degenerato* se ha una sola foglia. Un albero degenerato è una lista semplice.

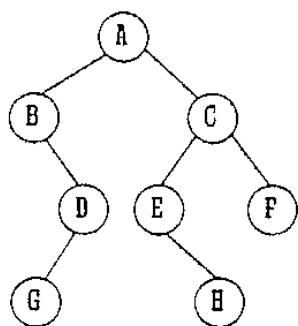
### 20.3 Struttura di un albero binario

La struttura di un albero, in particolare di un albero binario, è simile a quella delle liste concatenate, con la differenza che i nodi hanno più di un puntatore all'elemento successivo, perché ogni nodo può avere più successivi. Nel caso dell'albero binario i puntatori si dicono *destro* e *sinistro* (Figura 20.9).

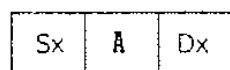
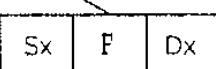
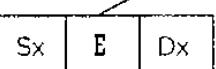
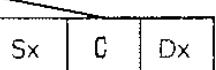
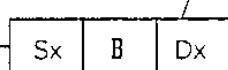
20-21 October 2013, <http://www.ias.edu/ias-conference-series/2013/2013-10-20-21>

Operazioni tipiche sugli alberi binari sono:

- determinarne l'altezza;
  - determinarne il numero di elementi;
  - copiarli;
  - visualizzarli;



(a) Albero

Sx = Sinistro  
Dx = Destro

Sx = Sinistro  
Dx = Destro

(b) Struttura

**Figura 20.9**

(a) Albero binario e (b) relativa struttura in nodi.

- confrontarli;
- eliminarli;
- valutare l'espressione algebrica che essi eventualmente rappresentano.

Come vedremo, queste operazioni si possono compiere visitando l'albero in maniera ricorsiva.

### 20.3.2 Rappresentazione di un albero binario

In C++ la struttura di un albero binario si può rappresentare così.

1. La classe nodo è annidata dentro la classe albero.

```

class albero
{
    class nodo
    {
        public:
    
```

```

        char* dati;
private:
    nodo* destro;
    nodo* sinistro;
    friend class albero;
];
public:
    nodo* radice;           // radice dell'albero
    albero() {radice = NULL; }
                    // ... altre funzioni membro
];

```

2. questa classe AlberoBin non contiene la classe NodoAlbero, però la dichiara come classe amica.

```

template <typename T> class AlberoBin;
// dichiara un oggetto nodo di un albero binario
template < typename T > class NodoAlbero
{
private:
    NodoAlbero <T>* sinistro;
    NodoAlbero <T>* destro;
    T dati;
public:
    NodoAlbero(T item, NodoAlbero <T> *ps, NodoAlbero <T> *pd) {
        dati = item;
        sinistro = ps;
        destro = pd;
    }
    // metodi per accedere agli attributi puntatore
    NodoAlbero <T>* OSinistro() {return sinistro;}
    NodoAlbero <T>* ODestro(void) {return destro;}
    void PSinistro (NodoAlbero <T>* Sx) { sinistro = Sx;}
    void PDestro (NodoAlbero <T>* Dx) { destro = Dx;}
    // metodi per accedere all'attributo dato
    T Odati() { return dati;}
    void Pdati( T e){ dati = e;}
    friend class AlberoBin <T>;
};

template <class T> class AlberoBin
{
private:
    NodoAlbero <T> * p;
public:
    AlberoBin() { p = NULL;}
    // altre funzioni membro
};

```

## 20.4 Visita di un albero

Ci sono vari modi per visitare i nodi di un albero; ciò che si richiede in genere è che ogni nodo sia esaminato una sola volta e l'attraversamento dell'albero avvenga secondo un criterio predeterminato. Partendo dalla radice, in linea generale un albero può essere visitato o "in profondità" oppure "in ampiezza".

Nella visita in profondità, dopo il nodo corrente viene esaminato un suo figlio, quale dipende dal particolare algoritmo; se non c'è alcun figlio si ritorna dal padre ("backtracking" semplice) verso un eventuale altro figlio e così via. Così facendo tutti i discendenti di ogni nodo vengono visitati prima dei suoi eventuali fratelli o pari livello.

Nella visita in ampiezza, dopo il nodo corrente viene visitato un altro nodo di pari livello. Così facendo tutti i nodi di un livello verranno visitati prima di tutti i nodi del livello successivo.

Focalizzando l'attenzione sugli alberi binari, esistono tre strategie di visita notevoli (Figura 20.10), l'inordine, la preordine e la postordine.

### 20.4.1 Visita in preordine

La visita in preordine visita prima la radice, quindi il sottoalbero sinistro e da ultimo quello destro. L'algoritmo è quindi ricorsivo: sia per processare il sottoalbero sinistro sia quello destro si richiama lo stesso procedimento di visita in preordine. Per esempio, nel caso dell'albero della Figura 20.11 si visita prima la radice A, quindi il sottoalbero sinistro in preordine, cioè prima B, poi D e quindi E. Successivamente si visita in preordine il sottoalbero destro, cioè la sua radice C, poi F e quindi G. Di conseguenza la visita in preordine dell'albero della Figura 20.11 restituisce nell'ordine A-B-D-E-C-F-G. In C++ l'algoritmo potrebbe essere espresso così:

```
void PreOrdine(Nodo* p)
{
    if (p)
    {
        cout << p->dati << " "; // visita la radice
        PreOrdine(p->sinistro); // visita il sottoalbero sinistro
```

