Università della Calabria

Dipartimento di Chimica e Tecnologie Chimiche

Dispensa del corso di Informatica per Chimici

Corso di Laurea in Chimica

I anno

Donato D'Ambrosio e William Spataro

Dipartimento di Matematica e Informatica

Edizione A.A. 2015-2016

Indice

Parte I

Rappresentazione dati, architettura del Calcolatore e programmazione a basso livello

Capitolo 1

La codifica dei dati

I calcolatori elettronici sono costruiti assemblando in elementi più complessi, detti *circuiti*, semplici dispositivi elettronici noti con il nome di *transistor*. Alcune delle caratteristiche peculiari dei transistor sono le seguenti:

- I transistor sono dispositivi che possono assumere solo due stati differenti, corrispondenti a due differenti livelli di tensione elettrica, che indicheremo attraverso i simboli 0 (zero) e 1 (uno).

- I transistor possono essere realizzati a scala microscopica, per cui è possibile assemblare centinaia di milioni di elementi in pochi cm^2. Il microprocessore quad core Intel Core i7 2600QM (della famiglia Sandy Bridge) conta circa 995 milioni di transistor.

- I transistor sono in grado di cambiare di stato un un tempo molto piccolo. Più questo tempo è piccolo, più veloce risulta il calcolatore elettronico.

- I transistor sono dispositivi *bistabili*. Questo vuol dire che se viene rilevato che un transistor si trova nello stato $q = 0$, con tutta probabilità il transistor si trova veramente nello stato $q = 0$. In altri termini, i transistor sono affidabili.

L'affidabilità dei transistor è un requisito imprescindibile per la costruzione di un elaboratore elettronico. Se venisse a cadere questa proprietà, i computer produrrebbero risultati e comportamenti inattendibili e non servirebbero a granché. In linea teorica, è possibile immaginare dispositivi che siano in grado di assumere un numero anche molto elevato di stati in modo affidabile, ma nella realtà solo i transistor riescono a garantire un livello di affidabilità sufficiente. Allo stato attuale non esistono alternative al transistor nella costruzione dei calcolatori elettronici.

Prefixes for bit and byte multiples

Decimal			Binary		
Value	SI		Value	IEC	JEDEC
1000	k	kilo	1024	Ki kibi	K kilo
1000^2	M	mega	1024^2	Mi mebi	M mega
1000^3	G	giga	1024^3	Gi gibi	G giga
1000^4	T	tera	1024^4	Ti tebi	
1000^5	P	peta	1024^5	Pi pebi	
1000^6	E	exa	1024^6	Ei exbi	
1000^7	Z	zetta	1024^7	Zi zebi	
1000^8	Y	yotta	1024^8	Yi yobi	

Figura 1.1: Unità di misura multipli del byte. La figura è tratta da Wikipedia all'indirizzo web http://en.wikipedia.org/wiki/Byte.

Assemblando insieme n transistor si ottiene una sequenza di dispositivi ognuno dei quali, come detto, può assumere uno degli stati nell'insieme $V = \{0, 1\}$. Ad esempio, una sequenza di 8 transistor potrebbe essere la seguente:

$$01100011$$

Le precedente sequenza è nota come *stringa binaria* e ogni singolo elemento (o cifra) è detto *bit*. Il bit è, dunque, il corrispettivo formale del transistor poiché entrambi possono assumere solo uno tra due possibili valori. Una sequenza di 8 bit è detta *byte*. La sequenza precedente è, pertanto, 1 byte. In Informatica si fa spesso riferimento a multipli del byte per caratterizzare la dimensione di alcuni dispositivi elettronici. I più importanti multipli del byte sono riportati in figura 1.1.

Si noti che le possibili stringhe binarie di lunghezza n sono esattamente

$$2^n$$

essendo le disposizioni con ripetizione di n oggetti estratti da un insieme di 2 elementi (cioè $V = \{0, 1\}$), ognuno dei quali può essere preso più volte. Di conseguenza, il numero di stringhe differenti che è possibile ottenere con $n = 8$ bit è esattamente $2^8 = 256$.

Sequenze di bit o, equivalentemente, di transistor sono utilizzati per memorizzare i dati che devono essere elaborati dal calcolatore. Esempi di dati possono essere:

- valori logici {false, true}
- numeri naturali {0, 1, 2, ...}
- numeri interi {..., -2, -1, 0, 1, 2, ...}
- numeri razionali {- 1/3, ..., -1/127, ..., 0, ..., 1/1208, ..., 3/8, ...}
- caratteri {A, B, C, ..., a, b, c, ..., 0, 1, 2, 3, ...,9, @, §, ...}
- stringhe {...,"Ma a cosa servirà mai questo corso di informatica?", ...}

Attenzione, il calcolatore elettronico non può in alcun modo distinguere se una data sequenza di bit rappresenti un numero naturale, un numero razionale o qualsiasi altro tipo di dato. Questo può essere correttamente determinato solo da chi ha codificato il dato. Il calcolatore elettronico, tuttavia, se opportunamente programmato, può operare correttamente sui dati codificati sotto forma di stringhe binarie. A tal fine, il calcolatore è dotato di opportuni circuiti adatti a operare sulle specifiche rappresentazioni dei dati. Questo vuol dire, ad esempio, che il calcolatore è dotato di circuiti aritmetici adatti a manipolare operandi di tipo intero (cioè numeri interi codificati come stringhe binarie), circuiti aritmetici in grado di manipolare operandi razionali (cioè numeri razionali codificati come stringhe binarie) e così via. Per comprendere come sia possibile, ad esempio, effettuare operazioni aritmetiche su numeri naturali o interi sarà necessario capire come sia possibile codificare questi oggetti in termini di stringhe binarie e studiare gli algoritmi aritmetici fondamentali del sistema di numerazione binario.

1.1 Codifica binaria dei numeri naturali

Il sistema di numerazione binario è in tutto e per tutto simile al sistema di numerazione decimale che siamo abituati a utilizzare, con la differenza che dispone delle sole due cifre 0 e 1 anziché delle solite 10 cifre, da 0 a 9. Ad esempio, in binario si conta esattamente come nel sistema decimale. Come si può osservare in tabella 1.1, è sufficiente iniziare con i numeri a una cifra (in questo caso solo 0 e 1) per poi passare progressivamente ai numeri che richiedono più cifre (due, tre e quattro cifre nell'esempio in tabella). Esiste quindi una relazione biunivoca tra numeri naturali codificati in binario e in decimale per cui, ad esempio, il numero binario 1111 corrisponde al numero naturale 15 espresso nel sistema di numerazione decimale.

Fissato il numero di bit utilizzati, n, abbiamo già visto che possiamo rappresentare 2^n numeri binari differenti e questi variano nel seguente intervallo:

$$[0, 2^n - 1]$$

binario	decimale
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1001	10
1011	11
1100	12
1101	13
1110	14
1111	15
...	...

Tabella 1.1: Contare in binario.

In particolare, si ottiene il valore 0 in corrispondenza della stringa nulla, il valore massimo, $2^n - 1$, in corrispondenza della stringa di tutti 1. Ad esempio, nel caso di $n = 8$, i numeri rappresentabili sono esattamente $n^8 = 256$ e variano da 0 (in corrispondenza della stringa 00000000) a 255 (in corrispondenza della stringa 11111111).

Si osservi, infine, che contare non è sempre il modo migliore per risalire al valore decimale di un numero naturale binario. Non è proprio immediato, ad esempio, capire quale sia il corrispettivo decimale del numero binario 10011100. Si può, in ogni caso, far riferimento agli algoritmi di conversione di base descritti di seguito.

1.1.1 Conversione di un numero naturale da binario a decimale

Si osservi che il sistema di numerazione decimale è *posizionale* poiché, detto $a = a_{n-1}a_{n-2}...a_2a_1a_0 \in \mathbb{N}$ un numero naturale, vale quanto segue:

$$a = a_{n-1}10^{n-1} + a_{n-2}10^{n-2} + ... + a_2 10^2 + a_1 10^1 + a_0 10^0$$

essendo 10 la base del sistema di numerazione decimale. Ad esempio il numero $a = 127$ si può scrivere come segue:

$$127 = 1 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0 = 100 + 20 + 7$$

Allo stesso modo, il sistema di numerazione binario è un sistema di numerazione posizionale poiché, detto $b = b_{n-1}b_{n-2}...b_2b_1b_0 \in \mathbb{N}$ un numero naturale espresso nel sistema di numerazione binario, vale la seguente relazione:

$$b = b_{n-1}10^{n-1} + b_{n-10}10^{n-10} + ... + b_{10}10^{10} + b_1 10^1 + b_0 10^0$$

Si noti con attenzione che questa volta la base e gli esponenti sono rappresentati nel sistema binario per cui 10 non rappresenta il numero dieci, ma il numero binario uno zero, che corrisponde in decimale al numero 2. Esprimendo base ed esponenti nel sistema decimale e svolgendo i calcoli otteniamo il corrispondente valore decimale del numero binario considerato. Ad esempio:

$$10101 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 16 + 4 + 1 = 21$$

Esercizio

Convertire in decimale i seguenti numeri binari:

(1) 10000

(2) 10011

(3) 10011000

(4) 11000101

(5) 11111111

(6) 100001111001100

Esercizio

Sapresti dire a colpo d'occhio, senza eseguire la conversione, quali sono i numeri pari e i numeri dispari dell'esercizio precedente?

1.1.2 Conversione di un numero naturale da decimale a binario

Per la conversione di un numero naturale dal sistema decimale al binario si utilizza un metodo iterativo che sfrutta l'algoritmo della divisione euclidea. La prima divisione si effettua considerando come dividendo il numero decimale da convertite

dividendo	divisore	quoziente	resto
19	2	9	1
9	2	4	1
4	2	2	0
2	2	1	0
1	2	0	1

Tabella 1.2: Esempio di conversione di un numero naturale dalla rappresentazione decimale alla rappresentazione binaria. Il numero binario risultate si ottiene considerando i resti delle divisioni successive presi al contrario, dall'ultimo al primo. Si noti che l'ultima divisione effettuata è quella che ha restituito quoziente nullo. In conclusione, $19_{10} = 10011_2$.

dividendo	divisore	quoziente	resto
219	10	21	9
21	10	2	1
2	10	0	2

Tabella 1.3: Esempio di conversione di un numero naturale dalla rappresentazione decimale alla rappresentazione decimale (non è un errore!). Il numero risultate si ottiene considerando i resti delle divisioni successive presi al contrario, dall'ultimo al primo. Si noti che l'ultima divisione effettuata è quella che ha restituito quoziente nullo. In conclusione, $219_{10} = 219_{10}$.

e come divisore la base del sistema binario (cioè 2). Se il quoziente è diverso da 0 (zero) si effettua un'ulteriore divisione per 2 in cui il nuovo dividendo è il quoziente appena ottenuto. In caso contrario il processo iterativo si conclude e il risultato della conversione è il numero binario che si ottiene considerando i resti delle divisioni effettuate a cominciare dall'ultima. Si consideri, ad esempio, la conversione del numero decimale $a = 19$ nel corrispettivo binario riportata in tabella 1.2. Considerando i resti al contrario, si ottiene il seguente risultato:

$$19_{10} = 10011_2$$

Per verificare la correttezza del procedimento conviene fare la controprova, convertire cioè il numero binario ottenuto in decimale e verificare che questo corrisponda al numero da cui siamo partiti:

$$10011 = 1 \cdot 2^4 + 1 \cdot 2^1 + 1 \cdot 2^0 = 16 + 2 + 1 = 19$$

Per capire il meccanismo che sottende all'algoritmo appena illustrato, effettuiamo una conversione da decimale a decimale come, ad esempio, quella riportata in

Figura 1.2: Esempio di somma tra due numeri naturali espressi in binario. Si noti il meccanismo dei riporti.

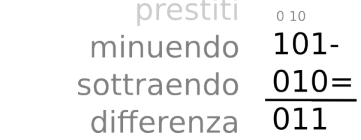

Figura 1.3: Esempio di sottrazione tra due numeri naturali espressi in binario. Si noti il meccanismo dei prestiti.

tabella 1.3. Dovrebbe risultare ora chiaro che ogni divisione per 10 isola nel resto la cifra meno significativa del dividendo. La prima divisione ha quindi come resto le unità, la seconda le decine e così via. Per ricomporre il numero di partenza si devono quindi prendere i resti al contrario, in modo da scrivere il numero partendo dalla cifra più significativa. Allo stesso modo, se invece di dividere per 10 dividiamo per 2, isoliamo nei resti successivi sempre le cifre meno significative dei successivi dividendi. Tuttavia, questa volta, i resti sono espressi nel sistema di numerazione binario (non potendo il resto essere maggiore o uguale al dividendo, che è 2) e, di conseguenza, prendendo i resti al contrario si ottiene l'espressione binaria del numero di partenza.

Esercizio

Convertire in binario i seguenti numeri decimali:

(1) 256

(2) 1973

(3) 20012

(4) 102199

(5) 110001

1.1.3 Operazioni aritmetiche tra numeri naturali binari

Le operazioni aritmetiche si effettuano applicando gli stessi algoritmi utilizzati nel sistema decimale. A titolo d'esempio, si considerino la somma e la sottrazione illustrate rispettivamente nelle figure 1.2 e 1.3.

1.2 Codifica binaria dei numeri interi

Per la codifica dei numeri interi sono state proposte due diverse convenzioni. La prima è nota come rappresentazione in *modulo e segno*, mentre la seconda come rappresentazione in *complemento a due*. Per entrambe è necessario specificare a priori il numero di bit utilizzati. Se non specificato diversamente, utilizzeremo 8 bit per la codifica degli interi. Nei paragrafi successivi sono illustrate le due convenzioni e i metodi di conversione da e verso il sistema decimale.

1.2.1 Codifica binaria dei numeri interi in modulo e segno

Questa convenzione prevede che il primo bit del numero rappresenti il segno, mentre i rimanenti rappresentano il modulo del numero. In particolare, se il primo bit è 0 (zero) si assume che il numero sia positivo mentre se è 1 allora si assume che il numero sia negativo:

0 positivo
1 negativo

Ad esempio, il numero 00001111 rappresenta il numero intero decimale +15 poiché il primo bit (cioè quello più a sinistra) è 0 e i rimanenti bit sono 0001111 (che corrisponde al numero intero 15 - si veda il paragrafo 1.1).

Si noti che la codifica degli interi in modulo e segno presenta alcuni inconvenienti. Tra questi, esiste il problema della doppia rappresentazione dello zero, visto che:

00000000 = +0
10000000 = -0

Questo è uno dei motivi per cui la convenzione utilizzata per la codifica dei numeri interi attualmente utilizzata nella pratica è quella del complemento a due, che è illustrata nel seguito. Poiché lo zero presenta due diverse codifiche, il numero di interi rappresentabile con la convenzione del modulo e segno è:

$$2^n - 1$$

essendo n il numero di bit utilizzati per la codifica. I numeri rappresentabili variano nel seguente intervallo:

$$[-2^{n-1} - 1, 2^{n-1} - 1]$$

Ad esempio, nel caso di $n = 8$ bit, i numeri rappresentabili sono in totale $2^8 - 1 = 255$ (anziché 256) e variano nell'intervallo $[-127, 127]$.

Conversione di un numero intero in modulo e segno da binario a decimale

La conversione da e verso il sistema binario è piuttosto immediata. Per quanto riguarda la conversione dal sistema binario al decimale, è sufficiente effettuare la conversione del modulo del numero utilizzando l'algoritmo di conversione studiato nel paragrafo 1.1 e utilizzare il bit più a sinistra per determinare il segno.

Esercizio

Convertire in decimale i seguenti numeri binari in modulo e segno.

(1) 00000000

(2) 01001101

(3) 11001101

(4) 10000111

(5) 11000100

Conversione di un numero intero in modulo e segno da decimale a binario

Esattamente lo stesso ragionamento si può riproporre per la conversione dal decimale al binario. Si converte il modulo del numero utilizzando la conversione per i naturali utilizzando 7 bit per la codifica e si completano gli otto bit richiesti col bit del segno. Ad esempio, se si deve convertire il numero intero decimale -14, si converte prima 14 in binario con 7 bit utilizzando la codifica dei naturali e si antepone poi il bit 1 poiché il numero che stiamo convertendo è negativo. Il numero binario che codifica -14 in modulo e segno è pertanto 10001110.

Esercizio

Convertire in binario i seguenti numeri decimali utilizzando la convezione del modulo e segno con 8 bit.

(1) 18

(2) 128

(3) 2014

(4) 70111

(5) 0

1.2.2 Codifica binaria dei numeri interi complemento a due

La codifica dei numeri interi in complemento a due riprende, con una piccola ma significativa variazione, la definizione (posizionale) di numero naturale illustrata nel paragrafo 1.1. Detto a un generico numero intero, la rappresentazione in complemento a due è definita secondo la seguente relazione:

$$a = a_{n-1}(-b^{n-1}) + a_{n-2}b^{n-2} + ... + a_2b^2 + a_1b^1 + a_0b^0$$

essendo b la base del sistema di navigazione (2 nel nostro caso) ed n il numero di cifre utilizzate per la codifica del numero. In altri termini, un numero binario in complemento a due prevede il segno negativo per la cifra più significativa.

Si noti che la convenzione del complemento a due non presenta il problema della doppia rappresentazione dello zero (si veda il paragrafo successivo) e, di conseguenza, consente di rappresentare

$$2^n$$

interi differenti. Infatti, le due stringhe 00000000 e 10000000 corrispondono, rispettivamente, ai numeri decimali 0 (zero) e -128.

Infine, i numeri in complemento a due variano nel seguente intervallo:

$$[-2^{n-1}, 2^{n-1} - 1]$$

essendo sempre n il numero di bit utilizzati per la codifica del numero.

Conversione di un numero intero in complemento a due da binario a decimale

La precedente definizione può essere utilizzata per la conversione dal sistema binario al sistema decimale. Ad esempio, il numero binario 11000001 espresso in complemento a due equivale al numero decimale -63. Infatti, applicando la definizione si ha:

$$11000001 = 1 \cdot -(2^7) + 1 \cdot 2^6 + 1 \cdot 2^0 = -63_{10}$$

Esercizio

Convertire in decimale i seguenti numeri binari espressi in complemento a due con 8 bit.

(1) 11111111

(2) 10101010

(3) 01010101

(4) 00000000

(5) 10000000

Conversione di un numero intero in complemento a due da decimale a binario

La conversione di un numero da decimale a binario in complemento a due è leggermente più complessa. Sia a il numero decimale da convertire ed n il numero di bit da utilizzare nella conversione. Se il numero da convertire è positivo, si deve verificare che esso sia rappresentabile con $n-1$ bit, il ché si può fare facilmente considerando la parte intera del logaritmo in base 2 del numero da convertire: $\lfloor log(a) \rfloor$.

Ad esempio, se si deve convertire il numero $a = 123$, allora si valuta subito l'espressione $\lfloor log(a) \rfloor = \lfloor log(123) \rfloor = 7$. Questo vuol dire che 123 è rappresentabile (come numero naturale) con 7 bit, ed essendo $7 = n - 1$, allora il numero 123 può essere convertito in complemento a due con $n = 8$ bit.

Per la conversione vera e propria, si converte il numero attraverso l'algoritmo per i naturali utilizzando $n-1$ bit (7 nel nostro caso particolare) e si completa a sinistra con la cifra 0 (essendo il numero di partenza positivo). Risulta così:

$$123 = 01111011$$

Se il numero da convertire è invece negativo, si verifica innanzitutto che, in modulo, il numero a da convertire sia effettivamente rappresentabile con le n cifre a disposizione. Appurato questo, si calcola quanto è necessario aggiungere al numero negativo in modulo più grande che si può ottenere con n bit in modo da soddisfare la seguente relazione:

$$a = -2^{n-1} + s$$

da cui risulta

$$s = a + 2^{n-1}$$

Si voglia, ad esempio, convertire il numero $a = -3$ in complemento a due con $n = 8$ bit. Innanzitutto si verifica che il numero sia codificabile con le cifre a disposizione, e questo risulta evidente nel nostro caso. Poi si considera la relazione:

$$-3 = -2^7 + s$$

in modo da avere in s la quantità da aggiungere a -2^7 per ottenere il numero desiderato. Risulta:

$$s = 125$$

A questo punto è sufficiente convertire s come un numero naturale utilizzando 7 bit e completare a sinistra con il bit 1:

$$-3 = 11111101$$

Per convincersi del procedimento è sufficiente riconvertire il numero 11111101 in decimale.

Esercizio

Convertire in binario in complemento a 2 con 8 bit i seguenti numeri decimali.

(1) -128

(2) -127

(3) -126

(4) -3

(5) -2

(6) -1

(7) 0

(8) 1

(9) 2

(10) 3

(11) 126

(12) 127

1.3 Codifica dei numeri razionali

I numeri interi sono molto utili se vogliono rappresentare oggetti numerabili. Se però vogliamo misurare qualcosa abbiamo bisogno dei numeri razionali. A tal proposito, è utile introdurre il concetto di precisione. Quando diciamo che una misura é precisa possiamo farlo in due modi: in modo assoluto e in modo relativo. In modo assoluto diremo, ad esempio, che abbiamo misurato le pareti di una stanza con la precisione di un centimetro, in questo caso l'errore massimo é assoluto e vale un centimetro. Una misura viene detta precisa in modo relativo quando ci interessa contenere l'errore al disotto di una certa parte della grandezza da misurare: ad esempio misuriamo la velocità di un automobile con la precisione del 10% se l'automobile va a 150 km all'ora l'errore é di 15 km/h. Se aggiungiamo l'errore di 15 km/h in più otteniamo 165 km/h. Se togliamo l'errore di 15 km/h otteniamo 135 km/h. La cifra che comunque non varia nel primo (sovrastima della velocità) e nel secondo caso (sottostima della velocità) é la prima cifra che viene quindi detta cifra esatta.

1.3.1 Rappresentazione di numeri razionali in virgola mobile

Nella rappresentazione dei numeri razionali in virgola mobile l'informazione relativa alle cifre significative viene archiviata con un gruppo di bit che viene detto mantissa. Il numero di zeri da aggiungere é conservato separatamente e viene detto esponente. Sia la mantissa che l'esponente sono quindi dei numeri interi e vengono rappresentati in complemento a due. Se x è il numero che si vuole rappresentare, si utilizza la seguente convenzione:

$$x = \pm 0.m \cdot B^e$$

dove m è detta mantissa, e è detto esponente, mentre B rappresenta la base del sistema numerico. Ad esempio, il numero 123.45 si rappresenta nel sistema numerico decimale nel seguente modo:

$$123.45 = \pm 0.12345 \cdot 10^3$$

In questo caso la mantissa, m, è 123, l'esponente, e, è 3 e la base, B, è 10 (sistema decimale). Nel caso in cui si scelga il sistema binario e si voglia scrivere il numero 101010000, si ha:

$$101010000 = 0.10101 \cdot 10^{01001}$$

dove $m = 10101_2$, $B = 10_2 = 2_{10}$ ed $e = 01001_{C2} = 9_{10}$.

Il problema di questo tipo di rappresentazione è la perdita di precisione. Supponiamo di operare nel sistema decimale e di voler utilizzare 4 cifre per m e 1 cifra per e. La rappresentazione del numero 123.45 non consente di preservare la seconda cifra decimale, perdendo in precisione:

$$123.45 = 0.1234 \cdot 10^3$$

Negli standard attualmente utilizzati (IEEE 754) i numeri a virgola mobile sono di due tipi:

- a 32 bit - singola precisione: 24 bit per la mantissa e 8 bit per l'esponente
- a 64 bit - doppia precisione: 53 bit per la mantissa e 11 bit per l'esponente

Se vogliamo trasformare la precisione espressa in bit in precisione espressa in numero di cifre decimali abbiamo:

- singola precisione: 6 cifre decimali
- doppia precisione : 15 cifre decimali

Questo risultato si ottiene partendo dal fatto che 10 bit servono per 3 cifre decimali. Quindi 24 bit danno circa 6 cifre decimali. La dimensione massima dei numeri rappresentabili é di 10 ± 38 e di 10 ± 308 che si ottiene dalla precisione in bit sull'esponente.

1.3.2 Rappresentazione di numeri razionali in virgola fissa

Il numero razionale è rappresentato come una coppia di numeri interi: la parte intera e la parte decimale. Ad esempio, se si utilizzano 8 bit sia per la parte intera che per la parte decimale, il numero 12.54 è codificato come segue:

$$(00001100, 00110100)$$

Questo tipo di codifica è stata abbandonata in favore della rappresentazione in virgola mobile illustrata precedentemente.

1.4 Codifica dei caratteri

I caratteri sono quei simboli alfanumerici (alfabetici, numerici, di punteggiatura, ecc.) che possono essere introdotti da una tastiera. Esistono tastiere diverse per ogni paese in quanto alcuni paesi usano tipi di caratteri diversi: ad esempio in Italia si usano i caratteri corrispondenti alle vocali accentate in modo grave o acuto come

à,à,é,è,í,ì,ó,ò,ú,ù. In Francia o Spagna si usa ad esempio la cedilla, cioè il carattere ç, e nelle tastiere di quei paesi é presente un tasto apposito. Se un calcolatore deve funzionare in questi paesi l'unica cosa che verrà sostituita é la tastiera.

La scelta del numero di bit da utilizzare per la codifica dei caratteri è una questione che si discute sin dagli anni 60. Alcuni costruttori suggerivano 6 bit, altri 7. Alla fine si sono affermati codifiche a 8 e 16 bit. Con la codifica a 6 bit (che offre un totale di solo 64 disposizioni) si potevano memorizzare caratteri alfabetici e numerici e i segni di interpunzione ma non si potevano inserire i caratteri minuscoli o i caratteri speciali dei vari paesi. La soluzione a 7 bit con le sue 128 possibilità permette di inserire tutti i tipi di carattere maiuscoli e minuscoli ma non i caratteri dei vari paesi. La soluzione più diffusa attualmente é quella a 8 bit con la quale si possono rappresentare 256 caratteri. Con questa scelta è possibile inserire anche caratteri speciali che servono per funzioni particolari come le comunicazioni o per rappresentare segni grafici speciali.

Ovviamente serve uno standard, cioè un convenzione universalmente riconosciuta, per la codifica dei caratteri. In caso contrario, se alla A un computer associa la codifica 65 (ovviamente in binario) e alla B la codifica 66 mentre un altro computer fa l'opposto, quando trasferiamo un testo da un computer all'altro le A diventano B e viceversa; as esempio BABA diventa ABAB.

Lo standard oggi più diffuso é l'ASCII (American Standard Code for Information Interchange) a 7 bit. Lo standard ASCII non dà purtroppo una definizione precisa dei caratteri dalla disposizione 128 fino alla 255 e vi sono varie possibilità una di queste la codifica ASCII ESTESA a 8 bit (figura 1.4).

Per i caratteri nei moderni sistemi operativi è utilizzata la codifica UNICODE a 16 bit. Il numero di possibili simboli rappresentabili é 65536 e si possono utilizzare anche per rappresentare caratteri ideografici come ad esempio il Kanij dei giapponesi.

Nota. Le cifre 0..9 rappresentate in ASCII o UNICODE sono simboli o caratteri e NON quantità numeriche. Non pertanto possibile usarle per indicare quantità e per le operazioni aritmetiche.

00000000	Null	00100000	Spc	01000000	@	01100000	`
00000001	Start of heading	00100001	!	01000001	A	01100001	a
00000010	Start of text	00100010	"	01000010	B	01100010	b
00000011	End of text	00100011	#	01000011	C	01100011	c
00000100	End of transmit	00100100	$	01000100	D	01100100	d
00000101	Enquiry	00100101	%	01000101	E	01100101	e
00000110	Acknowledge	00100110	&	01000110	F	01100110	f
00000111	Audible bell	00100111	'	01000111	G	01100111	g
00001000	Backspace	00101000	(	01001000	H	01101000	h
00001001	Horizontal tab	00101001	)	01001001	I	01101001	i
00001010	Line feed	00101010	*	01001010	J	01101010	j
00001011	Vertical tab	00101011	+	01001011	K	01101011	k
00001100	Form Feed	00101100	,	01001100	L	01101100	l
00001101	Carriage return	00101101	-	01001101	M	01101101	m
00001110	Shift out	00101110	.	01001110	N	01101110	n
00001111	Shift in	00101111	/	01001111	O	01101111	o
00010000	Data link escape	00110000	0	01010000	P	01110000	p
00010001	Device control 1	00110001	1	01010001	Q	01110001	q
00010010	Device control 2	00110010	2	01010010	R	01110010	r
00010011	Device control 3	00110011	3	01010011	S	01110011	s
00010100	Device control 4	00110100	4	01010100	T	01110100	t
00010101	Neg. acknowledge	00110101	5	01010101	U	01110101	u
00010110	Synchronous idle	00110110	6	01010110	V	01110110	v
00010111	End trans. block	00110111	7	01010111	W	01110111	w
00011000	Cancel	00111000	8	01011000	X	01111000	x
00011001	End of medium	00111001	9	01011001	Y	01111001	y
00011010	Substitution	00111010	:	01011010	Z	01111010	z
00011011	Escape	00111011	;	01011011	[	01111011	{
00011100	File separator	00111100	<	01011100	\	01111100	|
00011101	Group separator	00111101	=	01011101	]	01111101	}
00011110	Record Separator	00111110	>	01011110	^	01111110	~
00011111	Unit separator	00111111	?	01011111	_	01111111	Del

Figura 1.4: Codice ASCII esteso.

Capitolo 2

Architettura del calcolatore

Come già anticipato nel capitolo 1, i calcolatori elettronici sono costruiti assemblando in elementi più complessi, detti *circuiti*, semplici dispositivi elettronici noti con il nome di *transistor*. I circuiti sono, a loro volta, utilizzati per assemblare componenti sempre più complessi come, ad esempio, le memorie.

La struttura e l'organizzazione di tutti i calcolatori moderni si basano su un unico modello teorico dovuto al Matematico americano di origini ungheresi John von Neumann (figura 2.1) e noto come *architettura di von Neumann*. L'architettura di von Neumann è basata sulle seguenti tre assunzioni:

1. Un calcolatore è realizzato sulla base dello schema generale riportato in figura 2.2; questo schema prevede quattro sottosistemi principali (memoria, unità di controllo, unità aritmetico-logica e unità di ingresso-uscita) e un canale condiviso di comunicazione (il bus di sistema).

2. I programmi e i dati sono memorizzati nella memoria centrale dell'elaboratore elettronico come stringhe binarie.

3. Un'istruzione per volta è prelevata dalla memoria per essere decodificata nell'unità di controllo ed eseguita. Questa assunzione caratterizza l'architettura di von Neumann come un'architettura sequenziale.

Nei paragrafi successivi sono brevemente descritti i componenti principali dell'architettura di von Neumann, ad eccezione del sottosistema di ingresso-uscita.

2.1 Il sottosistema memoria (RAM)

La memoria del calcolatore elettronico utilizza una metodologia d'accesso ai dati chiamata *accesso random*. Per tale motivo, la memoria centrale del calcolatore

Figura 2.1: John von Neumann (28 Dicembre 1903 - 8 Febbraio 1957).

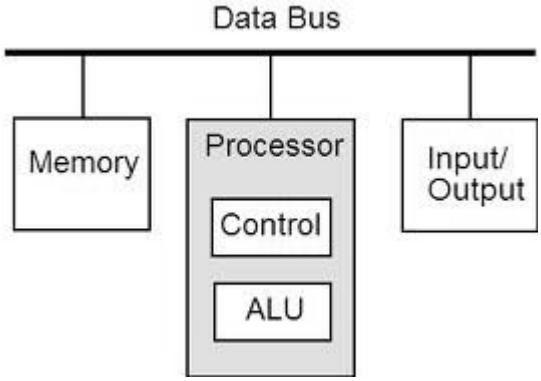

Figura 2.2: Schema generale dell'architettura di von Neumann.

è anche nota come RAM, acronimo di Random Access Memory. La RAM ha le seguenti caratteristiche:

1. E' divisa in unità di dimensione fissa, dette *celle*, ognuna delle quali è identificata da un numero (naturale) univoco, detto *indirizzo*.

2. Gli *accessi* ai dati (o alle istruzioni) in memoria avvengono specificando gli indirizzi delle celle che contengono i dati (o le istruzioni) e possono essere di due tipi: lettura o recupero del dato (*fetch*) e salvataggio o scrittura (*store*).

3. Il tempo richiesto per l'accesso è costante per tutte le celle e non dipende, quindi, dall'indirizzo.

La figura 2.3 illustra uno schema semplificato del sottosistema memoria.

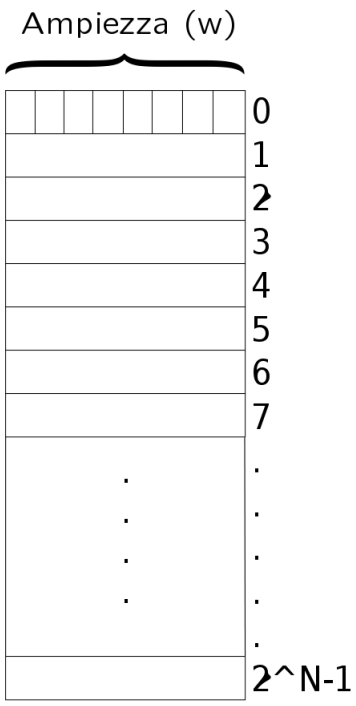

Figura 2.3: Schema generale della memoria RAM.

Lo standard *de facto* nella costruzione delle RAM prevede che l'ampiezza della memoria w, cioè il numero di bit della singola cella, sia di 8 bit, cioè di un byte. Il valore intero più grande che può essere memorizzato in una cella di 8 bit è quindi $2^8 - 1 = 255$. Pertanto, è necessario ricorrere a più celle contigue per memorizzare dati più grandi. Ad esempio:

- 2 o 4 byte (o, equivalentemente, celle di memoria) per memorizzare un numero intero
- 4 o 8 byte (o, equivalentemente, celle di memoria) per memorizzare un numero razionale

Questo implica, in genere, che sono necessari più accessi alla memoria per recuperare dati maggiori di un byte. Questo, ovviamente, penalizza le prestazioni del calcolatore.

Numero di bit N	Celle indirizzabili
16	$2^{16} \equiv 64$ kbyte
32	$2^{32} \equiv 4$ GB
64	$2^{64} \gg 4$ GB

Tabella 2.1: Dimensioni tipiche della memoria nei calcolatori elettronici.

La dimensione massima della memoria, o *spazio d'indirizzamento*, è il numero di celle indirizzabili, cioè 2^N, dove N è il numero di bit riservato alla codifica degli indirizzi. Valori tipici per N sono riportati in tabella 2.1.

Le operazioni base sulla memoria sono *fetch* (cioè recupero) e *store* (cioè immagazzinamento o memorizzazione). Queste operazioni possono essere descritte qualitativamente nel seguente modo:

- *fetch (indirizzo)*
  Recupera (o copia - operazione non distruttiva) il contenuto della cella di memoria all'indirizzo specificato e restituisce tale contenuto come risultato dell'operazione.

- *store (indirizzo, valore)*
  Memorizza il parametro valore nella cella di memoria specificata attraverso il parametro indirizzo. Il contenuto della cella di memoria oggetto dell'operazione viene sovrascritto dal nuovo valore (operazione distruttiva).

Ad esempio, l'operazione

fetch (42)

restituisce il contenuto della cella di memoria di indirizzo 42. In modo simile, l'operazione

store (42, 128)

scrive il valore 128 nella cella di memoria di indirizzo 42.