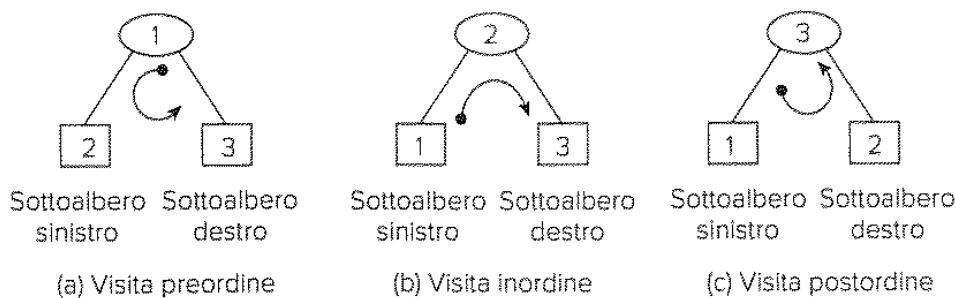
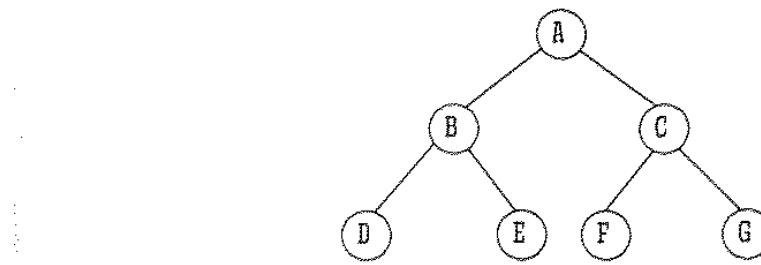


Figura 20.10

Visita di alberi binari.



**Figura 20.11** Visita preordine di un albero binario.

```

        PreOrdine(p -> destro); // visita il sottoalbero destro
    }
}
  
```

dove l'istruzione di stampa del dato nel nodo radice può essere rimpiazzata da qualunque tipo di manipolazione dello stesso.

#### 20.4.2 Visita inordine

La visita inordine processa prima il sottoalbero sinistro, quindi la radice e infine il sottoalbero destro.

Nell'albero della Figura 20.11 bisogna quindi visitare inordine il sottoalbero con radice B, cosa che richiede di visitare prima il sottoalbero con radice D. Quest'ultimo non ha sottoalberi quindi il primo nodo visitato è proprio la sua radice D. Dopodiché si visita la radice del sottoalbero con radice B, cioè il nodo B stesso, e poi il suo sottoalbero destro, cioè il nodo E. Quindi si risale alla radice A e poi al suo sottoalbero destro. La visita di quest'ultimo richiede prima la visita del suo sottoalbero sinistro, cioè F, la sua radice, cioè C, e da ultimo il suo sottoalbero destro G. Quindi la visita inordine dell'albero produce D-B-E-A-F-C-G.

La seguente funzione C++ schematizza l'algoritmo di visita inordine di un albero binario.

```

void InOrdine(Nodo *p)
{
    if (p)
    {
        InOrdine(p -> sinistro); // visita il sottoalbero sinistro
        cout << p -> dati << ' ';
        InOrdine(p -> destro); // visita il sottoalbero destro
    }
}
  
```

#### 20.4.3 Visita postordine

La visita in postordine processa prima il sottoalbero sinistro, poi quello destro e infine la radice.

Se si utilizza la visita in postordine dell'albero della Figura 20.11, si visita prima il sottoalbero con radice B, cosa che, ricorsivamente, richiede la visita del suo sottoalbero sinistro D, quindi il sottoalbero destro E, poi la sua radice B. Quindi si passa al sottoalbero con radice C, che richiede la visita del sottoalbero sinistro F, poi quello destro G, e la radice C. Da ultimo si visita la radice dell'albero A. Quindi la visita in postordine dell'albero produce la sequenza: D-E-B-F-G-C-A.

La seguente funzione C++ schematizza l'algoritmo di visita in postordine di un albero binario.

```
void PostOrdine(Nodo *p)
{
    if (p)
    {
        PostOrdine(p -> sinistro); // visita il sottoalbero sinistro
        PostOrdine(p -> destro); // visita il sottoalbero destro
        cout << p -> dati << ' '; // visita la radice
    }
}
```

#### Esempio 20.1

*Rappresentare sia il nodo sia le funzioni ricorsive mediante template.*

#### Classe nodo

```
template <typename T> class NodoAlberoBin {
public :
    NodoAlberoBin() {FiglioSinistro = FiglioDestro = 0;}
    NodoAlberoBin(const T& e)
    {
        info = e;
        FiglioSinistro = FiglioDestro = 0;
    }
    NodoAlberoBin(const T& e, NodoAlberoBin <T>* s, NodoAlberoBin <T>* d)
    {
        info = e;
        FiglioSinistro = s;
        FiglioDestro = d;
    }
private :
    T info;
    NodoAlberoBin <T> *FiglioSinistro, // sottoalbero sinistro
                  *FiglioDestro; // sottoalbero destro
};
```

*Visita PreOrdine di \*t*

```
template <typename T> void PreOrdine(NodoAlberoBin<T> *t)
```

```

    [
      if (t) {
        Visita(t);
        PreOrdine(t -> FiglioSinistro);
        PreOrdine(t -> FiglioDestro);
      }
    ]
  
```

*Visita InOrdine di \*t*

```

template <typename T> void InOrdine(NodoAlberoBin<T> *t)
{
  if (t) {
    InOrdine(t -> FiglioSinistro);
    Visita(t);
    InOrdine(t -> FiglioDestro);
  }
}
  