2.2 Unità Aritmetico-Logica (ALU)

L'Unità Aritmetico-Logica, meglio nota come ALU (Arithmetic Logic Unit), è la componente del calcolatore elettronico che è demandata all'esecuzione delle operazioni matematiche e logiche. Nei computer attuali, l'ALU è localizzata in un chip, il microprocessore, dove trova posto anche l'Unità di Controllo, descritta nella sezione seguente.

L'ALU è composta da 3 elementi fondamentali:

- *registri*;
- *circuiteria*
- *collegamenti*;

In particolare, un registro è una cella di memorizzazione che contiene un operando o il risultato di un'operazione aritmetica o di confronto. Contrariamente alla celle nella memoria centrale (o RAM) del calcolatore, i registri dell'ALU:

- Sono identificati da un designatore di registro del tipo $R0, R1, ..., RK$ oppure del tipo $A, B, ..., K$.
- Sono molto più veloci della RAM, nel senso che le operazioni di lettura e scrittura si effettuano in un tempo più piccolo.
- Hanno un ruolo ben preciso, cioè contenere esclusivamente operandi e non istruzioni.

La circuiteria è quella parte dell'ALU contenente l'hardware demandato all'esecuzione vera e propria delle operazioni aritmetiche e di confronto, mentre i collegamenti collegano appunto i registri ai circuiti aritmetici e logici.

2.3 Unità di Controllo (CU)

Come anticipato pocanzi, l'Unità di Controllo, o CU (Control Unit), è parte integrante del microprocessore insieme all'ALU. Compito della CU è:

1. eseguire il *fethc* (prelievo) dalla memoria RAM dell'istruzione da eseguire;
2. eseguire la *decodifica* dell'istruzione precedentemente prelevata;
3. demandare l'*esecuzione* dell'istruzione decodificata all'hardware competente.

Queste tre operazioni fondamentali vengono ripetute ad elevatissima frequenza senza soluzione di continuità fino al raggiungimento dell'ultima istruzione del programma.

Per comprendere il funzionamento della CU bisogna prima analizzare le caratteristiche delle *istruzioni in linguaggio macchina*. Queste sono generalmente simili a quella illustrata in figura 2.4a. A titolo esemplificativo, adotteremo nel seguito un modello di calcolatore ideale le cui istruzioni macchina sono lunghe esattamente 12 bit, di cui i primi 4 riservati al codice operativo, mentre i rimanenti 8 a un unico indirizzo (figura 2.4b). Il calcolatore semplificato ha, inoltre, un unico registro

Codice Operativo	Operazione	Breve descrizione
0000	LOAD X	Carica il contenuto della cella d'indirizzo X nel registro dell'ALU R
0001	STORE X	Salva il contenuto del registro R dell'ALU nella cella di indirizzo X
0010	CLEAR X	Setta a zero la cella di indirizzo X
0011	ADD X	Somma all'operando nel registro R dell'ALU il dato nella cella di indirizzo X e salva il risultato in R ($R \leftarrow R + X$)
0100	INCREMENT X	Somma 1 al valore contenuto nella cella d'indirizzo X ($X \leftarrow X + 1$)
0101	SUBTRACT X	$R \leftarrow R - X$
0110	DECREMENT X	$X \leftarrow X - 1$
0111	COMPARE X	Confronta X ed R e setta i bit di condizione (LT,EQ,GT); ad esempio, $X < R \Rightarrow$ (LT,EQ,GT) = (1,0,0)
1000	JUMP X	Salta alla cella d'indirizzo X
1001	JUMPGT X	Salta alla cella d'indirizzo X se in confronto GT è 1
1010	JUMPEQ X	Salta alla cella d'indirizzo X se in confronto EQ è 1
1011	JUMPLT X	Salta alla cella d'indirizzo X se in confronto LT è 1
1100	JUMPNEQ X	Salta alla cella d'indirizzo X se in confronto EQ è 0
1101	IN X	Legge un valore e lo scrive nella cella d'indirizzo X
1110	OUT X	Stampa il valore della cella d'indirizzo X
1111	HALT X	Termina l'esecuzione del programma

Tabella 2.2: Set d'istruzioni macchina del calcolatore semplificato utilizzato in questo testo.

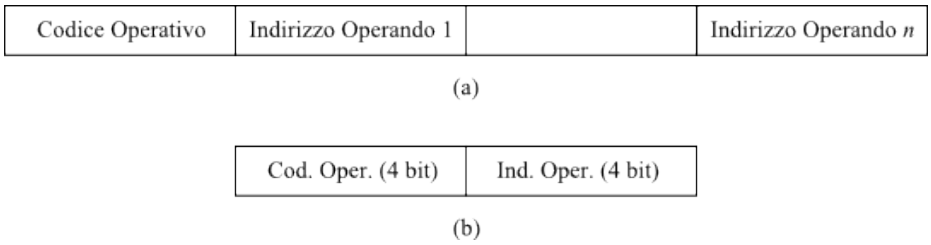

Figura 2.4: Istruzioni in linguaggio macchina: a) schema generale di istruzione con n operandi; b) schema di istruzione ad un operando del colcolatore semplificato utilizzato in questo testo.

nell'ALU al quale ci riferiamo col simbolo R ed è in grado di operare solo su dati di tipo intero.

Il codice operativo è l'identificativo dell'istruzione. Ogni istruzione appartenente all'insieme di istruzioni che la macchina può eseguire (*set di istruzioni della macchina*) ha un codice operativo univoco. Nel seguito considereremo il set d'istruzioni in tabella 2.2. Le classi fondamentali di istruzioni del linguaggio macchina considerato sono illustrate di seguito.

- *Trasferimento dati.* Nel nostro caso si fa riferimento alle istruzioni LOAD e STORE.

- *Aritmetico/Logiche.* Nel nostro caso si fa riferimento alle istruzioni ADD, INCREMENT, SUBTRACT e DECREMENT.

- *Confronto.* Nel nostro caso si fa riferimento all'istruzione COMPARE, che confronta il contenuto della cella della memoria RAM d'indirizzo X con quello del registro R dell'ALU e setta uno speciale registro della CU detto registro dei *bit di condizione*. Questo registro è composto da 3 bit, che chiamiamo LT, EQ e GT. Se l'esito del confronto risulta in una minoranza (cioé $X < R$), allora il bit LT è settato a 1, mentre i rimanenti sono settati a zero. Equivalentemente per gli altri casi. Nel dettaglio si ha:
  $X < R \Rightarrow$ (LT,EQ,GT) = (1,0,0)
  $X = R \Rightarrow$ (LT,EQ,GT) = (0,1,0)
  $X > R \Rightarrow$ (LT,EQ,GT) = (0,0,1)

- *Diramazione.* Nel nostro caso si fa riferimento alle istruzioni JUMP* che consentono di modificare l'esecuzione sequenziale delle istruzioni del programma in modo incondizionato (JUMP) o condizionato (JUMPLT, JUMPEQ, JUMPGT, JUMPNEQ).

Dal punto di vista architetturale, la CU si basa su due registri speciali:

- Program Counter (PC)
- Instruction Register (IR)

e su un circuito di decodifica del codice operativo delle istruzioni.

Il PC contiene l'indirizzo della prossima istruzione da eseguire. All'inizio il suo valore è quindi 0. Il contenuto del PC serve quindi per prelevare (*fetc*) l'istruzione da eseguire, che viene salvata nel registro IR. Questo è necessario perché l'istruzione possa essere analizzata (fase di decodifica) per essere poi correttamente eseguita dall'hardware appropriato (fase d'esecuzione). Una volta terminata l'esecuzione dell'istruzione, il PC viene incrementato automaticamente di 1 e il ciclo di *fetch*, decodifica ed esecuzione ricomincia da capo.

Capitolo 3

Programmazione a basso livello

La programmazione a basso livello è un modello di programmazione grazie al quale è possibile scrivere programmi anche molto complessi, ma che non consente di prescindere in alcun modo dalla conoscenza dell'hardware e delle specifiche tecniche del calcolatore che si sta programmando. A questo modello di programmazione appartengono il linguaggio macchina e il linguaggio Assembly, brevemente discussi nelle sezioni successive.

3.1 Il linguaggio macchina

Nel seguito faremo riferimento al modello semplificato di calcolatore elettronico presentato nel capitolo 4. Esso opera su numeri interi a 8 bit e prevede un'unico registro, R, nell'ALU. Inoltre, le istruzioni della macchina sono anc'esse a 8 bit, 4 bit per i codici operativi e 4 bit per gli indirizzi; il set completo di istruzioni della macchina è presentato in tabella 2.2. Attraverso questo set di istruzioni, è possibile scrivere programmi significativi. Si consideri, a tal proposito, l'esempio seguente.

Esempio

Scrivere un programma che effettui la somma di due operandi A e B, con A = +3 e B = +5, e stampi il risultato in output.

Soluzione

Innanzitutto è necessario generare la rappresentazione binaria degli operandi e immagazzinarla in opportune celle nella memoria del calcolatore. Per quanto appreso nella sezione 1.1, risulta: +3 = 00000011; +5 = 00000101. Oltre ai due operandi, è utile utilizzare un'area di memoria di indirizzo C dove immagazzinare il risultato

dell'operazione. C sarà inizializzato al valore 0 = 00000000. Si noti che si potrebbero inserire tali rappresentazioni nelle prime due celle disponibili nella memoria del calcolatore e cioè le prime tre, aventi indirizzi 0000, 0001 e 0011. Tuttavia questo produrrebbe un, cosi detto, errore logico poiché, per come è progettata l'Unità di Controllo, il contenuto della cella di indirizzo 0000 è la prima a essere prelevata (fetch), decodificata ed eseguita, all'avvio della computazione. Nella prima cella della memoria deve quindi essere collocata un'istruzione e non un operando. Lo stesso discorso vale per il secondo operando. Infatti, poichè l'Unità di Controllo esegue le istruzioni in sequenza, prelevandole una dopo l'altra da celle contigue a partire dalla cella di indirizzo 0000, esso non può essere collocato nelle seconda cella della memoria del calcolatore poichè potrebbe essere prelevato, decodificato ed eseguito come se fosse un'istruzione. Per ovviare a tutto questo, tuttavia, è sufficiente collocare i dati, o meglio le loro rappresentazioni binarie, nelle celle di memoria che seguono la cella di memoria che conterrà l'istruzione HALT, cioè l'istruzione di terminazione del programma (la quale termina il ciclo di fethc, decodifica ed esecusione dell'Unità di Controllo). Poichè non è noto a priori di quante istruzioni sarà composto il programma, questa è un'operazione che dovrà essere effettuata successivamente. Per il momento, quindi, non è possibile conoscere gli indirizzi delle celle di memoria che conterranno gli operandi. Per tale motivo faremo riferimento a tali indirizzi attraverso gli indirizzi simbolici A, B e C, che dovranno successivamente essere risolti (cioè sostituiti con i corrispettivi valori binari). A questo punto, è possibile elencare le istruzioni necessarie a eseguire il calcolo richiesto.

```
IND.  | C.O.  OPER.
------|------------
0000  | 0000  A
0001  | 0011  B
0010  | 0001  C
0011  | 1110  C
0100  | 1111  1111
0101  | 0000  0011
0110  | 0000  0101
0111  | 0000  0000
```

Il listato precedente elenca gli indirizzi delle celle di memoria del calcolatore nella prima colonna. La seconda e la terza colonna elencano rispettivamente il codice operativo e l'operando (o meglio, l'indirizzo dell'operando) dell'istruzione. Una volta definito il listato del programma, sono noti gli indirizzi delle celle che contengono gli operandi A = +3, B = +5 e C = 0: tali indirizzi sono rispettivamente 0101, 0110 e 0111. E' quindi ora possibile risolvere gli indirizzi simbolici A, B e C presenti nel listato precedente.

```
IND.  | C.O.  OPER.  | COMMENTO
------|--------------|---------------------
```

```
0000 | 0000 0101 | R <- A
0001 | 0011 0110 | R <- R + B
0010 | 0001 0111 | C <- R
0011 | 1110 0111 | Stampa  C
0100 | 1111 1111 | Termina programma
0101 | 0000 0011 | +3
0110 | 0000 0101 | +5
0111 | 0000 0000 |  0
```

L'ultimo listato, nel quale sono stati risolti gi indirizzi simbolici, presenta un'ulteriore colonna di commento. La prima istruzione del programma carica il primo operando, A (il cui indirizzo è specificato come parametro del codice operativo), nel registro dell'ALU, R. La seconda istruzione esegue la somma di ciò che è contenuto nel registro R (e cioè A) con l'operando B (il cui indirizzo è specificato come parametro del codice operativo). Come risultato della seconda istruzione, il contenuto della cella B viene sovrascritto col risultato del calcolo A + B. La terza istruzione stampa in output il risultato della somma appena effettuata. La quarta istruzione termina il programma. Terminano il listato le 2 rappresentazioni binarie degli operandi. □

L'esempio precedente mette in evidenza la difficoltà della programmazione in linguaggio macchina. Un ulteriore aspetto che scoraggia l'uso di questo linguaggio è il fatto che una qualunque variazione del numero di righe del programma determina la variazione degli indirizzi degli operandi, poichè questi ultimi dovranno necessariamente essere immagazzinati in celle di memoria differenti. Sarà quindi necessario modificare gli indirizzi di tutti gli operandi, in tutte le istruzioni dove essi compaiono. Tale operazione risulta estremamente pesante per il programmatore ed è soggetta ad errori.

3.2 Assemblatore e linguaggio Assembly

La programmazione in linguaggio macchina presenta alcuni svantaggi. Tra questi:

- Utilizza il formato binario (quasi incomprensibile per gli esseri umani).
- Utilizza indirizzi di memoria numerici.
- Risulta difficile da leggere e modificare.
- Non è possibile creare automaticamente la rappresentazione binaria dei dati.

Per sopperire a queste limitazioni venne progettato il linguaggio di programmazione Assembly, considerevolmente più amichevole del linguaggio macchina. I tre principali vantaggi dell'Assembly rispetto al linguaggio macchina sono i seguenti:

1. Utilizzo di *codici operativi simbolici*.

2. Utilizzo di *indirizzi simbolici*.

3. Presenza di *pseudo-operatori* che forniscono servizi orientati all'utente come la generazione dei dati.

Il linguaggio Assembly del calcolatore semplificato proposto nel capitolo 4 è composto da istruzioni del seguente formato:

[etichetta:] CODICE-OPERATIVO-MNEMONICO INDIRIZZO-SIMBOLICO [//commento]

Attraverso l'etichetta opzionale è possibile definire un identificatore permanente per l'istruzione o il dato. Ad esempio:

INIZIO: LOAD X

associa l'etichetta INIZIO all'istruzione LOAD X di modo che sarà possibile riferirsi successivamente all'istruzione attraverso l'etichetta. Così, se si vuole interrompere la struttura sequenziale d'esecuzione del programma in un suo generico punto per *saltare* all'istruzione LOAD X, sarà possibile farlo con un'altra istruzione del tipo

JUMP INIZIO

Un secondo vantaggio delle etichette è la manutenibilità. Infatti, se si aggiungono o si eliminano istruzioni al programma non sarà necessario modificare alcun campo indirizzo, cosa invece necessaria nel caso si utilizzino indirizzi numerici.

L'ultimo vantaggio dell'Assembly è la generazione automatica dei dati che permette di esprimere i dati (ad esempio numeri interi) in un formato più naturale rispetto alla loro rappresentazione binaria in linguaggio macchina. A tal proposito l'Assembly qui considerato prevede la pseudo-istruzione .DATA per generare la rappresentazione binaria degli interi. Ad esempio

CINQUE: .DATA +5

chiede all'assemblatore di generare la rappresentazione binaria dell'intero +5 etichettando la cella dove tale rappresentazione viene memorizzata con il nome simbolico CINQUE. In questo modo è possibile far riferimento all'intero +5 attraverso l'etichetta, come nell'esempio seguente:

LOAD CINQUE

Si noti che in Assembly non è consentito definire dati prima che siano state specificate tutte le istruzioni del programma. Questo deriva dalla natura sequenziale della macchina di von Neumann che inizia con l'eseguire l'istruzione nella cella d'indirizzo 0, per poi passare all'istruzione nella cella d'indirizzo 1 e così via fino all'istruzione HALT. Se in mezzo alle istruzione venisse inserito un dato, questo sarebbe interpretato come un'istruzione e sarebbe eseguito dal computer generando un errore logico.

Lo schema generale di un'applicazione in linguaggio Assembly qui considerato è il seguente:

```
.BEGIN  // Questa deve essere la prima riga del programma
    .
    .   // Qui vanno le istruzioni in Assembly
    .
 HALT
    .
    .   // Qui vanno le pseudo-istruzioni per la
    .   // generazione dei dati
    .
.END    // Questa deve essere l'ultima linea del programma
```

Esempio

Scrivere un programma Assembly che effettui la somma di due operandi A e B, con A = +3 e B = +5, e stampi il risultato in output. Tale esercizio è stato già risolto in linguaggio macchina nella sezione 3.1.

Soluzione

```
// Qui inizia il codice sorgente del programma

        .BEGIN
        LOAD    A       // R <- A
        ADD     B       // R <- B + R
        STORE   C       // C <- R
        OUT     C       // Stampa C in oputput
        HALT            // Fine del programma

// Qui si definiscono i dati

A:      .DATA   +3
B:      .DATA   +5
C:      .DATA    0

// Questa deve essere l'ultima linea del programma sorgente
```

```
        .END
```

Esempio

Leggere una sequenza di numeri non negativi un numero per volta e calcolare la somma. Al primo numero negativo inserito, stampare la somma degli interi non negativi precedentemente inseriti e terminare il programma.