```

*Visita PostOrdine di \*t*

```

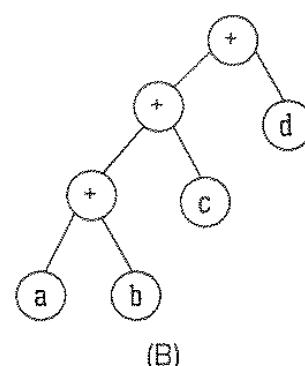
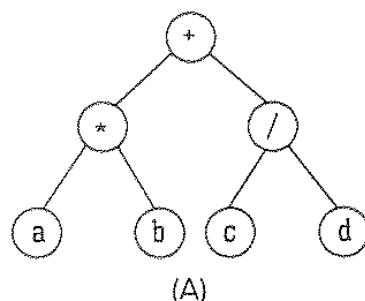
template <typename T> void postordine(NodoAlberoBin<T> *t)
{
  if (t) {
    postordine(t -> FiglioSinistro);
    postordine(t -> FiglioDestro);
    Visita(t);
  }
}
  
```

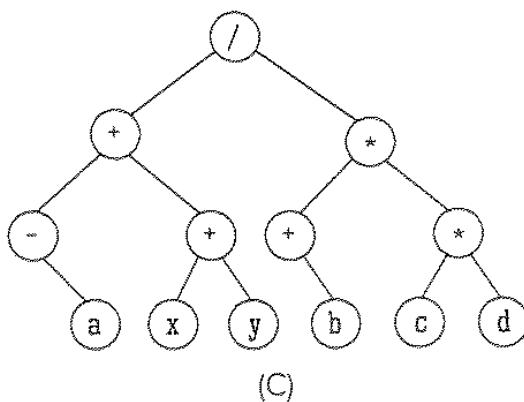
*dove la funzione Visita() può essere rimpiazzata con:*

```
cout << t->info;
```

### Esempio 20.2

Indicare le sequenze di visita dei seguenti alberi binari.





Sequenze prodotte dalle visite preordine, inordine e postordine.

	Albero A	Albero B	Albero C
PreOrdine	+*ab/cd	***abcd	/*-a+xy*+ b* cd
InOrdine	a*b+c/d	a+b+c+d	-a+x+y/+ b* c*d
PostOrdine	ab*cd/+	ab*c+d+	a-xy**+b*cd**/

#### 20.4.4 Profondità di un albero binario

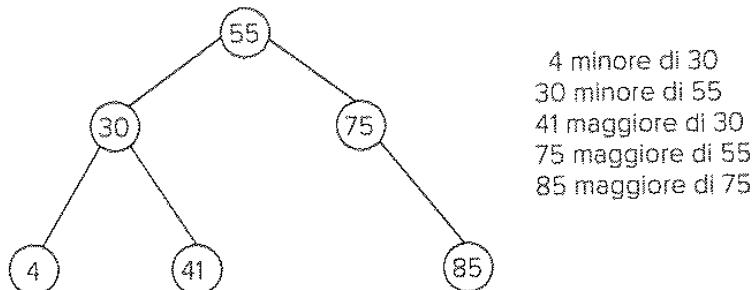
La funzione Profondita valuta la *profondità* dell'albero binario passatole mediante un puntatore al nodo radice. Il caso più semplice è quello di un albero vuoto, la cui profondità è 0. Se l'albero non è vuoto bisogna valutare separatamente le profondità ProfonditaSx e ProfonditaDx di ogni sottoalbero. L'algoritmo consiste nelle chiamate ricorsive della funzione Profondita con i due puntatori ai due sottoalberi come parametro. La funzione Profondita restituisce la profondità del sottoalbero più profondo aumentata di 1 (quella della radice).

```
int Profondita(Nodo *p)
{
    if (!p) return 0;
    else
    {
        int ProfonditaSx = Profondita(p->sx);
        int ProfonditaDx = Profondita(p-> dx);
        if (ProfonditaSx > ProfonditaDx)
            return ProfonditaSx + 1;
        else
            return ProfonditaDx + 1;
    }
}
```

#### 20.5 Albero binario di ricerca

Gli alberi visti finora non hanno alcun ordine definito riguardo i valori presenti dentro i nodi.

In un albero binario *di ricerca*, dato un qualunque nodo, tutti i nodi del suo sottoalbero sinistro hanno valori minori (o maggiori) del suo, mentre tutti i nodi di quello destro li hanno maggiori (o minori) del suo. Il seguente è un esempio di albero binario di ricerca.



#### 20.5.1 Creazione di un albero binario di ricerca

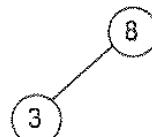
Supponiamo di dover memorizzare i seguenti numeri in un albero binario di ricerca.

8    3    1    20    10    5    4

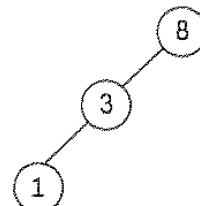
Inizialmente bisogna inserire il valore 8 in un albero vuoto, quindi l'unica scelta è metterlo nella radice.



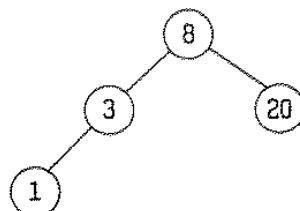
Poiché il successivo 3 è minore di 8, esso deve andare nel sottoalbero sinistro.