Soluzione

```
// Qui inizia il codice sorgente del programma
        .BEGIN
        CLEAR   SOMMA
        IN      N

// Le prossime istruzioni controllano se N < 0
AGAIN:  LOAD    ZERO    // R <- 0
        COMPARE N       // N <=> 0 (confronta N con R)
        JUMPLT  NEG

// Siamo qui se N >= 0. Aggiunge N alla somma corrente
        LOAD    SOMMA   // R <- SOMMA
        ADD     N       // R <- R + N
        STORE   SOMMA   // SOMMA <- R

// Legge il prossimo valore in ingresso
        IN      N       // Legge N

// Ora torna indietro e ripete il ciclo
        JUMP    AGAIN   // Salta all'istruzione con etichetta AGAIN

// Si arriva qui solo se N < 0
NEG:    OUT     SOMMA   // Stampa SOMMA
        HALT            // Fine del programma

// Qui si definiscono i dati
SOMMA:  .DATA   0
N:      .DATA   0
ZERO    .DATA   0

// Questa deve essere l'ultima linea del programma sorgente
        .END
```

Esercizio

Scrivere una versione semplificata, utilizzando un numero inferiore di righe di codice, dell'esempio precedente.

Parte II

Programmazione ad alto livello in C++

Capitolo 4

Introduzione alla programmazione in C++

A differenza dei linguaggi studiati in precedenza, il C++ è un linguaggio di programmazione ad alto livello in quanto è caratterizzato da istruzioni e costrutti che sono simili al linguaggio naturale. Inoltre, la programmazione in C++ non richiede la conoscenza specifica dell'hardware. Infatti, da questo punto in poi, programmeremo calcolatori ben più complessi di quello considerato nella prima parte del corso senza, tuttavia, la necessità di conoscere nel dettaglio l'architettura della macchina.

Il C++ si può considerare come l'evoluzione del linguaggio C, progettato e sviluppato da Dennis Ritchie (figura 4.1), uno dei pionieri dell'Informatica moderna. Il C++ è stato invece progettato e originariamente implementato da Bjarne Stroustrup (figura 4.2), professore in Computer Science alla Texas AM University. Analogamente all'Assembly, esiste un programma, detto *compilatore*, che si occupa di convertire il *programma sorgente* scritto in C++ in un *programma eseguibile* in linguaggio macchina. Sarà poi il sistema operativo a gestire la fase di caricamento in memoria del programma eseguibile e di avviare la sua esecuzione.

4.1 Variabili e istruzioni

In C++, un'istruzione è un'espressione che termina con un punto e virgola. Sono, ad esempio, istruzioni del C++:

```
int a;
cin >> a;
cout << "Il valore di a e' " << a;
```

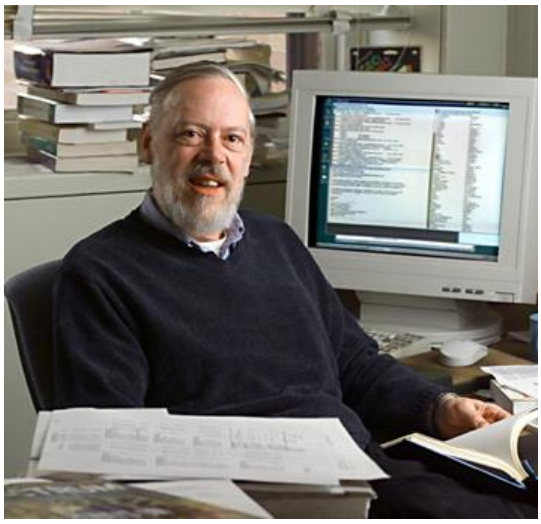

Figura 4.1: Dennis Ritchie (1941-2011), padre del linguaggio C e di UNIX.

La prima è un'istruzione di *dichiarazione*. Il nome simbolico `a`, chiamato anche *variabile*, variabile simbolica, identificatore oppure oggetto, identifica l'area di memoria nella quale sarà immagazzinato un valore numerico di tipo intero. In analogia con il linguaggio Assembly, una variabile può essere considerata come un indirizzo di memoria simbolico. Il nome simbolico `int`, che è una parola riservata del linguaggio, indica proprio che la variabile è di tipo intero[1]

La seconda istruzione è di *input* e permette di acquisire dati. Il nome simbolico `cin` corrisponde al così detto *stream di input*. Uno stream è una sorta di canale di comunicazione all'interno del quale può essere immesso un dato da un'estremità (ingresso) in modo che tale dato fluisca verso l'altra estremità (uscita). Nel caso di `cin`, l'estremità di ingresso corrisponde al terminale utente mentre l'estremità di uscita alla variabile che dovrà contenere il dato inserito. `>>` è l'operatore di inserimento nello stream, o operatore di input. Se l'utente inserisce dal terminale il valore 975, dopo l'esecuzione della seconda istruzione, la variabile `a` conterrà proprio il valore 975.

La terza istruzione è di *output*. Il nome simbolico `cout` corrisponde allo stream di output e tutto ciò che viene inserito nello stream fluisce verso lo standard output, che corrisponde al terminale utente. `<<` è l'operatore di inserimento nello stream di output, o operatore di output. L'operatore scrive sullo stream di output, `cout`, prima la stringa `Il valore di a e'` e poi il valore contenuto nell'area di memoria identificata da `a`, noto anche come valore della variabile. Se `a` vale 975,

[1]Si noti che il C++ è un linguaggio di programmazione *case sensitive* per cui distingue tra lettere minuscole e maiuscole. Ad esempio, l'istruzione INT a = 0;

Figura 4.2: Bjarne Stroustrup, padre del linguaggio C++.

allora l'effetto di questa istruzione è la stampa sul terminale utente della frase `Il valore di a e' 975`.

4.2 Commenti

Come per l'Assembly, anche in C++ è possibile inserire commenti. In C++ è possibile inserire commenti sulla stessa riga attraverso la sequenza di caratteri `//`. Ad esempio:

```
// Questo è un commento su singola riga. Se voglio estendere
// il commento su più righe devo riutilizzare la sequenza "//".
```

In C++ è anche possibile definire commenti su più righe racchiudendo il commento tra i simboli speciali `/*` e `*/`. Ad esempio:

```
/* Questo è un commento su più righe. Se voglio estendere
   il commento su più righe non sono costretto a inserire
   ulteriori sequenze speciali.
*/
```

4.3 Funzioni

Nei programmi C++, le istruzioni sono raggruppate in *funzioni*. Ogni programma, per essere eseguito, deve contenere una funzione speciale detta `main()`. L'esecuzione del programma comincia dalla prima istruzione di `main()` e procede sequenzialmente. `main()` può contenere istruzioni, come quelle viste pocanzi, e richiami

ad altre funzioni. Una funzione consiste di 4 parti di cui le prime tre formano il *prototipo della funzione*:

1. un tipo di ritorno (es. int);
2. il nome della funzione (es. main);
3. una lista di parametri separati da virgole (eventualmente la lista dei parametri può essere vuota);
4. il corpo della funzione (racchiuso tra una coppia di parentesi graffe aperta-chiusa).

Un esempio di funzione `main()` potrebbe essere il seguente:

```
    int main()
{
    // Istruzioni semplici
    int a;
    cin >> a;
    cout << "Il valore di a e' " << a;

    // Richiami ad altre funzioni
    readFromFile();
    extractInfo();
    saveOnFile();
    return 0;
}
```

L'istruzione `return` è un'istruzione predefinita del linguaggio che fornisce un metodo per terminare l'esecuzione della funzione ritornando eventualmente un valore. Nell'esempio precedente, `return 0` determina la fine dell'esecuzione della funzione `main`, e quindi del programma, e ritorna il valore 0 (in questo caso al sistema operativo, che si è occupato di caricare il programma in memoria e di avviarne l'esecuzione). Si noti che il valore 0 è un intero, e questo è in accordo con il tipo di ritorno `int` della funzione `main()`. Il valore 0, per convenzione, indica per convenzione la corretta terminazione del programma.

Perché il programma precedente sia eseguibile, è necessario definire le funzioni chiamate nel `main()`. Si noti che queste devono essere definite prima della funzione `main()` che ne fa uso, per motivi di *visibilità* (o *scope*, in inglese). Un'implementazione banale di esempio potrebbe essere la seguente:

```
void readFromFile() { cout << "readFromFile()\n" }
void extractInfo() { cout << "extractInfo()\n" }
void saveOnFile() { cout << "saveOnFile()\n" }

int main()
{
    // Istruzioni semplici
    int a;
    cin >> a;
    cout << "Il valore di a e' " << a;

    // Richiami ad altre funzioni
    readFromFile();
    extractInfo();
    saveOnFile();
    return 0;
}
```

Il tipo di ritorno `void` indica che la funzione non ritorna alcun valore, mentre il carattere speciale `\n` determina il ritorno a capo del cursore sul terminale utente dopo la stampa. In realtà, è anche possibile collocare la definizione delle funzioni dopo la loro invocazione. In quest'ultimo caso, tuttavia, sarà necessario premettere, prima dell'invocazione, il prototipo, come nel seguente esempio:

```
void readFromFile(); //Prototipo della funzione readFromFile()
void extractInfo(); //Prototipo della funzione extractInfo()
void saveOnFile(); //Prototipo della funzione saveOnFile()

int main()
{
    // Istruzioni semplici
    int a;
    cin >> a;
    cout << "Il valore di a e' " << a;

    // Richiami ad altre funzioni
    readFromFile();
    extractInfo();
    saveOnFile();
    return 0;
}
```

```
void readFromFile() { cout << "readFromFile()\n" }
void extractInfo() { cout << "extractInfo()\n" }
void saveOnFile() { cout << "saveOnFile()\n" }
```

La necessità di premettere il prototipo delle funzioni deriva dal fatto che il compilatore C++ necessita di conoscere a priori le istruzioni che incontra all'interno del programma. Alcune istruzioni sono note a priori, cioè le parole riservate del linguaggio come `main` o `return`, mentre altre vanno necessariamente dichiarate prima che vengano usate. Questo è proprio il caso delle funzioni definite dal programmatore, così come il nome delle variabili.

4.4 Direttive al preprocessore e *namespace*

In realtà, il precedente programma non è ancora corretto poiché manca la definizione dell'istruzione di output, che non fa parte propriamente del linguaggio. Le definizioni di `cout` e `<<` sono tuttavia incluse nel file di intestazione `iostream`, che fa parte della *libreria standard* del C++. La versione corretta del programma è, dunque, la seguente:

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

void readFromFile(); //Prototipo della funzione readFromFile()
void extractInfo(); //Prototipo della funzione extractInfo()
void saveOnFile(); //Prototipo della funzione saveOnFile()

int main()
{
    // Istruzioni semplici
    int a;
    cin >> a;
    cout << "Il valore di a e' " << a;

     // Richiami ad altre funzioni
    readFromFile();
    extractInfo();
    saveOnFile();
    return 0;
}
```

```
void readFromFile() { cout << "readFromFile()\n" }
void extractInfo() { cout << "extractInfo()\n" }
void saveOnFile() { cout << "saveOnFile()\n" }
```

Attraverso la direttiva `#include`, il contenuto del file `iostream` è inserito nel programma e, con esso, la definizione di `cout` e dell'operatore di inserimento nello stream, `<<`. Nello specifico, `#include` è una *direttiva al preprocessore*, un programma che viene eseguito prima del compilatore e che si occupa di eseguire operazioni preliminari, proprio come inserire il contenuto di file specificati all'interno del programma che si sta scrivendo.

L'istruzione `using namespace std;` è una *direttiva d'uso*. I nomi della libreria standard sono infatti incapsulati nello *spazio di nomi*, o *namespace*, standard `std`. La direttiva d'uso informa il compilatore che si intende far riferimento ai nomi inclusi nel namespace specificato. Si noti che la direttiva `using namespace std;` rende disponibili tutti i nomi presenti nel namespace std, la qual cosa, in generale, può risultare eccessiva. Se si vogliono rendere disponibili solo alcuni nomi del namespace, è possibile utilizzare le direttive specifiche, come nel seguente esempio nel quale si rendono visibili esclusivamente i nomi `cin` e `cout`.

```
#include <iostream>
using std::cin;
using std::cout;
```

Esiste, infine, la possibilità di evitare l'utilizzo della direttive d'uso premettendo al nome incluso nel namespace il nome del namespace stesso seguito da una coppia di due punti. Ad esempio, le istruzioni seguenti consentono l'impiego dei nomi `cin` e `cout` senza la necessità di alcuna direttiva d'uso.

```
std::cin >> a;
std::cout << a;
```

4.5 Tipi fondamentali del linguaggio

Il C++ fornisce un insieme di tipi di dato predefiniti, elencati in tabella 4.1, grazie ai quali è possibile definire alcuni tipi base di variabili quali i caratteri, i numeri interi e i numeri in virgola mobile. Di seguito alcuni esempi di definizione di variabili che utilizzano i tipi base del linguaggio:

```
    bool ipotesi_1 = false;  /* definizione di variabile booleana
                                con valore iniziale false */
    bool ipotesi_2 = true;   /* definizione di variabile booleana
```

```
                              con valore iniziale true */
bool ipotesi_3 = 0;       /* definizione di variabile booleana
                              con valore iniziale 0 (false) */
bool ipotesi_4 = 1;       /* definizione di variabile booleana
                              con valore iniziale 1 (true) */
char c1 = 'a';            /* definizione di variabile carattere
                              (8 bit) con valore iniziale 'a', che
                              corrisponde al valore numerico 97 */
char c2 = 97;             /* definizione di variabile carattere
                              (8 bit) con valore iniziale 97, che
                              corrisponde al carattere
                              ASCII esteso 'a'  */
short a = 0;              /* definizione di variabile intera
                              (16 bit) con valore iniziale 0 */
short int a = 0;          /* definizione equivalente alla precedente
                              short a = 0; */
int a = -374;              /* definizione di variabile intera
                              (32 bit) con valore iniziale 0 */
float x1 = 0.887;         /* definizione di variabile razionale
                              in singola precisione (32 bit) con
                              valore iniziale 0 */
double epsilon = 1.0e-6; /* definizione di variabile
                              razionale in singola precisione
                              (64 bit) con valore iniziale 0 */
```

I tipi interi `short` e `int` prevedono anche l'opzione *senza segno*, utile per rappresentare numeri nell'insieme dei Naturali. Ad esempio:

```
unsigned short lato = 5; /* definizione di variabile intera
                            senza segno a 16 bit con valore
                            iniziale 0 */
unsigned int a = 127;    /* definizione di variabile intera
                            senza segno a 32 bit con valore
                            iniziale 0 */
```

Per il tipo `char`, che è di per sè senza segno, esiste invece la variante `signed`. In tal modo, poichè il tipo `char` varia nell'intervallo $[0, 2^8 - 1]$, il tipo `signed char` varia nel intervallo $[-2^7, 2^7 - 1]$.

Per conoscere i range di variazione dei tipi predefiniti è possibile utilizzare le macro definite nel file header `<climits>`. Ad esempio, il seguente programma stampa i limiti del tipo `int`:

Tipo di dato	bit utilizzati dal compilatore `g++` 4.7.2	Dati rappresentati
`bool`	8	valori booleani `false` (0) e `true` (1)
`char`	8	caratteri della tabella ASCII estesa o numeri naturali (interi senza segno)
`short`	16	numeri interi
`int`	32	numeri interi
`float`	32	numeri razionali in singola precisione
`double`	64	numeri razionali in doppia precisione

Tabella 4.1: Tipi base del linguaggio C++.

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

int main()
{
cout << "L'intervallo dei valori rappresentabili "
 << "con il tipo int e' ["
 << INT_MIN
 << ", "
 << INT_MAX
 << "]\n";
return 0;
}
```

Allo stesso modo, è possibile conoscere i limiti dei tipi razionali facendo riferimento all'header file `<cfloat>`. Per la lista completa delle macro definite nei due header file `<climits>` e `<cfloat>` è possibile consultare la documentazione del proprio compilatore.

4.5.1 Il tipo string

Il tipo `string` non è propriamente un tipo base del linguaggio poiché fa parte della libreria standar del C++ ed è costruito sulla base dei tipi fondamentali. Attraverso il tipo `string` è possibile definire stringhe, cioè sequenze di caratteri. Per utilizzare il tipo `string` è necessario includere l'header `<string>`.

```
#include <iostream>
#include <string>
```

```
using std::cout;
using std::string;

int main()
{
string str1 = "La mia prima stringa";
string str2 = "la mia seconda stringa!";
string str3 = str1 + " concatenata con " + str2;

cout << str1 << "\n";
cout << str2 << "\n";
cout << str3 << "\n";

return 0;
}
```

Come è possibile notare, l'operatore `+` è utilizzato per la concatenazione delle stringhe. Inoltre, nell'esempio precedente si è fatto uso delle direttive di definizione specifiche `using std::cout;` e `using std::string;` anziché della direttiva più gnerale `using namespace std;`.

4.6 Costanti

Il C++ mette a disposizione il *qualificatore* `const` che, premesso alla definizione di una variabile, la rende costante, cioè di sola lettura. Ad esempio, l'istruzione

```
const double PI = 3.14159265359;
```

definisce la costante π. Così, il valore di `PI` potrà essere letto ma non modificato, visto che sarebbe un errore logico modificarne il valore. Ogni tentativo di modificare il valore di una costante produrrà un errore a tempo di compilazione.

Oltre al qualificatore `const`, in C++ è possibile definire costanti in stile C attraverso l'uso della direttiva di precompilazione `define`. Ad esempio, l'espressione

```
#define PI 3.14159265359
```

definisce il nome PI associandogli il valore 3.14159265359. Nella fase di precompilazione, che precede la fase di compilazione vera e propria, tutte le occorrenze di PI saranno sostituite col la costante numerica 3.14159265359.

Operatore	Significato	Operandi	Esempio
+	Somma	Interi	c = a + b;
-	Sottrazione	Interi	c = a - b;
*	Moltiplicazione	Interi	c = a * b;
/	Divisione	Interi e razionali	d = a / b;
%	Resto della divisione	Interi	r = a % b;

Tabella 4.2: Operatori aritmetici del linguaggio C++.

4.7 Operatori aritmetici e di confronto

Il C++ mette a disposizione un insieme di operatori aritmetici e di confronto predefiniti. I principali operatori aritmetici sono riportati in tabella 4.2 mentre i principali operatori di confronto in tabella 4.3. Si noti, dagli esempi riportati nelle due tabelle, che a differenza del linguaggio Assembly, il programmatore non deve provvedere esplicitamente al movimento degli operandi dalla memoria all'ALU e viceversa.

Si noti inoltre che l'operatore di confronto per uguaglianza è rappresentato dalla sequenza `==`. Vista la somiglianza, tale operatore di confronto può essere facilmente confuso con l'operatore di assegnamento `=`. Prestare molta attenzione a questo "trabocchetto" poiché utilizzare l'operatore di assegnamento laddove dovrebbe essere usato quello di confronto può dar luogo a errori logici, che sono in genere i più difficili da risolvere. Infatti, così come l'espressione di confronto `a == b`, una volta valutata, restituisce un valore di verità `true` o `false`, anche l'istruzione di assegnamento `a = b`, una volta eseguita, ritorna un valore di verità: `true`, se l'assegnamento è andato a buon fine, `false` in caso contrario.

4.8 Operatori logici, espressioni logiche e loro valutazione

Come abbiamo visto, espressioni del tipo `a == b`, oppure `count >= n`, restituiscono un valore di verità `true` o `false`. Il C++ mette a disposizione alcuni operatori (o connettivi) logici, l'`AND`, l'`OR` e il l'`NOT`, grazie ai quali è possibile combinare più espressioni al fine di costruire espressioni logiche più complesse. In C++, i simboli che identificano i suddetti operatori sono `&&`, `||` e `!`, rispettivamente per l'`AND`, l'`OR` e il l'`NOT`.

L'`AND` è un operatore binario, nel senso che realizza un'espressione logica complessa a partire da due espressioni logiche. Ad esempio:

```
(a == b) && (count >= n)
```

Operatore	Significato	Commento	Esempio
`==`	uguale a	restituisce `true` se `a` e `b` sono uguali, `false` altrimenti	`a == b`
`!=`	diverso da	restituisce `true` se `a` e `b` sono diversi, `false` altrimenti	`a != b`
`<`	minore di	restituisce `true` se `a` è minore di `b`, `false` altrimenti	`a < b`
`>`	maggiore di	restituisce `true` se `a` è maggiore di `b`, `false` altrimenti	`a > b`
`<=`	minore o uguale a	restituisce `true` se `a` è minore o uguale di `b`, `false` altrimenti	`a <= b`
`>=`	maggiore o uguale a	restituisce `true` se `a` è maggiore o uguale di `b`, `false` altrimenti	`a >= b`

Tabella 4.3: Operatori di confronto del linguaggio C++.

In questo caso, per definizione di `AND`, l'espressione è vera nel caso in cui entrambe le sotto-espressioni sono vere e cioè `a` è davvero uguale a `b` e `count` è davvero maggiore o uguale ad `n`.

Se le precedenti espressioni sono combinate in `OR`, come nel seguente esempio:

```
(a == b) || (count >= n)
```

allora l'espressione è vera nel caso in cui almeno una delle sotto-espressioni è vera e cioè `a` è uguale a `b`, oppure `count` è maggiore o uguale ad `n`, oppure `a` è uguale a `b` e contemporaneamente `count` è maggiore o uguale ad `n`. Si noti che, come l'`AND`, anche l'`OR` è un operatore binario.

L'operatore `NOT`, invece, è un operatore unario. Il `NOT` inverte il valore di verità dell'espressione al quale è applicato. Ad esempio, l'espressione

```
!(a == b)
```

è vera quando l'espressione `(a == b)` è falsa ed è falsa quando `(a == b)` è vera. Quanto detto finora può essere riassunto nelle così dette *tabelle di verità* dei tre operatori logici `AND` e `OR` (tabella 4.4) e `NOT` (tabella 4.5).

4.8.1 Precedenza

Grazie agli operatori logici, è possibile costruire espressioni anche molto complesse. Ad esempio, si consideri la seguente espressione:

```
!( ( (a == b) || (count >= n) ) && !(a == b) )
```

A	B	(A && B)	(A || B)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Tabella 4.4: Tabella di verita per gli operatori logici AND e OR. Si noti che sono stati usati i simboli 0 e 1 rispettivamente al posto di false e true e i simboli A e B al posto di espressioni del tipo (a == b).

A	!A
0	1
1	0

Tabella 4.5: Tabella di verita per l'operatore logico NOT. Si noti che sono stati usati i simboli 0 e 1 rispettivamente al posto di false e true e il simbolo A al posto di espressioni del tipo (a == b).

L'uso delle parentesi definisce le precedenze. In questo caso vengono valutate le sotto-espressioni nel seguente ordine:

```
1) (a == b)
1) (count >= n)

2) (a == b) || (count >= n)
2) !(a == b)

3) ((a == b) || (count >= n)) && !(a == b)

4) !(((a == b) || (count >= n)) && !(a == b))
```

L'uso delle parentesi è fondamentale. Infatti, senza di esse la precedente espressione sarebbe stata la seguente:

```
!(a == b) || (count >= n) && !(a == b)
```

e l'ordine di valutazione sarebbe diventato:

```
1) (a == b)
1) (count >= n)

2) !(a == b) // che compare una prima volta in testa
             // e una seconda volta in coda
```

Livello di precedenza	Operatore	Nome operatore
1	`!`	NOT logico
2	`*`	moltiplicazione
2	`/`	divisione
2	`%`	resto della divisione tra interi
3	`+`	somma
3	`-`	sottrazione
4	`<`	minore di
4	`<=`	minore o uguale di
4	`>`	maggiore di
4	`>=`	maggiore o uguale di
5	`==`	uguale a
5	`!=`	diverso da
6	`&&`	AND logico
7	`||`	OR logico
8	`=`	assegnamento

Tabella 4.6: Precedenza degli operatori in C++.

```
3) (count >= n) && !(a == b)

4) !(a == b) || (count >= n) && !(a == b)
```

Com'è chiaro, la definizione delle precedenze può cambiare in modo significativo un'espressione logica. La lista completa delle precedenze degli operatori del C++ è illustrata in tabella 4.6.

4.8.2 Valutazione di espressioni logiche

Grazie agli operatori logici, è possibile costruire espressioni anche molto complesse. Ad esempio, consideriamo ancora l'espressione definita prima:

```
!(( (a == b) || (count >= n) ) && !(a == b) )
```

Per valutare quali siano i casi in cui l'espressione è vera, si può costruire la tabella di verità dell'espressione stessa. Se con A indichiamo `(a == b)` e con B `(count >= n)`, la precedente espressione diventa:

```
!(( A || B ) && !A )
```

e la sua tabella di verità è quella in tabella 4.7. Si noti che l'espressione logica risulta vera nei seguenti casi:

A	B	!A	(A || B)	(A || B) && !A	!((A || B) && !A)
0	0	1	0	0	1
0	1	1	1	1	0
1	0	0	1	0	1
1	1	0	1	0	1

Tabella 4.7: Esempio di valutazione di un'espressione logica con la tabella di verita.

- A e B sono entrambe false o, equivalentemente, `(!A && !B)`
- A è vera e B è falsa o, equivalentemente, `(A && !B)`
- A e B sono entrambe vere o, equivalentemente, `(A && B)`

Possiamo quindi scrivere la soluzione generale come:

```
(!A && !B) || (A && !B) || (A && B)
```

È ancora possibile ridurre la precedente espressione ricorrendo alle proprietà dell'*algebra di Boole*, le più importanti delle quali sono elencate in tabella 4.8. Sfruttando la proprietà distributiva di `AND` rispetto a `OR` si ha, infatti:

```
(!A && !B) || (A && (B || !B))
```

E ancora, grazie alla legge dell'elemento 1, sia ha:

```
(!A && !B) || (A && 1) = (!A && !B) || A
```

Che è equivalente alla soluzione originaria in `OR`: `(!A && !B) || (A && !B) || (A && B)`.

Esistono espressioni sempre vere o sempre false. Le prime sono dette *tautologie*, le seconde *negazioni*. Un esempio di tautologia è l'espressione

```
A || !A
```

mentre un esempio di negazione è l'espressione

```
A && !A
```

Esercizio

Valutare i casi in cui le seguenti espressioni logiche sono vere:

(1) `A && B || A && !B`

(2) `A || !B && A || !(A && B)`

(3) `!A && B || A && C`

(4) `(B || !C) && (A || !C)`

(5) `(A && B) || (!A && !B)`

Proprietà	Espressione di equivalenza
Prima legge di De Morgan	`!(A && B) = !A || !B`
Seconda legge di De Morgan	`!(A || B) = !A && !B`
Associativa OR	`A || (B || C) = (A || B) || C`
Associativa AND	`A && (B && C) = (A && B) && C`
Commutativa OR	`A || B = B || A`
Commutativa AND	`A && B = B && A`
Distributiva di AND rispetto a OR	`A && (B || C) = A && B || A && C`
Distributiva di OR rispetto a AND	`A || B && C = (A || B) and (A || C)`
Assorbimento (A assorbe B)	`A || A && B = A`
Legge dell'elemento 1	`!A || A = 1`

Tabella 4.8: Alcune delle principali proprietà dell'algebra di Boole.

4.9 Compilazione ed esecuzione con g++

Una volta scritto il programma attraverso un editor di testo, esso dovrà essere salvato su file, preferibilmente con estensione `.cpp`, che identifica appunto i file sorgenti in C++. Altre possibili estensioni per i file sorgenti C++ sono: `.C`, `CC` e `cxx`.

Nel seguito, per la compilazione dei file sorgenti, faremo riferimento al compilatore in ambiente GNU/Linux. Supponiamo che il precedente programma sia stato salvato nel file `my_first_program.cpp`. Perché esso possa essere compilato, sarà sufficiente eseguire il seguente comando da terminale:

```
g++ my_first_program.cpp
```

Se si vuole attribuire un nome più significativo all'eseguibile, è sufficiente ricorrere all'opzione `-o` di g+:

```
g++ my_first_program.cpp -o my_fisrt_program
```

Se il sorgente contiene degli errori di sintassi, essi saranno segnalati. In caso contrario, il compilatore genererà il file eseguibile `a.out`. Si noti che, se il sorgente presenta errori logici, essi non verranno segnalati e il compilatore genererà comunque un file eseguibile. Gli errori logici sono, in genere, i più difficili da risolvere. Il caricamento in memoria e l'avvio dell'esecuzione del programma sono a carico del sistema operativo. Sempre da terminale, è sufficiente eseguire il comando:

```
./my_fisrt_program
```

L'operatore `./` è caratteristico del sistema operativo Linux. In Linux, è possibile eseguire applicazioni sono da alcune directory predefinite, ad esempio `/bin`, `/sbin` o `/usr/local/bin`. Pertantom se la directory in cui è si trova l'eseguibile `my_fisrt_program` non è tra quelle, è necessario informare il sistema operativo che si sta forzando il caricamento e la successiva esecuzione dal un'altra directory, e questo avviene proprio attraverso l'operatore `./`. In generale, dunque, non è sufficiente digitare esclusivamente il nome dell'eseguibile e premere il tasto invio perché il programma venga realmente eseguito.

Capitolo 5

Programmazione strutturata in C++

Il teorema di Böhm e Jacopini, pubblicato nel 1966 sulla prestigiosa rivista *Communications of the ACM*[1] asserisce che qualunque algoritmo può essere implementato utilizzando esclusivamente tre strutture:

- la struttura *sequenziale*;
- la struttura, o istruzione, di *selezione*;
- la struttura, o istruzione, di *iterazione*.

La struttura sequenziale è quella che caratterizza i linguaggi di programmazione visti fino a ora in questo corso e prevede, per l'appunto, l'esecuzione delle istruzioni del programma una di seguito all'altra nell'ordine in cui sono state scritte. La selezione è un'istruzione speciale che consente di compiere una scelta, cioè eseguire un blocco di istruzioni piuttosto che un altro, in funzione del valore di verità di una data condizione. L'iterazione è anch'essa un'istruzione speciale che consente di ripetere un blocco di istruzioni fintanto che una data condizione risulta vera.

Il teorema di Böhm e Jacopini costituisce un risultato teorico molto importante che cambiò radicalmente il paradigma di sviluppo dei programmi, precedentemente basati su istruzioni del tipo `jump` e `goto`.

[1] Corrado Böhm and Giuseppe Jacopini, 1966. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of the ACM* 9 (5): 366-371. DOI:10.1145/355592.365646

5.1 La struttura di selezione

In C++, l'istruzione di selezione è l'`if-else`. La sintassi dell'istruzione è la seguente:

```
if (condizione)
    blocco_istruzioni_ramo_if
else
    blocco_istruzioni_ramo_else
```

I nomi `if` ed `else` sono parole riservate del linguaggio C++. La condizione tra parentesi è una qualsiasi espressione logica; essa può assumere quindi i valori di verità `true` o `false`. Un esempio può essere: `(a > b && count < n)`. In questo esempio, vengono valutate preventivamente i confronti `a > b` e `count < n`, ognuno dei quali ritorna un valore di verità. Successivamente, tali valori di verità sono combinati in AND logico, dando luogo a un'unico valore logico `true` oppure `false`. Si noti, che l'espressione logica più semplice possibile è quella costituita da un'unica variabile boolena. Per tale motivo, se `ben_posto` è una variabile booleana, è possibile utilizzarla direttamente come condizione dell'`if`. Se la condizione è vera, viene eseguito il blocco di istruzioni `blocco_istruzioni_ramo_if`, in caso contrario viene eseguito il blocco di istruzioni `blocco_istruzioni_ramo_else`. Per blocco di istruzioni s'intende un'unica istruzione oppure una sequenza di istruzioni racchiuse tra parentesi graffe.

Non sempre, nel caso in cui la condizione risulti falsa, è necessario eseguire un'azione. In tal caso, si può far riferimento alla variante dell'istruzione di selezione priva del ramo `else`. La sua sintassi è la seguente:

```
if (condizione)
    blocco_istruzioni_ramo_if
```

Esempio

Leggere tre interi da input e verificare se costituiscono i lati di un triangolo, ricordando che un triangolo è ben posto se, comunque preso un lato, la sua misura è minore della somma delle misure dei rimanenti due lati.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
```

```
    int a, b, c; // Dichiarazione multipla sulla stessa riga

    cout << "Inserisci tre interi: ";
    cin >> a >> b >> c;

    if ( a < b + c && b < a + c && c < a + b )
        cout << "Il triangolo è bene posto." << "\n";
    else
        cout << "Il triangolo NON è bene posto." << "\n";

    return 0;
}
```

Esempio

Scrivere un programma C++ che legga tre interi da input e determini e stampi il massimo su output.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main()
{
    int a, b, c;  /* dichiarazione di più variabili sulla
                     stessa riga grazie all'operatore ,
                  */

    cout << "Inserisci tre interi: ";
    cin >> a >> b >> c;

    if ( a >= b && a >= c )
        cout << "Il massimo è" << a << endl;
    else
        if ( b >= a && b >= c )
            cout << "Il massimo è" << b << endl;
        else
            cout << "Il massimo è" << c << endl;

    return 0;
}
```

Esercizio

Leggere tre interi da input e verificare se costituiscono i lati di un triangolo facendo uso di una variabile booleana `ben_posto`. Inizializzare tale variabile a `true` (ipotesi ottimistica; si ipotizza cioè che i valori che saranno inseriti costituiscono effettivamente le misure dei lati di un triangolo) e porla uguale a `false` nel caso in cui si scopra che i tre lati non formano un triangolo (tentativo di smontare l'ipotesi).

Esercizio

Leggere tre interi da input, verificare se costituiscono i lati di un triangolo e, in caso affermativo, determinare che tipo di triangolo è (equilatero, isoscele o scaleno).

5.2 La struttura di iterazione

In C++, l'istruzione di iterazione è il ciclo `while`. La sintassi dell'istruzione è la seguente:

```
while (condizione_di_continuazione)
    blocco_istruzioni
```

Il nome `while` è una parola riservata del linguaggio C++. Come nel caso dell'`if`, la condizione tra parentesi tonde è una qualsiasi espressione logica. Fintanto che la `condizione_di_continuazuione` del ciclo è vera, viene eseguito il blocco di istruzioni `blocco_istruzioni`; non appena la condizione diventa falsa il blocco di istruzioni non viene più eseguito e si passa all'esecuzione dell'istruzione che segue l'istruzione `wile`.