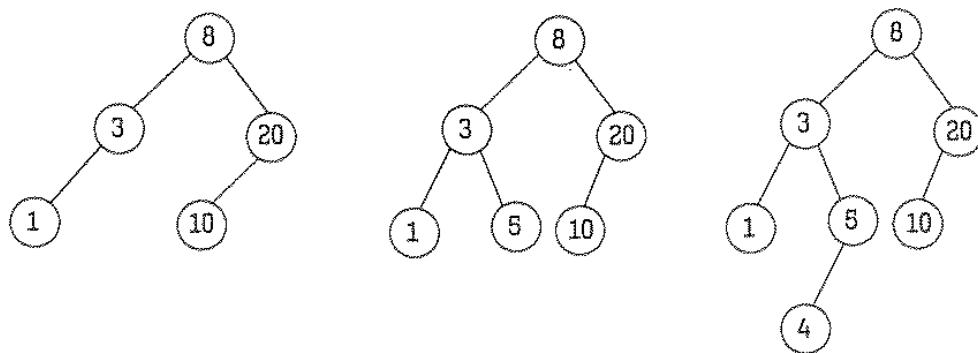
Quindi bisogna inserire 1 che è minore sia di 8 che di 3, quindi deve andare nel sottoalbero sinistro di 3.



Il successivo 20, maggiore di 8, deve andare alla sua destra.



e così via. Ogni nuovo elemento viene inserito come foglia dell'albero corrente.



### 20.5.2 Implementazione di un nodo di un albero binario di ricerca

Gli alberi binari di ricerca servono a ritrovare velocemente i dati in esso memorizzati. Supponiamo di avere un albero binario in cui ciascun nodo contiene il nome e il numero di matricola di uno studente.

```

class Nodo {
public :
    int numero_matricola;
    char cognome_nome[30];
    Nodo *sinistro, *destro;
    Nodo (int id = 0, char *n = 0, Nodo *i = 0, Nodo *d = 0)
        : numero_matricola(id), sinistro(i), destro(d)
        { strcpy(cognome_nome,n) ; }
};
  
```

### 20.5.3 Ricerca

La ricerca di un nodo inizia dalla radice e segue questi passi:

1. si confronta la chiave cercata con quella della radice;
2. se coincidono s'è trovato l'elemento cercato;
3. se la chiave cercata è maggiore di quella della radice la ricerca viene avviata ricorsivamente nel sottoalbero destro, altrimenti in quello sinistro.

Nel nostro caso la funzione di ricerca ha due argomenti, il puntatore all'albero da esaminare e il numero di matricola dello studente da cercare. Se lo trova, la funzione restituisce il puntatore al nodo dello studente, altrimenti restituisce il valore 0. Il codice C++ è:

```

Nodo* ricerca(Nodo* p, int cercato)
{
    if (!p) return 0;
    else if (cercato == p->numero_matricola) return p;
    else if (cercato < p->numero_matricola)
        return ricerca(p->sinistro, cercato);
    else return ricerca(p->destro, cercato);
}
  
```

### 20.5.4 Inserire un nodo

L'inserimento di un nuovo elemento in un albero di ricerca deve essere fatto in maniera tale che il risultato dell'operazione sia ancora un albero di ricerca. Si visita l'albero alla ricerca dell'elemento da inserire, e se lo si trova non si fa nulla (perché l'elemento c'è già); altrimenti si inserisce l'elemento nel posto dove la ricerca è terminata (senza trovare nulla), perché quello è il posto dove l'elemento sarebbe dovuto stare se fosse stato presente (Figura 20.12).

Per esempio, nella Figura 20.12 per inserire l'elemento 8 si parte dal nodo radice 25; l'8 dovrebbe stare nel sottoalbero sinistro perché  $8 < 25$ . Rispetto poi al nodo 10, l'8 dovrebbe stare nel sottoalbero sinistro che al momento è vuoto. Quindi il nodo 8 sarà il figlio sinistro del nodo 10.

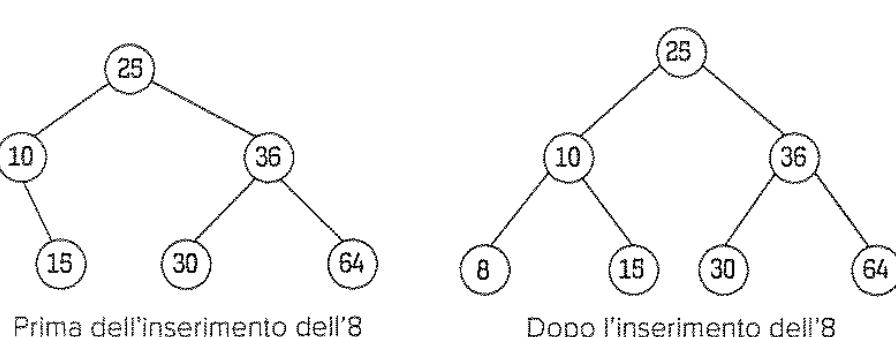
L'inserimento di un nodo inizia quindi dalla radice e segue questi passi:

1. *assegnare dinamicamente memoria per il nuovo nodo;*
2. *cercare l'elemento nell'albero per inserirlo come foglia;*
3. *appendere il nuovo elemento all'albero.*

La funzione inserire ha tre argomenti: il puntatore alla radice dell'albero, il puntatore al nuovo elemento e il numero di matricola dello studente. L'algoritmo crea un nodo per il nuovo studente e lo va a inserire nel posto corretto come indicato:

```
void inserire(Nodo* p, int nuova_mat, char* nuovo_studente)
{
    if (!p) p = new Nodo(nuova_mat, nuovo_studente);
    else if (nuova_mat < p->numero_matricola);
        inserire(p->sinistro, nuova_mat, nuovo_studente);
    else inserire(p->destro, nuova_mat, nuovo_studente);
}
```

Si noti che se l'albero corrente è vuoto ( $!p$ ) il nuovo nodo ne diventa la radice.