Esempio

Leggere da input una sequenza di interi maggiori di zero, calcolare e stampare la somma in output. La sequenza è terminata dal *valore sentinella* zero.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
    int n;
    int somma = 0; // somma dev'essere necessariamente posta a zero
```

```
    cout << "Inserisci un intero: ";
    cin >> n;

    while ( n > 0 )
    {
        somma = somma + n;

        cout << "Inserisci un intero: ";
        cin >> n;
    }

    cout << "La somma è " << somma << "\n";

    return 0;
}
```

Esercizio

Leggere da input una sequenza di interi maggiori di zero, calcolare e stampare il massimo e la somma in output. La sequenza è terminata dal *valore sentinella* zero.

Esercizio

Leggere da input una sequenza di `N` interi, calcolare e stampare la somma in output. Il valore dell'intero `N` dovrà essere letto da input e dovrà essere maggiore di zero. Nel caso in cui `N` risultasse minore o uguale a zero, dovrà essere chiesto l'inserimento di un nuovo valore e la richiesta dovrà essere reiterata in caso di ulteriori errori. `N`, in gergo informatico, è detto *contatore* e indica il numero di iterazioni che dovranno essere eseguite.

Esercizio

Leggere da input una sequenza di interi maggiori di zero, contare quante volte il valore inserito è maggiore del valore inserito precedentemente e stampare il risultato in output. La sequenza è terminata dal *valore sentinella* zero.

5.3 Varianti dell'istruzione di selezione

5.3.1 L'operatore condizionale

L'operatore condizionale rappresenta un'alternativa per semplici istruzioni `if-else`. As esempio, anziché scrivere:

```
bool is_equal;
if ( a == b)
    is_equal = true;
else
    is_equal = false;
```

si può scrivere:

```
bool is_equal = (a == b) ? true : false;
```

L'operatore condizionale ha, infatti, la seguente sintassi:

```
espressione_1 ? espressione_2 : espressione_3;
```

`espressione_1` viene valutata sempre e può risultare `true` o `false`. Se è `true`, viene valutata `espressione_2`, in caso contrario viene valutata `espressione_3`.

Esempio

Determinare il massimo tra due variabili e stamparlo in output.

Soluzione

```
#include <iostream>
using std::cout;
using std::endl;

int main()
{
    int i = 3, j = 8;

    cout << "Il valore massimo tra "
         << i << " e " << j << " e' "
         << ( i > j ? i : j)
         << endl;

    return 0;
}
```

5.3.2 L'istruzione switch

L'istruzione `switch` consente di eseguire diversi blocchi di istruzioni in relazione al valore di un'espressione intera. La sintassi dell'istruzione `switch` è la seguente:

```
switch ( espressione_intera )
{
    case ( valore_costante_1 ):
        blocco_istruzioni_1
    break;
    case ( valore_costante_2 )
        blocco_istruzioni_2
    break;
    ...
    ...
    ...
    default:  // è opzionale
        blocco_istruzioni_default
}
```

Innanzitutto, viene valutata l'espressione intera `espressione_intera`. A seconda del valore assunto da `espressione_intera`, viene eseguito il blocco di istruzioni relativo al `case` il cui valore costante è uguale al valore assunto da `espressione_intera`. Qualora non esistesse un `case` corrispondente, se presente il caso `default`, sarà eseguito il `blocco_istruzioni_default`. La parola riservata `break` è opzionale e determina la fine del blocco di istruzioni che la precede; qualora venisse omesso, l'esecuzione del programma continua dal blocco di istruzioni successivo, indipendentemente dal valore della costante del case successivo.

Esempio

Scrivere un programma che presenti il seguente menu di scelta di quattro elementi

```
(1) Stampa lista
(2) Aggiungi elemento
(3) Elimina elemento
(4) Cancella lista
```

che accetti un input numerico da parte dell'utente corrispondete a una scelta di menu e che invochi una funzione che stampi l'azione corrispondente alla scelta effettuata. Gestire anche un eventuale errore d'inserimento.

Soluzione

```
/*

  Programma menu.cpp
  gestisce un menu con 4 opzioni
  tramite switch
```

```
   es. menu

   (1) Stampa lista
   (2) Aggiungi elemento
   (3) Elimina elemento
   (4) Cancella lista

*/


#include <iostream>
using std::cin;
using std::cout;


void stampa_menu()
{
  cout << "MENU DI SCELTA LISTA:" << "\n";
  cout << "(1) Stampa lista" << "\n";
  cout << "(2) Aggiungi elemento" << "\n";
  cout << "(3) Elimina elemento" << "\n";
  cout << "(4) Cancella lista" << "\n";
  cout << "\n";
}

void stampa_lista()
{
  cout << "Sto stampando la lista..." << "\n";
}

void aggiungi_elemento()
{
  cout << "Sto aggiungendo un elemento alla lista..." << "\n";
}

void elimina_elemento()
{
  cout << "Sto eliminando un elemento dalla lista..." << "\n";
}

void cancella_lista()
{
  cout << "Sto cancellando la lista..." << "\n";
}

void stampa_bestemmia()
{
  cout << "Maledetto, mi vuoi far perdere tempo?!" << "\n";
```

```
}


int main()
{
  int scelta_utente;

  stampa_menu();

  cout << "Scegli l'opzione dal menu sopra: ";
  cin >> scelta_utente;

  switch( scelta_utente )
  {
    case 1:
      stampa_lista();
    break;
    case 2:
      aggiungi_elemento();
    break;
    case 3:
      elimina_elemento();
    break;
    case 4:
      cancella_lista();
    break;
    default:
      stampa_bestemmia();
    break;
  }

  return 0;
}
```

5.4 Varianti dell'istruzione di iterazione

5.4.1 L'istruzione do-while

L'istruzione di iterazione principale, il `while`, ha la condizione di continuazione del ciclo in testa. Per tale motivo, se la condizione è inizialmente falsa, il blocco di istruzioni non viene mai eseguito. Il linguaggio C++ mette a disposizione l'istruzione `do-while` che ha la condizione in coda. Per tale motivo, il blocco di istruzione è eseguito almeno una volta. La sintassi dell'istruzione `do-while` è la seguente

La sintassi dell'istruzione è la seguente:

```
do
    blocco_istruzioni
while (condizione_di_continuazione);
```

Esempio

Leggere da input una sequenza di interi maggiori di zero, calcolare e stampare la somma in output. La sequenza è terminata dal *valore sentinella* zero.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
    int n;
    int somma = 0; // somma dev'essere necessariamente posta a zero

    do
    {
        cout << "Inserisci un intero: ";
        cin >> n;

        somma = somma + n;

    } while ( n > 0 );

    cout << "La somma è " << somma << "\n";

    return 0;
}
```

5.4.2 L'istruzione for

Il ciclo `for` è l'ultimo esempio di istruzione di iterazione messo a disposizione dal C++. Il ciclo `for` è molto utilizzato nei contesti di iterazione regolata da contatore, cioè quei casi in cui sono noti a priori il numero di iterazioni. La sintassi del ciclo `for` è la seguente:

```
for (inizializzazione; condizione_di_continuazione; espressione)
    blocco_istruzioni
```

L'`inizializzazione` è un'istruzione generalmente utilizzata per assegnare un valore iniziale a una variabile contatore che è poi modificato nel corso del ciclo. La `condizione_di_continuazione` è un'espressione logica il cui valore di verità `true` determina l'esecuzione del `blocco_istruzioni`. Non appena la condizione di continuazione diventa falsa, il ciclo s'interrompe e si passa all'esecuzione dell'istruzione successiva. L'`espressione` è invece solitamente utilizzata per modificare il valore di una o più variabili inizializzate in `inizializzazione` (ad esempio incrementare o decrementare un contatore).

Esempio

Scrivere un programma che calcoli la somma dei primi 100 numeri naturali (cioè quelli compresi tra 0 e 99, essendo 0 il primo numero naturale).

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

#define N 100

int main()
{
    int cont;  // contatore
    int somma = 0;

    for ( cont = 0; cont < N; cont = cont+1 )
        somma = somma + cont;

    cout << "La somma è " << somma << "\n";

    return 0;
}
```

Attraverso la direttiva del precompilatore `#define`, il programma definisce la costante `N` assegnandole il valore 100. Il programma dichiara, inoltre, le variabili intere `cont` e `somma`; la prima variabile sarà utilizzata come contatore per il ciclo for, la seconda come accumulatore dei primi 100 interi. Il ciclo `for`, per prima cosa, inizializza `cont` al valore 0 (che è il primo numero naturale). Viene quindi valutata la condizione di continuazione `cont < N`, che risulta vera essendo `cont` uguale a 0. Viene quindi eseguito il blocco di istruzioni del ciclo, in questo caso formato da una sola istruzione, e viene, infine incrementato il contatore di uno. Alla fine della prima iterazione, dunque, il contatore, che inizialmente valeva 0,

vale 1. A questo punto, si valuta nuovamente la condizione di continuazione e si procede esattamente come prima. L'ultima iterazione a essere eseguita è quella in cui `cont` vale `N - 1`. Infatti, una volta terminata l'iterazione $(N-1)$-esima, `cont` assume il valore `N` e la condizione di continuazione risulta falsa, il chè determina la fine del ciclo. In altri termini, il ciclo `for` dell'esempio può essere letto così:

per `cont` *che va da* `0` *a* `N-1`*, aggiungi* `cont` *a* `somma`

Esempio

Leggere da input una sequenza di `N` interi, calcolare e stampare la somma in output. Il valore dell'intero `N` dovrà essere letto da input e dovrà essere maggiore di zero. Nel caso in cui `N` risultasse minore o uguale a zero, dovrà essere chiesto l'inserimento di un nuovo valore e la richiesta dovrà essere reiterata in caso di ulteriori errori. Si noti che tale esempio era già stato proposto come esercizio nella sezione relativa al ciclo `while`.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
    int N;          // il numero di interi da leggere
    int a;          // intero corrente
    int somma = 0; // somma dev'essere necessariamente posta a zero

    do {
        cout << "Inserisci il numero di dati da sommare: ";
        cin >> N;
        if ( N <= 0)
            cout << "Devi inserire un numero maggiore di 0!\n";
    } while ( N <= 0 );

    for (int i = 0; i < N; i++) // si noti l'operatore ++
    {                           // i++; equivale a i = i + 1;
        cout << "Inserisci il dato numero " << i << ": ";
        cin >> a;
        somma += a;             // si noti l'operatore +=
                                // somma += a; equivale a
                                // somma = somma + a;
    }
```

```
    cout << "La somma è " << somma << "\n";

    return 0;
}
```

Operatori compatti

Si noti, nell'esempio precedente, l'uso delle degli operatori di incremento e assegnamento con somma, rispettivamente `++` e `+=`. Il loro significato è descritto nei commenti al codice. Esistono anche gli operatori `--` per il decremento e `-=`, forma compatta che esegue prima una sottrazione e poi un assegnamento.

5.4.3 Le istruzioni break e continue

L'istruzione `break`, già incontrata come parte dell'istruzione `switch`, termina l'istruzione `while`, `do-while`, `for` o, per l'appunto `switch`.

Esempio

Leggere da input una sequenza di `N` interi e stampare la stringa "Hai vinto!"" se l'utente inserisce il numero 7. In caso contrario, qualora il valore 7 non venga mai inserito, stampare la stringa "Non hai vinto!"". Il valore dell'intero `N` dovrà essere letto da input e dovrà essere maggiore di zero. Nel caso in cui `N` risultasse minore o uguale a zero, dovrà essere chiesto l'inserimento di un nuovo valore e la richiesta dovrà essere reiterata in caso di ulteriori errori.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
  int N;                          // numero di tentativi
  int a;                          // tentativo
  const int numero_fortunato = 7; // bisogna indovinarlo

  do{
    //chiedo all'utente di inserire N
    cout << "Inserisci il numero di tentativi: ";
```

```
    cin >> N;

    if (N <= 0)
      cout << "Numero tentativi deve essere maggiore di zero!\n";

  } while (N <= 0);

  for (int i = 0; i < N; i++)
  {
    cout << "Tentativo " << i << ": ";
    cin >> a;

    if ( a == numero_fortunato )
      break;
  }

  if (a == numero_fortunato)
    cout << "Hai vinto!\n";
  else
    cout << "NON hai vinto!\n";

  return 0;
}
```

Si noti l'uso del qualificatore `const` per la definizione di `numero_fortunato`. L'utilizzo di `const` preverrà qualsiasi tentativo di modifica del numero fortunato, il chè sarebbe un errore logico.

Esempio

Generare in modo random un numero intero (`numero_fortunato`) compreso tra 0 e 9. Leggere da input una sequenza di N interi e stampare la stringa "Hai vinto!"" se l'utente inserisce il `numero_fortunato`. In caso contrario, qualora `numero_fortunato` non venga mai inserito, stampare la stringa "Hai vinto!"". Il valore dell'intero N dovrà essere letto da input e dovrà essere maggiore di zero. Nel caso in cui N risultasse minore o uguale a zero, dovrà essere chiesto l'inserimento di un nuovo valore e la richiesta dovrà essere reiterata in caso di ulteriori errori.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

#include <cstdlib>     // srand(), rand()
#include <ctime>       // time()
```

```
int main()
{
  int N;                                  // numero di tentativi
  int a;                                  // tentativo

  srand( time(0) );

  const int numero_fortunato = rand() % 10;

  do{
    //chiedo all'utente di inserire N
    cout << "Inserisci il numero di tentativi: ";
    cin >> N;

    if (N <= 0)
      cout << "Il numero di tentativi deve essere maggiore di zero!\n
          ";

  } while (N <= 0);

  for (int i = 0; i < N; i++) // i++ equivale a i = i + 1;
  {
    cout << "Tentativo " << i << ": ";
    cin >> a;

    if ( a == numero_fortunato )
      break;
  }

  if (a == numero_fortunato)
    cout << "Hai vinto!\n";
  else
    cout << "NON hai vinto!\n";

  return 0;
}
```

Si noti l'inclusione degli header file `cstdlib` e `ctime`. Il primo contiene le definizioni delle funzioni `srand()` e `rand()`, mentre il secondo contiene la definizione della funzione `rand()`. Tali file header, poiché ereditati dal linguaggio C (dove prendevano rispettivamente il nome di `stdlib.h` e `time.h` - nomi utilizzabili in realtà anche in C++), non presentano namespace, per cui diviene superfluo l'uso delle direttive d'uso. Per quanto tiguarda la generazione dei numeri casuali, la funzione `srand()`, che prende come parametro un intero, deve essere invocata per prima e una sola volta. Essa è necessaria per inizializzare il generatore di numeri pseudo-casuali. I numeri pseudo-casuali generati sono funzione del parametro intero della funzione,

detto seed. Nell'esempio, il seed è ottenuto invocando la funzione `time(0)`, che restituisce un intero corrispondente al numero di secondi trascorsi dall'1 gennaio 1970 a oggi. Una volta inizializzato in generatore di numeri pseudo-casuali, è sufficiente invocare la funzione `rand()`, la quale ritorna un intero pseudo-casuale compreso tra 0 e `RAND_MAX`. Per ricondurre il valore ottenuto nell'intervallo dei numeri interi compresi tra 0 e 9, è sufficiente utilizzare l'operatore modulo `%`, che restituisce il resto della divisione intera.

L'istruzione `continue` interrompe invece l'esecuzione del blocco di istruzione in cui compare ma non determina la fine del ciclo, che continua con l'iterazione successiva.

Esempio

Leggere da input una sequenza di `N` interi e contare quante volte viene inserito un valore diverso dal numero 7. Il valore dell'intero `N` dovrà essere letto da input e dovrà essere maggiore di zero. Nel caso in cui `N` risultasse minore o uguale a zero, dovrà essere chiesto l'inserimento di un nuovo valore e la richiesta dovrà essere reiterata in caso di ulteriori errori.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

int main()
{
    int N;          // il numero di interi da leggere
    int a;          // intero corrente
    int cont = 0;   // contatore di non 7

    cout << "Programma: conta i non 7";

    do {
        cout << "Inserisci il numero di dati da inserire: ";
        cin >> N;
        if ( N <= 0)
            cout << "Devi inserire un numero maggiore di 0!\n";
    } while ( N > 0 );

    for (int i = 0; i < N; i++)
    {
        cout << "inserisci il dato numero " << i <<  ": ";
        cin >> a;
```

```
        if ( a == 7 )
            continue;  // se a == 7 si passa all'iterazione
                successiva

        cont++;         // l'incremento è eseguito solo se a != 7
    }

    cout << "Il numero di non 7 inseriti e' " << cont << "\n";

    return 0;
}
```

Capitolo 6

Funzioni, riferimenti, puntatori e allocazione dinamica della memoria

Come anticipato nella sezione 4.3, le funzioni C++ consistono di 4 parti, di cui le prime tre formano il *prototipo della funzione*:

1. un tipo di ritorno;
2. il nome della funzione;
3. una lista di parametri separati da virgole;
4. il corpo della funzione.

Negli esempi di funzioni C++ visti fino a ora, la lista di parametri è sempre stata vuota. Tuttavia, nel momento in cui l'elaborazione di un certo compito che richiede uno o più dati iniziali è demandata a una funzione, è necessario *passare* alla funzione uno o più *argomenti*. Questo avviene al momento della chiamata (o invocazione) della funzione e gli argomenti passati alla funzione sono utilizzati per inizializzare le variabili definite nella lista di parametri. Tali parametri sono quindi vere e proprie variabili la cui visibilità (o scope) è limitata alla funzione in cui sono definite[1].

In C++, il passaggio di parametri alle funzioni può avvenire secondo tre meccanismi principali:

[1] Qualsiasi tentativo di utilizzare un parametro al difuori della funzione in cui è stato definito genera un errore di sintassi; in caso di omonimia, tra argomento e parametro, la variabile effettivamente utilizzata è quella definita localmente nello stesso blocco di codice. In altri termini, l'argomento è utilizzato nel corpo della funzione chiamante (ad esempio `main`), il parametro nel corpo della funzione chiamata.

- per *copia*;
- per *puntatore*;
- per *riferimento*.

Come vedremo, tuttavia, il passaggio per puntatore è in realtà un passaggio per copia di una variabile speciale, detta puntatore. I meccanismi sopra citati sono descritti nelle sezioni seguenti.

6.1 Passaggio di parametri per copia (o per valore)

Come il nome suggerisce, il passaggio di parametri per copia determina la copia dei valori delle variabili che vengono passate come argomenti alla funzione. I parametri sono inizializzati al valore degli argomenti e sono visibili esclusivamente all'interno del corpo della funzione. In tal modo, la funzione opera su copie locali degli argomenti e qualsiasi eventuale modifica dei valori delle copie locali non si traduce in una modifica dei valori degli argomenti.

Esempio

Scrivere un programma che calcoli e stampi il massimo tra due interi letti in input, demandando la valutazione e la stampa a una funzione.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

void valuta_massimo_e_stampa(int n1, int n2)
{
    cout << "Il massimo tra " << n1 << " e " << n2 << " e': ";

    if (n1 > n2)
        cout << n1 << "\n";
    else
        cout << n2 << "\n";
}

int main()
{
```

```
  int a, b;

  cout << "Inserisci due interi di cui calcolare il massimo: ";
  cin >> a >> b;

  valuta_massimo_e_stampa(a, b);

  return 0;
}
```

Il programma legge due interi, `a` e `b`, di cui si vuole conoscere il massimo, e li passa, come argomenti, alla funzione `valuta_massimo_e_stampa`. Questa funzione, infatti, accetta due parametri interi, `n1` e `n2`. I parametri `n1` e `n2`, visibili esclusivamente all'interno della funzione `valuta_massimo_e_stampa`, sono inizializzati rispettivamente ai valori di `a` e `b`. In tal modo, in virtù di tale corrispondenza uno a uno, è possibile confrontare `n1` e `n2` in luogo di `a` e `b` al fine di valutare il massimo e stamparlo in output.