**Figura 20.12**  
Inserimento in un albero binario di ricerca.

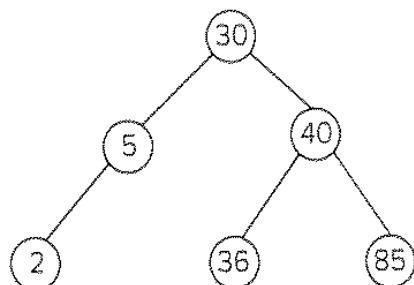
### 20.6.5 Eliminazione di un nodo

Anche l'eliminazione di un nodo da un albero binario è un'estensione dell'algoritmo di ricerca. Ovviamenete l'eliminazione deve produrre comunque un albero di ricerca. I passi da seguire sono:

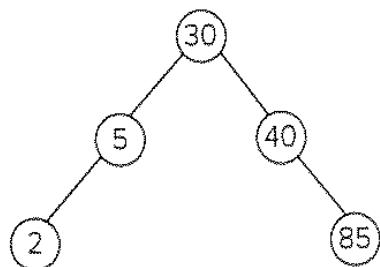
1. cercare l'elemento da eliminare;
2. aggiustare i puntatori dei suoi antenati se il nodo da eliminare ha meno di due figli, oppure sostituirlo con il nodo con chiave più vicina in modo da conservare la natura di albero di ricerca binario.

#### Esempio 20.3

Sopprimere l'elemento con chiave 36 nel seguente albero di ricerca:

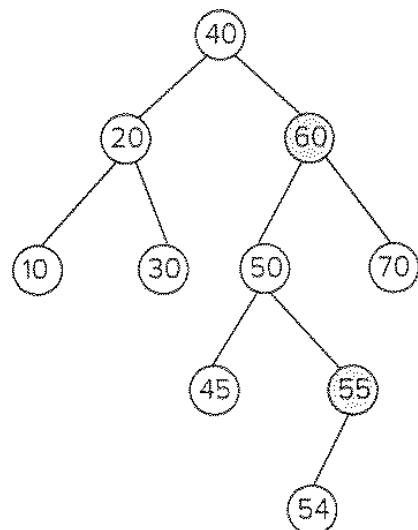


L'albero risultante è

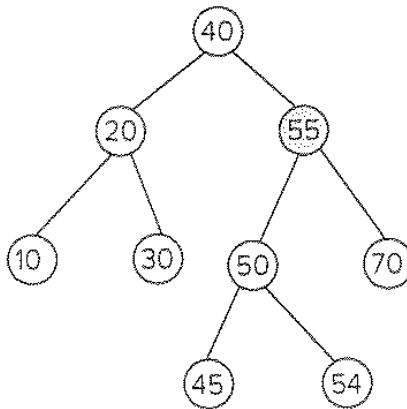


#### Esempio 20.4

Eliminare l'elemento con chiave 60 nel seguente albero:



Si rimpiazza 60 con l'elemento maggiore del suo sottoalbero sinistro (55) o con l'elemento più piccolo del suo sottoalbero destro (70). Se si opta per il primo caso, si sposta il 55 alla radice del sottoalbero e si riaggiusta l'albero.



In questo capitolo abbiamo visto cosa sono e come si gestiscono gli alberi. Strutture dati dinamiche, sono particolari grafi molto utilizzati per varie applicazioni. In particolare abbiamo introdotto l'*albero binario* come quello in cui ogni elemento possiede al massimo due figli. Essi sono utilizzati per rappresentare espressioni logico-aritmetiche: i nodi interni contengono gli operatori, e le

foglie gli operandi. Gli alberi binari di ricerca sono invece caratterizzati da un ordinamento: dato un qualunque nodo, tutti i nodi del suo sottoalbero sinistro hanno valori minori del suo, mentre tutti i nodi del suo sottoalbero destro hanno valori maggiori. Questi alberi di ricerca sono fondamentali per memorizzare grandi quantità di dati che devono poi essere ritrovati in maniera veloce.

#### ESERCIZI

- Albero
- Albero binario
- Albero di ricerca binario
- Visita inordine
- Nodo
- Visita postordine
- Visita preordine
- Sottoalbero

## Esercizi

**Esercizio 1:** Scrivere un programma che processa un albero binario i cui nodi contengono caratteri; le funzioni sono le seguenti:

I (seguito da un carattere): inserisce un carattere

B (seguito da un carattere): elimina un carattere

RE: Visita inordine

RP: Visita in preordine

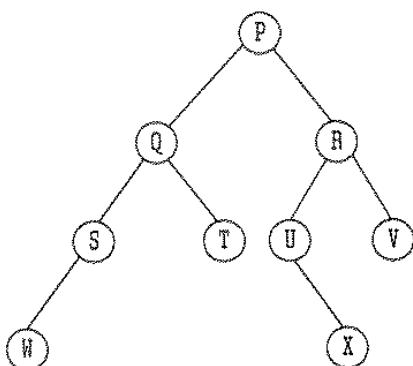
RT: Visita in postordine

**Esercizio 2:** Scrivere una funzione che accetta un albero in input e restituisce il numero dei suoi nodi.

**Esercizio 3:** Progettare una funzione iterativa che cerchi un elemento in un albero binario di ricerca.

**Esercizio 4:** Spiegare perché nessuna di queste strutture è un albero binario.

**Esercizio 5:** Dato il seguente albero:



- a) valutarne l'altezza;
- b) valutare se l'albero è equilibrato;
- c) elencare tutti i nodi foglia;
- d) visitarlo inordine, in preordine e in postordine.

**Esercizio 6:** Per ciascuna delle seguenti sequenze di lettere:

- (I) M,Y,T,E,R
- (II) R,E,M,Y,T
- (III) T,Y,M,E,R
- (IV) C,O,R,N,F,L,A,K,E,S

- a) disegnare l'albero binario di ricerca che esse generano;
- b) visitare l'albero inordine, in preordine e in postordine.

**Esercizio 7:** Costruire gli alberi d'espressione corrispondenti alle seguenti:

a)  $(A + B) / (C - D)$

b)  $A + B + C / D$

c)  $A - (B - (C - D)) / (E + F)$

d)  $(A + B) * ((C + D) / (E + F))$

e)  $(A - B) / ((C * D) - (E / F))$

**Esercizio 8:** La visita preordine di un certo albero binario produce **ADFGHKLPQRWZ**, mentre la visita inordine produce **GFKDLAHMRQPZ**. Disegnare l'albero binario.

**Esercizio 9:** Dato un albero binario di elementi di tipo `int`, scrivere funzioni che calcolino:

a) la somma dei suoi elementi;

b) la somma dei suoi elementi che sono multipli di 3.

**Esercizio 10:** Scrivere una funzione booleana che prende in input due alberi e restituisce `true` se e solo se essi sono identici.

**Esercizio 11:** Scrivere una funzione booleana che prende in input due alberi e restituisce `true` se e solo se essi hanno elementi in comune.

**Esercizio 12:** Progettare un algoritmo iterativo che implementi le visite inordine, preordine e postordine.

**Esercizio 13:** Un albero binario di ricerca si può implementare mediante un array; il nodo in  $i$ -esima posizione nell'array ha il figlio sinistro in posizione  $2i$  e il suo figlio destro in posizione  $2i + 1$ . A partire da questa rappresentazione progettare le funzioni per gestire l'albero (il cui numero di nodi sarà comunque limitato dalla dimensione dell'array).

**Esercizio 14:** Dato un file di testo, utilizzare un albero binario di ricerca come struttura ausiliaria per ordinare ascendentemente le parole del testo.

**Esercizio 15:** Dato un albero binario di ricerca, scrivere una funzione che elenchi i suoi nodi in senso ascendente.

**Esercizio 16:** Codificare la classe `Albero` come amica della classe `Nodo` per consentire l'implementazione di metodi ricorsivi. La classe `Albero` deve contenere il metodo

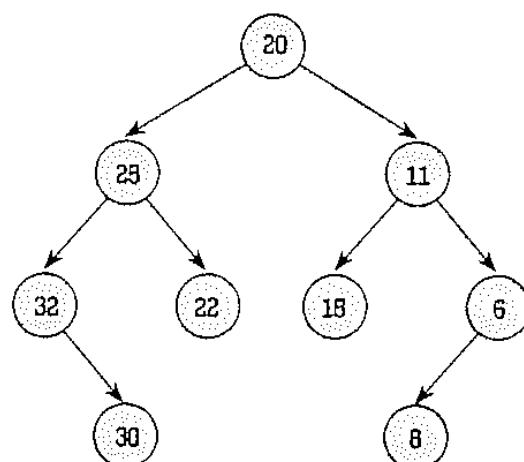
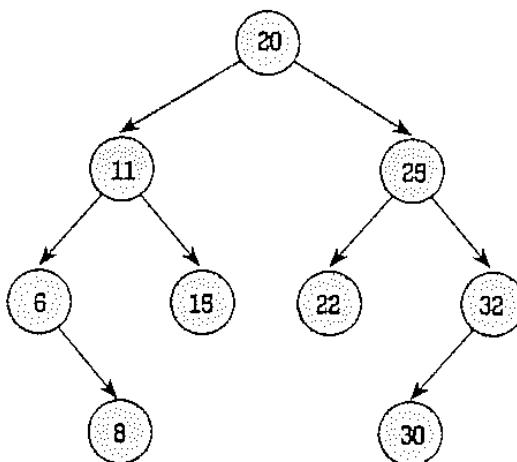
Ottenere per ricavare sia il figlio sinistro sia quello destro della radice dell'albero.

Esercizio Aggiungere alla classe albero dell'esercizio precedente il costruttore di copia.

Esercizio Aggiungere alla classe albero dell'esercizio precedente una funzione che copi

l'oggetto puntato nell'oggetto ricevuto come parametro.

Esercizio Scrivere un metodo della classe albero che restituisca il simmetrico dell'oggetto albero: per simmetrico intendiamo un albero in cui ogni nodo ha sottoalberi sinistro e destro invertiti, come nel seguente esempio.



Esercizio Aggiungere alla classe albero dell'esercizio precedente funzioni membro per compiere la visita ricorsiva inordine, preordine e postordine.

Esercizio Aggiungere alla classe albero dell'esercizio precedente una funzione che restituisca il numero dei suoi nodi.

Esercizio Aggiungere alla classe albero dell'esercizio precedente una funzione che restituisca il numero delle sue foglie.

Esercizio Aggiungere alla classe albero dell'esercizio precedente una funzione che prenda in input un numero naturale  $n$  e mostri i nodi che appartengono al livello  $n$ .

Esercizio Costruire una funzione membro della classe albero per visualizzare tutti i nodi di un albero binario di ricerca il cui valore del campo chiave sia maggiore di un valore accettato in input.

Esercizio Costruire due funzioni membro della classe albero, una ricorsiva e l'altra iterativa, per inserire un elemento accettato in input in un albero binario di ricerca.

Esercizio Costruire due funzioni membro della classe albero, una ricorsiva e l'altra iterativa, per eliminare un elemento accettato in input da un albero binario di ricerca.