6.1.1 Parametri costanti

Si noti che il meccanismo della copia, così come considerato nell'esempio precedente, presenta un potenziale problema dovuto al fatto che la funzione possiede sia i diritti in lettura che in scrittura sui parametri: se il programmatore, per errore, dovesse modificare, ad esempio, il valore di `n1` prima di effettuare il confronto con `n2`, la funzione potrebbe comportarsi in modo non corretto. Ad esempio, potrebbe accadere quanto illustrato nel seguente codice sorgente:

```
void valuta_massimo_e_stampa(int n1, int n2)
{
    n1 = -1078;

    cout << "Il massimo tra " << n1 << " e " << n2 << " e': ";

    if (n1 > n2)
        cout << n1 << "\n";
    else
        cout << n2 << "\n";
}
```

dove la prima istruzione della funzione va a modificare *clamorosamente* e senza alcuna giustificazione apparente il valore del parametro `n1`, generando un errore logico. Per evitare di incorrere in tali errori logici, è possibile adoperare *parametri costanti*, così come illustrato nella seguente riedizione del precedente esercizio.

```
#include <iostream>
using std::cin;
using std::cout;

void valuta_massimo_e_stampa(const int n1, const int n2)
{
    cout << "Il massimo tra " << n1 << " e " << n2 << " e': ";

    if (n1 > n2)
        cout << n1 << "\n";
    else
        cout << n2 << "\n";
}

int main()
{
  int a, b;

  cout << "Inserisci due interi di cui calcolare il massimo: ";
  cin >> a >> b;

  valuta_massimo_e_stampa(a, b);

  return 0;
}
```

In tal modo, ogni tentativo di modificare i valori dei parametri genera un errore di sintassi, rendendo il programma più robusto rispetto a potenziali errori logici.

6.2 Restituzione di un valore attraverso il tipo di ritorno

Il tipo di ritorno consente alla funzione di restituire, attraverso l'uso della parola riservata `return`, un valore alla funzione chiamante (ad esempio la funzione `main`).

Esempio

Scrivere la funzione `get_a_random_number` che restituisca un numero casuale tra un valore minimo e un valore massimo. Impedire accidentali modifiche dei valori minimo e massimo utilizzando parametri costanti. Scrivere inoltre la funzione `main` dalla quale invocare `get_a_random_number` ed eventuali altre funzioni di utilità.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

#include <cstdlib>  // srand(), rand()
#include <ctime>    // time()

void init_random_numbers()
{
    srand( time(NULL) );         // equivale a srand( time(0) );
}

int get_a_random_number(const int minimum, const int maximum)
{
    //calcolo ampiezza dell'intervallo [minimum, maximum]
    int width = maximum - minimum + 1;

    // ritorna un numero casuale tra minumum e maximum
    return rand()%width + minimum;
}

int main()
{
    const int MIN = 1;
    const int MAX = 100;
    int coin;

    init_random_numbers();
    coin = get_a_random_number(MIN, MAX);

    cout << "Il numero estratto nell'intervallo [" << MIN << ", " <<
       MAX << "] e' " << coin << "\n";

    return 0;
}
```

L'espressione `get_a_random_number(MIN, MAX)` è sostituita con il valore restituito dalla funzione tramite l'istruzione `return`. Così, se la funzione ritorna il numero casuale 73, allora l'espressione `int coin = get_a_random_number(MIN, MAX)` equivale all'espressione `int coin = 73`. Questo meccanismo basato sul tipo di ritorno funziona fintanto che la funzione deve restituire un unico valore. In tutti i casi in cui occorre restituire più valori, è necessario ricorrere ai meccanismi di passaggio dei parametri per puntatore o per riferimento, illustrati nelle sezioni seguenti.

6.3 Puntatori

Un puntatore è una variabile che contiene un indirizzo di memoria e si dichiara specificando un tipo e aggiungendo il simbolo `*`. Ad esempio:

```
int* pi;
```

dichiara un puntatore a intero, senza attribuirgli alcun valore iniziale. Il seguente esempio, invece, dichiara un puntatore e lo inizializza al puntatore nullo `NULL`. Il puntatore nullo è un modo per attribuire un valore iniziale a un puntatore senza attribuirgli nella pratica un indirizzo di memoria. `NULL` è definito nell file header `cstdlib`, che deve quindi essere incluso nel programma. In alternativa, è sufficiente includere un header che a sua volta, in modo diretto o indiretto, includa la definizione di `NULL`: questo è il caso di `iostream`;

```
int* pi = NULL;
```

Si noti che un puntatore può essere inizializzato a `NULL` anche nel seguente modo:

```
int* pi = 0;
```

Questo solleva il programmatore dal dover includere un header con la definizione di `NULL`, ma si traduce spesso in un programma meno leggibile.

Il seguente esempio dichiara, invece, un puntatore assegnandogli l'indirizzo di una variabile intera attraverso l'operatore indirizzo `&`.

```
int* pi = NULL;
int a = 7;

pi = &a;
```

Una volta che il puntatore *punta* a una variabile, come nel caso dell'esempio precedente, è possibile utilizzare l'operatore `*` per *deferenziare* il puntatore, cioè per accedere al valore della variabile cui punta il puntatore. Ad esempio, si consideri il seguente codice:

```
1.    int* pi = NULL;
2.    int a = 7;

3.    pi = &a;

4.    cout << "L'indirizzo di  a e': " << &a << "\n";
5.    cout << "Il valore   di pi e': " << pi << "\n";

6.    cout << "Il valore di  a  e': " <<  a  << "\n";
7.    cout << "Il valore di *pi e': " << *pi << "\n";
```

Nel listato precedente, dove per comodità sono stati inseriti i numeri di riga, dopo aver assegnato al puntatore a intero `pi` l'indirizzo della variabile intera `a` (riga 3), alle righe 4 e 5 seguono rispettivamente le stampe dell'indirizzo di `a` e del valore di `pi` (che è un indirizzo). Ovviamente, tali stampe produrranno lo stesso risultato. Successivamente, rispettivamente alle righe 6 e 7, seguono le stampe del valore della variabile `a` e del valore della variabile cui punta `pi`. Un possibile output del precedente esempio potrebbe essere il seguente:

```
L'indirizzo di a  e': 0x7fffa80c9394
Il valore   di pi e': 0x7fffa80c9394
Il valore di  a  e': 7
Il valore di *pi e': 7
```

dove gli indirizzi sono espressi in esadecimale[2]. Come si può vedere, il puntatore `pi` contiene l'indirizzo della variabile `a`. Inoltre, è possibile accedere al valore della variabile `a` in modo *diretto* utilizzando la variabile `a` stessa e in modo *indiretto* deferenziando il puntatore `pi`.

6.3.1 Allocazione dinamica della memoria

La dichiarazione delle variabili comporta l'allocazione della memoria necessaria a immagazzinare i dati del tipo specificato all'atto della dichiarazione stessa. Ad esempio, la dichiarazione

```
int n;
```

richiede l'allocazione di un numero di celle di memoria necessarie a immagazzinare un oggetto di tipo `int`[3]. Una volta allocata, operazione che avviene all'atto del caricamento del programma, la memoria allocata rimane tale fino al termine dell'esecuzione del programma.

Il C++ fornisce, tuttavia, due funzioni, `new` e `delete`, che consentono rispettivamente di allocare la memoria all'occorrenza e di deallocarla quando non è più necessaria. Per far questo, si utilizza un puntatore al tipo di dato che dovrà essere immagazzinato, come illustrato nel seguente esempio:

```
double* d_ptr = new double(10.7);
cout << "Il valore allocato e' " << *d_ptr;
delete d_ptr;
```

[2]Per convenzione, i numeri esadecimali iniziano per `0x`.

[3]Il compilatore `g++` utilizza 32 bit per gli `int`, quindi 4 celle di memoria da un byte.

L'istruzione `double* d_ptr = new double(10.7);` alloca nella memoria del calcolatore una sequenza contigua di celle tali da poter contenere un dato di tipo `double` e gli assegna il valore iniziale 10. Inoltre, ritorna l'indirizzo della prima cella di memoria allocata, che viene immediatamente assegnato al puntatore `d_ptr` grazie all'operatore `=`. Si noti che tale istruzione può essere anche scritta come:

```
double* d_ptr = NULL;
d_ptr = new double(10.7);
```

Nella stampa successiva, si accede al valore attraverso la deferenziazione del puntatore, dopodiche si dealloca la memoria precedentemente allocata attraverso `delete`. La specificazione del tipo (`double` nell'esempio) è necessaria per quantificare il numero di celle necessarie a contenere il dato[4].

6.4 Passaggio di parametri per puntatore

Il passaggio di parametri per puntatore, come già anticipato all'inizio del capitolo, è in realtà un passaggio per copia. Infatti, se nella lista dei parametri di una funzione è presente un puntatore, al momento della chiamata è necessario specificare o un altro puntatore o un indirizzo di memoria, al fine di inizializzare il parametro stesso. Tuttavia, pur trattandosi di un passaggio per copia, attraverso l'accesso indiretto al valore delle variabili puntate, è possibile modificare i valori di queste ultime, mantenendo le variazioni anche dopo la fine dell'esecuzione della funzione.

Esempio

Scrivere una funzione che scambi i valori di due variabili di tipo double utilizzando il passaggio dei parametri per puntatore. Definire anche la funzione `main` all'interno della quale inserire la chiamata alla funzione.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

void scambia(double* x, double* y)
{
  double temp = *x;
  *x = *y;
  *y = temp;
```

[4] Il compilatore `g++` utilizza 64 bit per i `double`, quindi 8 celle di memoria da un byte.

```
}

int main()
{
  double a, b;

  cout << "Inserisci due double: ";
  cin >> a >> b;

  cout << "Prima dello scambio:\n";
  cout << "a = " << a << "\n";
  cout << "b = " << b << "\n";

  scambia(&a, &b);

  cout << "Dopo lo scambio:\n";
  cout << "a = " << a << "\n";
  cout << "b = " << b << "\n";

  return 0;
}
```

Innanzitutto, si osservi che gli argomenti della chiamata sono due indirizzi di memoria, rispettivamente delle variabili `a` e `b`. In tal modo, i parametri della funzione `scambia` sono inizializzati agli indirizzi di `a` e `b` (si dice che i parametri `x` e `y` puntano alle variabili `a` e `b`). Così, tramite `x` e `y`, attraverso il meccanismo della deferenzianzione, è possibile operare sui valori degli argomenti `a` e `b`. In tal modo, quando la funzione termina, i parametri `x` e `y` vengono *distrutti* ma i valori degli argomenti risultano effettivamente variati. Un possibile output del precendente programma potrebbe essere il seguente:

```
Inserisci due double: 12 -9
Prima dello scambio:
a = 12
b = -9
Dopo lo scambio:
a = -9
b = 12
```

Esempio

Scrivere un programma che legga una sequenza di interi positivi e ritorni il minimo e il massimo della sequenza. Lettura e ricerca del massimo e del minimo della sequenza devono essere demandati a una funzione, così come la stampa in output del massimo e del minimo.

Soluzione

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

void leggi_sequenza_e_trova_min_max(int* minimum, int* maximum)
{
  int n;

  *minimum = INT_MAX;
  *maximum = 0;

  cout << "Inserisci una sequenza di interi terminata da -1\n";
  do{
    cout << "Inserisci un elemento della sequenza: ";
    cin >> n;
    if ( n > 0){
      if (*minimum > n)
        *minimum = n;
      if (*maximum < n)
        *maximum = n;
    }
  } while (n > 0);
}


void stampa(int minimum, int maximum)
{
  cout << "Il mimimo della sequenza è " << minimum << "\n";
  cout << "Il massimo della sequenza è " << maximum << "\n";
}


int main()
{
  int minimo, massimo;

  leggi_sequenza_e_trova_min_max(&minimo, &massimo);

  stampa(minimo, massimo);

  return 0;
}
```

La lista di parametri della funzione `leggi_sequenza_e_trova_min_max` ha due puntatori a intero. La lista degli argomenti della chiamata alla funzione, infatti, è costituita da due indirizzi di memoria, rispettivamente quelli della variabile `minimo`

e `massimo`. Si noti che l'uso dei puntatori come parametri di funzione obbliga a usare l'operatore `*` qualora sia necessario accedere al valore della variabile puntata, come nell'esempio precedente. L'algoritmo inizializza `*minimum` al massimo valore possibile (cioè `INT_MAX`) e `*maximum` al minimo valore possibile (cioè zero). Un ciclo controllato da un valore sentinella richiede quindi a ogni iterazione l'inserimento di un intero, `n`, e lo confronta col minimo e col massimo. Se il numero risulta minore del minimo, allora `*minimum` viene impostato a `n`; allo stesso modo, se il numero risulta maggiore del massimo, allora il `*maximum` viene impostato a `n`.

6.4.1 Parametri puntatore a valore puntato costante, puntatore costante e puntatore costante a valore puntato costante

Come nel caso del passaggio per copia, è possibile prevenire la modifica del valore puntato da un parametro puntatore attraverso l'uso della parola riservata `const`. Si supponga, ad esempio, di voler impedire la modifica del valore puntato dal puntatore a intero `int* ptr`. Per ottenere questo risultato è sufficiente, all'atto del passaggio di `ptr` a una funzione, utilizzare il parametro `const int* int_ptr`, che dichiara un puntatore a valore puntato costante. In tal modo, ogni tentativo di modificare il valore `*int_ptr` restituisce un errore di sintassi.

Esempio

Scrivere una funzione che riceva in input un puntatore intero e verifichi se il valore puntato è un numero primo. Fare in modo che non sia possibile modificare il valore puntato attraverso l'uso di un parametro a valore puntato costante. Definire, inoltre, la funzione `main`, all'interno della quale inserire la chiamata alla funzione.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

bool primo(const int* n)
{
    // Ogni tentativo di modificare *n
    // produrrebbe un errore di sintassi

  bool primo = true;

  for (int i = 2; i < *n - 1; i++)
```

```
    if (*n % i == 0)
    {
      primo = false;
      break;
    }
  return primo;
}

int main()
{
  int n;
  int* ptr_int = &n;

  cout << "Inserisci il valore di n: ";
  cin >> n;

  if ( primo(ptr_int) )
    cout << n << " e' primo.\n";
  else
    cout << n << " non e' primo.\n";

  return 0;
}
```

Quando invece si vuole prevenire la modifica del puntatore stesso, di modo che esso non possa puntare ad alcun altro indirizzo di memoria, allora è necessario utilizzare parametri puntatore costante. Questo potrebbe essere il caso della funzione `scambia` vista precedentemente, nella quale è necessario variare i valori puntati (per effettuare lo scambio) ma non è ammissibile alcuna variazione dei puntatori stessi (cioè, i puntatori non possono puntare ad altre locazioni di memoria). Per definire un puntatore costante è sufficiente utilizzare la parola riservata `const` tra la specificazione del tipo puntatore e il nome della variabile, ad esempio: `int* const ptr_int`.

Esempio

Scrivere una funzione che scambi i valori di due variabili di tipo double utilizzando il passaggio dei parametri per puntatore. Prevenire eventuali tentativi di modifica dei puntatori utilizzando parametri puntatore costante. Definire, inoltre, la funzione `main`, all'interno della quale inserire la chiamata alla funzione.

Soluzione

```
#include <iostream>
using std::cin;
```

```
using std::cout;

void scambia(double* const x, double* const y)
{
    // Ogni tentativo di modificare i puntatori
    // x e y facendoli puntare a indirizzi differenti
    // produrrebbe un errore di sintassi

  double temp = *x;
  *x = *y;
  *y = temp;
}

int main()
{
  double a, b;

  cout << "Inserisci due double: ";
  cin >> a >> b;

  cout << "Prima dello scambio:\n";
  cout << "a = " << a << "\n";
  cout << "b = " << b << "\n";

  scambia(&a, &b);

  cout << "Prima dello scambio:\n";
  cout << "a = " << a << "\n";
  cout << "b = " << b << "\n";

  return 0;
}
```

Infine, nei casi in cui è necessario passare a una funzione un valore attraverso un puntatore volendo impedire tanto variazioni sul valore stesso che sul puntatore, si può far uso di un parametro puntatore costante a valore puntato costante. Un tale puntatore si dichiara utilizzando due volte il qualificatore `const`: la prima volta prima del tipo, la seconda subito prima del nome della variabile. Una possibile dichiarazione di un puntatore costante a intero a valore puntato costante è la seguente: `const int* const ptr_int`.