# Indice analitico

## SIMBOLI

- , 361  
--, 73, 361  
!, 361  
? ;, 99  
0, 361, 374  
[], 361, 374  
\*, 361  
&, 361  
+, 361  
++, 73, 361  
<<, 256  
<ctime>, 148  
->, 360  
>>, 256  
|| (or), 77  
~, 361  
&& (and), 77  
#define, 59  
#include, 37  
! (not), 77

## A

acos(x), 146  
ADT, 20  
Alberi, 475  
- binari, 479  
- degenerati, 480  
- di espressione, 481  
- equilibrio, 479  
- operazioni, 480  
- profondità, 487  
- rappresentazione, 481  
- ricerca, 487  
- struttura, 480  
- visita, 483  
Algoritmo, 12  
ALU, 4, 8  
Ambiguità, 328  
ANSI/ISO C++, 41

Array, 161  
- - allocazione in memoria, 164  
- - definizione di un, 162  
- - di caratteri, 168  
- - dimensione di un, 164  
- - elementi di un, 163  
- - inizializzazione di un, 166  
- - multidimensionale, 169  
ASCII, 10  
asin(x), 146  
Assegnamento, 56  
- - operatore di, 70  
Astrazione, 18  
atan(x), 146

**B**

Binding, 336  
Bit, 6  
- - di stato, 272  
- - manipolazione dei, 79  
bitwise, operatori, 79  
bool, 49, 50  
break, 105, 115  
buffer, 262  
Byte, 6

**C**

Campi, 176  
Caratteri, 49  
- - alfabetici, 9  
- - di controllo, 9  
- - numerici, 9  
- - speciali, 9  
casting, 82  
catch, 391, 395  
ceil(x), 146

cerr, 256  
char, 49, 267  
*Chip*, 8  
Ciclo, 101  
- - annidato, 117  
- - corpo del, 101  
- - formato del, 112  
- - infinito, 105, 110  
- - sentinella, 104

- continue, 115  
 Conversioni di tipo, 82  
 Cortocircuito, valutazione in, 78  
 Costante, 51, 57  
 Costruttore, 301  
 $\cos(x)$ , 146  
`cout`, 62, 256  
`cout.put()`, 241  
 CPU, 4  
`cstring`, 44, 245  
`C-string`, 235  
 CU, 4
- D**  
 Dato  
   – alfanumerico, 6  
   – globale, 15  
   – locale, 15  
   – strutture, 176  
   – tipi di, 49  
`Debugging`, 37, 45  
`dec`, 267, 271  
 Decremento, 73  
`delete`, 68, 225, 360, 361, 379  
 Distruttori, 305, 325  
`double`, 49  
`do while`, 111, 112  
 DVD, 8
- E**  
 Eccezioni, 389  
   – gestione delle, 390  
   – lancio di, 395  
   – progetto delle, 393  
   – specifica delle, 398  
`endl`, 267  
`ends`, 267  
`enum`, 49  
`EOF`, 260  
 Ereditarietà, 18, 20, 21, 315  
   – ambiguità, 328  
   – multipla, 326  
   – privata, 320  
   – protetta, 320  
   – pubblica, 318  
   – tipi di, 317  
 Escape, sequenza di, 50  
 Espressioni, 65  
   – associatività, 67  
   – composte, 67  
   – precedenza, 67  
 Estrattore, 256
- Estrazione, operatore di, 61  
 $\exp(x)$ , 147  
 $\exp(x)$ , 147  
`extern`, 143
- F**  
 $\text{fabs}(x)$ , 146  
`false`, 50, 77  
`File`, 8  
`fixed`, 271  
`float`, 49, 267  
 $\text{floor}(x)$ , 146  
`flush`, 267  
 Flussi, 255  
   – binari, 256  
   – di testo, 255  
   – manipolatori, 268  
   – standard, 258  
 $\text{fmod}(x, e)$ , 146  
 Foglia, 475  
`for`, 106, 112  
 Formato, indicatori di, 271  
`free`, 222  
`friend`, 364  
`fstream`, 259  
 Funzioni, 15, 41, 123  
   – aleatorie, 147  
   – amiche, 364  
   – argomenti, 41, 136  
   – chiamata, 42  
   – chiamata di, 128  
   – definite dall'utente, 42  
   – definizione, 42  
   – dichiarazione, 42  
   – esponenziali, 147  
   – in linea, 138, 299  
   – logaritmiche, 147  
   – matematiche, 146  
   – membro, 239, 297  
   – nome delle, 126  
   – numeriche, 146  
   – parametri, 125  
   – passaggio di parametri, 132  
   – prototipi delle, 130  
   – ricorsive, 457  
   – risultati delle, 127  
   – sovraccaricamento delle, 153  
   – struttura delle, 125  
   – template di, 156, 342  
   – trigonometriche, 146  
   – virtuali, 336  
   – visibilità delle, 140
- G**  
`Garbage collector`, 221  
 Generalizzazione, 20  
`get()`, 263, 280  
`getline()`, 263  
`goto`, 115  
 Grafo diretto, 475  
 GUI, 22
- H**  
`Header file`, 37, 40, 56  
`Heap`, 221  
`hex`, 267, 271
- I**  
`IDE`, 34  
`if`, 90  
`if else`, 93  
`ifstream`, 258  
 Immagazzinamento, classi di, 139  
 Immagini, rappresentazione delle, 11  
 Incapsulamento, 16, 18, 20  
 Incremento, 73  
 Indicatori di stato, 104  
`Information hiding`, 291  
 Ingresso, 3  
   – dispositivi di, 5  
`inline`, 301  
 Inordine, visita, 484  
`Input/output`, 3, 60  
 Inseritore, 256  
 Inserzione, operatore di, 62  
`int`, 49, 267  
 Internet, 3  
 Interpreti, 24  
`interrupt`, 392  
`ios`, 257  
`iostream`, 39, 56, 256  
`isalnum(c)`, 145  
`isalpha(c)`, 145  
`iscntrl(c)`, 145  
`isdigit(c)`, 145  
`isgraph(c)`, 145  
`islower(c)`, 145  
`isprint(c)`, 145  
`ispunct(c)`, 145  
`isspace(c)`, 145  
`istream`, 256  
`istrstream`, 259  
 Istruzione  
   – ciclica, 113  
   – di salto, 114  
`isupper(c)`, 145

- I**
- isxdigit(c), 145
  - Iterazione, 101, 460
- L**
- Libreria, funzioni di, 44, 144
  - Liste
    - circolari, 424, 435
    - dinamiche, 445
    - doppiamente concatenate, 432
    - operazioni con le, 426
    - puntatori, 427
    - semplici, 424
  - long, 49
- M**
- main(), 41, 128
  - memcpy(), 247
  - Memoria, 5
    - automatica, 231
    - celle, 6
    - contenuto, 6
    - dinamica, 232
    - gestione dinamica della, 221
    - indirizzo, 6
    - organizzazione della, 6
    - overflow di, 230
    - secondaria, 8
    - statica, 231
  - Microprocessore, 8
  - mktime(t), 149
  - Modem, 9
  - Mouse, 5
  - Multiprocessore, 23
  - Multitasking, 23
  - Multiutenza, 23
- N**
- Namespace, 40, 54
  - new, 68, 223, 360, 361, 379
  - newline, 256
  - null, 199
  - Numeri
    - interi, 10
    - reali, 11
- O**
- Occultamento, 16, 18, 20, 291
  - oct, 271
  - ofstream, 259
  - Oggetti, 16, 289, 295
  - OOP, 16, 289
- P**
- Operandi, 65
  - operator, 360
  - Operatori, 22, 65, 67
    - aritmetici, 71
    - binari, 368
    - bitwise, 79
    - booleani, 77
    - condizionali, 80
    - logici, 77
    - relazionali, 75
    - sovraccaricamento degli, 359
    - speciali, 65
    - virgola, 81
  - Ordinamento, 407
    - a bolle, 417
    - algoritmi di, 411
    - per inserimento, 414
    - per scambio, 411
    - per selezione, 413
    - QuickSort, 468
    - Shell, 418
  - ostream, 256
  - ostringstream, 259
  - Output, 3
    - formattazione, 267
  - Overflow, 440
  - Overloading, 22, 307
- R**
- RAM, 7
  - rand(), 147
  - read(), 264, 280
  - register, 144
  - resetiosflags(f), 267
  - return, 128
  - RGB, 11
  - Ricerca, 407
    - algoritmi di, 410
    - binaria, 409
    - sequenziale, 407
  - Ricorsione, 457, 460
  - Riferimenti, 193
  - right, 271
  - Ripetizione, 89
  - Riusabilità, 21
  - Run time, 45
- S**
- scientific, 271
  - Scope, 53
  - seekg(), 282
  - Selettore, 96
  - Selezione, 89
  - Sentinella, 104
  - Separatori, 48
  - Sequenza, 89
  - setbase(n), 267
  - setfill(c), 267

- setiosflags(f), 267  
 setprecision(n), 267  
 setw(n), 267  
*Shell*, 418  
 short, 49  
 showbase, 271  
 showpoint, 271  
 showpos, 271  
 Sintassi, del costruttore, 323  
 sin(x), 147  
 Sistema operativo, 22  
 sizeof, 68, 81, 164  
*Software*, 4, 22  
 Sottoalbero, 475  
 Sottoclasse, 20  
 Sottoprogrammi, 15  
 Sovraccaricamento, 359  
 Spazio di nomi, 40, 54  
 sqrt(x), 146  
 srand(seme), 147  
*Stack*, 439, 461  
 Standard  
 – input, 255  
 – output, 255  
 static, 140, 141  
 Stato, indicatori di, 260  
 stdio, 271  
*Storage classes*, 139  
 strcat(), 247  
 strchr(), 247  
 strcmp(), 247  
 strcpy, 245  
 strcpy(), 247  
 strcspn(), 247  
 Stream, 255  
 Stringhe, 235  
 – conversione di, 251  
 – inizializzazione delle, 236  
 – lettura delle, 237  
 strlen(), 247  
 strncat(), 247  
 strncmp(), 247  
 strncpy(), 247  
 strnset(), 247  
 strpbrk(), 247  
 strrchr(), 247  
 strspn(), 247  
 strstr(), 247  
 strtok(), 247  
 struct, 177  
 Strutture  
 – annidate, 180  
 – array di, 183  
 – campi delle, 176  
 – dichiarazione delle, 177  
 – dimensione delle, 179  
 Subroutine, 15  
 Superclasse, 20  
 switch, 96, 105
- T
- tan(x), 147  
 tellg(), 282, 283  
 template, 349  
 Template, 156, 341  
 throw, 391, 395  
 time(ora), 149  
 tolower(c), 145
- U
- toupper(c), 145  
 true, 50, 77  
 try, 391, 393
- V
- UML, 18, 29  
 Underflow, 439  
 Unicode, 10  
 union, 187  
 Unioni, 187  
 unitbuf, 271  
 unsigned, 49  
 uppercase, 271
- W
- Variabili, 51  
 – automatiche, 142  
 – definizione di, 51  
 – esterne, 143  
 – globali, 54  
 – inizializzazione di, 51  
 – locali, 53, 140  
 – registro, 144  
 – statiche, 140  
 – visibilità delle, 53, 139
- Vettori, 161  
 Virgola mobile, 49  
 Visibilità, 139  
 void, 49, 50
- W
- Warnings, 46  
 while, 101, 105, 112  
 write(), 266, 280