Esempio

Scrivere una funzione che riceva in input un puntatore intero e verifichi se il valore puntato è un numero primo. Fare in modo che non sia possibile né modificare il valore puntato, né il puntatore stesso, attraverso l'uso di un parametro puntatore

costante a valore puntato costante. Definire, inoltre, la funzione `main`, all'interno della quale inserire la chiamata alla funzione.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

bool primo(const int* const n)
{
    // ogni tentativo di modificare *n
    // produrrebbe un errore di sintassi
    //
    // inoltre, ogni tentativo di modificare il puntatore
    // n facendolo puntare an indirizzo differente
    // produrrebbe un errore di sintassi

  bool primo = true;

  for (int i = 2; i < *n - 1; i++)
    if (*n % i == 0)
    {
      primo = false;
      break;
    }
  return primo;
}

int main()
{
  int n;
  int* ptr_int = &n;

  cout << "Inserisci il valore di n: ";
  cin >> n;

  if ( primo(ptr_int) )
    cout << n << " e' primo.\n";
  else
    cout << n << " non e' primo.\n";

  return 0;
}
```

6.5 Riferimenti

Gli inconvenienti del passaggio di parametri per puntatore sono principalmente due: il primo consiste nel dover utilizzare l'operatore `*` ogni volta che si ha necessità di accedere in lettura o scrittura al valore delle variabili puntate; il secondo consiste nel dover specificare l'indirizzo degli argomenti attraverso l'operatore `&` al momento della chiamata[5]. Per ovviare a tali inconvenienti, a differenza del linguaggio C, il C++ definisce il tipo riferimento.

Un riferimento può essere considerato come una sorta di *alias*, cioè un nome alternativo per una variabile. Un riferimento si dichiara aggiungendo una `&` al tipo della variabile; l'inizializzazione a una variabile dello stesso tipo è, inoltre, obbligatoria. Un riferimento, ad esempio a una variabile intera, si dichiara quindi nel seguente modo:

```
int n = 3;
int& rif_n = n;
```

Per capire meglio di cosa si tratti, si provi a stampare gli indirizzi di `n` e di `rif_n` col seguente codice:

```
cout << "&n     = " << &n << "\n";
cout << "&rif_n = " << &rif_n << "\n";
```

Un possibile output potrebbe essere il seguente:

```
&n     = 0x7fffb54a0ec4
&rif_n = 0x7fffb54a0ec4
```

In pratica riferimento e variabile riferita hanno lo stesso indirizzo per cui diventa equivalente utilizzare l'uno o l'altra per modificare il valore della locazione di memoria a quell'indirizzo. Pertanto, quando si agisce sul riferimento `rif_n` è come se si agisse sulla variabile `n` e ogni variazione al valore dell'alias comporta una variazione del valore della variabile. Ad esempio, si consideri la seguente sequenza di istruzioni:

```
rif_n = 12;
cout << n;
```

In questo caso, il valore della variabile intera `n` viene modificato, in modo indiretto, attraverso il riferimento `rif_n`. Così, la stampa in output del valore di `n` produrrà come risultato 12.

Si noti, infine, che nel caso dei riferimenti non è necessaria alcuna operazione di deferenziazione, come avveniva invece nel caso dei puntatori.

[5]In realtà esiste almeno un altro inconveniente legato alla ridefinizione degli operatori nelle classi C++ che, tuttavia, non sono argomento di questo testo.

6.6 Passaggio di parametri per riferimento

I riferimenti possono essere utilizzati come parametri di funzione e offrono un modo alternativo all'uso dei puntatori nei casi in cui si voglia modificare il valore degli argomenti passati alla funzione al momento della chiamata. In questo caso, il parametro riferimento riceve l'indirizzo dell'argomento e diventa, di fatti, un nome alternativo dell'argomento. Quando si opera sul parametro, si opera sull'argomento e le eventuali modifiche effettuate persistono anche dopo il termine della funzione. A differenza del passaggio per puntatore, non è necessario deferenziare i riferimenti per accedere al valore immagazzinato in memoria. Inoltre, al momento della chiamata non è necessario fornire gli indirizzi delle variabili argomento, ma è sufficiente fornire le variabili stesse, come nel caso del passaggio per copia.

Esempio

Scrivere una funzione che scambi i valori di due variabili di tipo double utilizzando il passaggio dei parametri per riferimento. Definire anche la funzione `main` all'interno della quale inserire la chiamata alla funzione.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;

void scambia(double& x, double& y)
{
  double temp = x;
  x = y;
  y = temp;
}

int main()
{
  double a, b;

  cout << "Inserisci due double: ";
  cin >> a >> b;

  cout << "Prima dello scambio:\n";
  cout << "a = " << a << "\n";
  cout << "b = " << b << "\n";

  scambia(a, b);
```

```
  cout << "Dopo lo scambio:\n";
  cout << "a = " << a << "\n";
  cout << "b = " << b << "\n";

  return 0;
}
```

Esempio

Scrivere una funzione che legga una sequenza di interi positivi e ritorni il minimo e il massimo della sequenza utilizzando parametri riferimento.

Soluzione

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

void leggi_sequenza_e_trova_min_max(int& minimum, int& maximum)
{
  int n;

  minimum = INT_MAX;
  maximum = 0;

  cout << "Inserisci una sequenza di interi terminata da -1\n";
  do{
    cout << "Inserisci un elemento della sequenza: ";
    cin >> n;
    if ( n > 0){
      if (minimum > n)
        minimum = n;
      if (maximum < n)
        maximum = n;
    }
  } while (n > 0);
}


void stampa(int minimum, int maximum)
{
  cout << "Il mimimo della sequenza è " << minimum << "\n";
  cout << "Il massimo della sequenza è " << maximum << "\n";
}
```

```
int main()
{
  int minimo, massimo;
  leggi_sequenza_e_trova_min_max(minimo, massimo);
  stampa(minimo, massimo);
  return 0;
}
```

Nella funzione `main` sono dichiarate le due variabili intere `minimo` e `massimo`, che dovranno assumere il valore minimo e massimo della sequenza di interi che dovrà essere letta in input. Tali variabili vengono passate prima alla funzione `leggi_sequenza_e_trova_min_max` e, successivamente, alla funzione `stampa`. Nel caso della prima funzione, i parametri `int& minimum` e `int& maximum` sono riferimenti agli argomenti listati nella chiamata alla funzione, rispettivamente `minimo` e `massimo`. In tal modo, ogni modifica ai riferimenti `minimum` e `maximum` si traduce in un'immediata modifica delle variabili `minimo` e `massimo`, che manterranno i valori modificati anche dopo il termine della funzione.

6.6.1 Parametri riferimento costanti

Come nel caso del passaggio di parametri per copia (e puntatore), anche nel caso del passaggio per riferimento è possibile utilizzare il qualificatore `const` per prevenire la modifica dei valori passati come argomento.

Esempio

Riscrivere la funzione `stampa` del precedente esempio utilizzando parametri riferimento costanti.

Soluzione

```
void stampa(const int& minimum, const int& maximum)
{
  cout << "Il mimimo della sequenza è" << minimum << "\n";
  cout << "Il massimo della sequenza è" << maximum << "\n";
}
```

6.7 Parametri riferimento a puntatore e valore di default

Si supponga di voler scrivere una funzione che allochi la memoria per una variabile di tipo `float` e ritorni il puntatore all'area di memoria allocata. E' possibile

risolvere il problema in due modi differenti: il primo consiste nello scrivere una funzione con tipo di ritorno `float*`, il secondo nell'utilizzo di un parametro di tipo `float*&` e cioè (leggendo da destra verso sinistra) un riferimento a un puntatore di tipo float.

Esempio

Scrivere una funzione che allochi la memoria per una variabile di tipo `float` e ritorni il puntatore all'area di memoria allocata attraverso il tipo di ritorno della funzione.

Soluzione

```
#include <iostream>
using std::cout;

float* alloca_float(float init_val = 0.0)
{
  return new float (init_val);
}

void dealloca_float(float* ptr_float)
{
  delete ptr_float;
}

int main()
{
  float* ptr_float = NULL;

  cout << "Dopo l'istruzione float* ptr_float = NULL; \n";
  cout << "  ptr_float = " <<  ptr_float << "\n\n";

  ptr_float = alloca_float();

  cout << "Dopo l'istruzione ptr_float = alloca_float(1024); \n";
    cout << " ptr_float = " <<  ptr_float << "\n";
  cout << "*ptr_float = " << *ptr_float << "\n";

  dealloca_float(ptr_float);

  return 0;
}
```

La funzione `alloca_float` ritorna un puntatore a `float`. Tale puntatore è il puntatore restituito dalla funzione `new` e punta alla prima cella di memoria allocata per

contenere un `float`. La funzione ha, inoltre, un parametro con *valore di default*. Questo vuol dire che la chiamata alla funzione può anche essere effettuata senza argomento, così come avviene nell'esempio. In questo caso, il parametro `init_val` assume il valore di default specificato, che vale 0.0. In caso contrario, il valore di default è ignorato e il parametro assume il valore dell'argomento. Se si volesse inizializzare la memoria che si va ad allocare al valore 1024.0, si dovrebbe utilizzare la chiamata `ptr_float = alloca_float(1024.0);`. Un possibile output del precedente programma potrebbe essere il seguente:

```
Dopo l'istruzione float* ptr_float = NULL;
 ptr_float = 0

Dopo l'istruzione ptr_float = alloca_float(1024);
  ptr_float = 0x1a6a010
 *ptr_float = 0
```

Si noti, infine, che è possibile (ma non consigliato) allocare la memoria senza specificare alcun valore iniziale. In tal caso, la funzione `alloca_float` diventa la seguente:

```
float* alloca_float()
{
  return new float;
}
```

Esempio

Scrivere una funzione che allochi la memoria per una variabile di tipo `float` e ritorni il puntatore all'area di memoria allocata tramite un parametro.

Soluzione

```
#include <iostream>
using std::cout;

void alloca_float(float*& ptr_float, float init_val = 0.0)
{
  ptr_float = new float (init_val);
}

void dealloca_float(float* ptr_float)
{
  delete ptr_float;
}
```

```
int main()
{
  float* ptr_float = NULL;

    cout << "Dopo l'istruzione float* ptr_float = NULL; \n";
  cout << "  ptr_float = " <<  ptr_float << "\n\n";

  alloca_float(ptr_float, 1024);

  cout << "Dopo l'istruzione ptr_float = alloca_float(1024); \n";
    cout << " ptr_float = " <<  ptr_float << "\n";
  cout << "*ptr_float = " << *ptr_float << "\n";

  dealloca_float(ptr_float);

  return 0;
}
```

Anche in questo caso, essendo il secondo parametro inizializzato a un valore di *default*, è possibile sostituire la chiamata con due argomenti con una chiamata a singolo argomento: `alloca_float(ptr_float);`. In tal caso, il valore iniziale per il `float` puntato da `ptr_float` sarà zero.

6.8 Overloading e Template di funzione

In C++ (ma non in C) è possibile, tramite il meccanismo dell'Overloading di funzioni, assegnare lo stesso nome a due o più funzioni, a patto che esse differiscano per numero e/o tipo di parametri.

Esempio

Scrivere un programma che calcoli il minimo tra due valori interi oppure tra due caratteri, demandano la valutazione a due funzioni aventi lo stesso nome.

Soluzione

```
#include <iostream>
using std::cout;

int minimo(int a, int b)
{
  if (a < b)
        return a;
    else
```

```
        return b;
}

char minimo(char a, char b)
{
  if (a < b)
        return a;
    else
        return b;
}

int main()
{
    cout << "Il minimo tra A ed F è" << minimo('A', 'F') << "\n";
    cout << "Il minimo tra -7 e 12 è" << minimo(-7, 12) << "\n";

  return 0;
}
```

Esiste, tuttavia, una soluzione più elegante e compatta al precedente esercizio. Grazie al meccanismo del Template di funzione, è possibile definire funzioni in grado di accettare parametri di tipi differenti. Una funzione template (o modello), come il nome stesso suggerisce, è una sorta di entità astratta che deve essere istanziata, cioè resa concreta, e trasformata in una funzione reale prima che possa essere effettivamente utilizzabile. L'instanziazione della funzione template è a carico del compilatore C++ e avviene in base alle richieste, cioè ai tipi degli argomenti presenti nelle chiamate.

Una funzione template si scrive definendo un tipo di dato astratto tramite le parole riservate `template` e `class` e poi scrivendo la funzione e utilizzando il tipo di dato astratto appena definito per almeno un parametro. Ad esempio, il tipo astratto `type` si dichiara scrivendo:

```
template <class type>
```

Esempio

Scrivere un programma scambi il valore di due valori numerici (short int, int, float, double, ecc.) utilizzando una funzione template.

Soluzione

```
#include <iostream>
using std::cin;
using std::cout;
```

```
template <class type>
void scambia(type& x, type& y)
{
  type temp = x;
  x = y;
  y = temp;
}

int main()
{
  int a, b;

  cout << "Inserisci due int: ";
  cin >> a >> b;

  cout << "Prima dello scambio:\n";
  cout << "a = " << a << "\n";
  cout << "b = " << b << "\n";

  scambia(a, b);

  cout << "Dopo lo scambio:\n";
  cout << "a = " << a << "\n";
  cout << "b = " << b << "\n";



    double c, d;

  cout << "Inserisci due double: ";
  cin >> c >> d;

  cout << "Prima dello scambio:\n";
  cout << "c = " << c << "\n";
  cout << "d = " << d << "\n";

  scambia(c, d);

  cout << "Dopo lo scambio:\n";
  cout << "c = " << c << "\n";
  cout << "d = " << d << "\n";

  return 0;
}
```

Un possibile output del precedente programma potrebbe essere il seguente:

```
Inserisci due int: 7 13
Prima dello scambio:
```

```
a = 7
b = 13
Dopo lo scambio:
a = 13
b = 7

Inserisci due double: -77.03 58.00176
Prima dello scambio:
c = -77.03
d = 58.0018
Dopo lo scambio:
c = 58.0018
d = -77.03
```

Capitolo 7

Gli array

L'array è un tipo particolare di dato che consente di raggruppare elementi dello stesso tipo. Gli array unidimensionali sono utili per modellare sequenze di dati tipo i vettori, mentre gli array bidimensionali sono utili per modellare dati in forma tabellare, tipo le matrici. È possibile definire anche array di dimensione superiore. Tuttavia, questo testo non copre la loro trattazione. Lo studente che abbia compreso appieno la logica degli array uni- e bidimensionali non avrà comunque problemi a dedurre definizione e utilizzo degli array di dimensione superiore.

7.1 Array unidimensionali

Gli array unidimensionali consentono di raggruppare dati omogenei, cioè dello stesso tipo, in un'area contigua di memoria. La presente sezione tratta l'allocazione statica e dinamica degli array unidimensionali e illustra alcuni algoritmi fondamentali di ricerca e ordinamento. In particolare, sono illustrati l'algoritmo della ricerca lineare, l'algoritmo di ordinamento *bubble sort* e l'algoritmo della ricerca binaria.

7.1.1 Allocazione statica degli array

Nell'allocazione statica, lo spazio necessario alla memorizzazione dell'array viene richiesto sin dalla fase di caricamento del programma in memoria e rimane occupato fino al termine del programma. Un esempio di dichiarazione statica di un array è il seguente:

```
const int dim_array = 8;
int v[dim_array];
```

In tal modo si dichiara un array di nome `v` di `dim_array` elementi interi. La dimensione dell'array dev'essere una costante[1].

Se si vuole inizializzare gli elementi dell'array, è possibile farlo adoperando la seguente sintassi:

```
const int dim_array = 8;
int v[dim_array] = {-7, 12, 19, 0, 6, 78, 3, 36};
```

Si noti che, in quest'ultimo caso, non è obbligatorio specificare la dimensione dell'array tra le parentesi quadre, poiché essa è dedotta dal compilatore dalla dimensione della lista di valori tra parentesi graffe. La seguente dichiarazione è, pertanto, equivalente alla precedente:

```
int v[] = {-7, 12, 19, 0, 6, 78, 3, 36};
```

A ogni elemento della sequenza è associato un indice: 0 al primo elemento, 1 al secondo e così via; l'ultimo elemento dell'array ha indice `dim_array - 1`. In tal modo è possibile accedere (sia in lettura che in scrittura) agli elementi della sequenza, attraverso l'operatore indice (o di *subscript*), `[]`, semplicemente specificando l'indice dell'elemento tra parentesi quadre. Ad esempio, in riferimento alla precedente definizione dell'array `v`, il seguente codice sorgente:

```
for (int i = 0; i < dim_array; i++)
    cout << "v[" << i << "] = " << v[i] << "\n";
```

produce il seguente output:

```
v[0] = -7
v[1] = 12
v[2] = 19
v[3] = 0
v[4] = 6
v[5] = 78
v[6] = 3
v[7] = 36
```

Si noti che il C++ non offre alcun meccanismo di protezione per l'accesso a elementi al difuori della dimensione dell'array. Il seguente codice, ad esempio, non genera alcun errore di sintassi, nonostante si stia accedendo in scrittura a un elemento che non appartiene all'array:

```
v[dim_array] = 2048;
```

[1] Il compilatore g++ accetta valori non costanti per la dimensione dell'array, ma questo non è aderente allo standard, quindi va evitato.

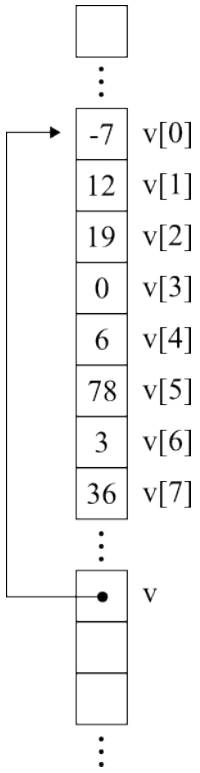

Figura 7.1: Rappresentazione astratta di un array in memoria RAM. Il nome dell'array, v, è un puntatore al primo elemento della sequenza contigua di interi che definisce l'array stesso. È possibile accedere, sia in lettura che in scrittura, agli elementi dell'array specificando tra parentesi quadre il nome dell'array seguito dalla posizione (o indice) dell'elemento all'interno della sequenza. Il primo elemento della sequenza ha indice 0, il secondo ha indice 1 e così via. Se *l* rappresenta la lunghezza dell'array, allora l'ultimo elemento ha indice *l-1*.

L'ultimo elemento dell'array è, infatti, quello di indice `dim_array-1`, partendo la sequenza di indici da 0. Il programmatore deve prestare quindi la massima attenzione a non incorrere in questi errori logici.

La figura 7.1 illustra uno schema astratto di rappresentazione dell'array `v` nella memoria del calcolatore. Come è possibile notare, il nome dell'array, `v`, non è altri che un puntatore al primo elemento dell'array. In altri termini, `v` contiene l'indirizzo di memoria del primo elemento della sequenza. Per accedere quindi al valore del primo elemento dell'array, ad esempio al fine di stamparlo in output, sarebbe sufficiente deferenziare `v` nel seguente modo:

```
cout << "Il valore del primo elemento dell'array v e': "
     << *v << "\n";
```

Tuttavia, alla deferenzianzione con l'operatore `*` è sicuramente da preferire quella che fa uso l'uso dell'operatore indice:

```
cout << "Il valore del primo elemento dell'array v e': "
     << v[0] << "\n";
```

Peraltro, attraverso l'operatore indice, risulta più agevole accedere agli altri elementi dell'array, cosa che con la deferenziazione basata sull'operatore `*` richiederebbe l'uso dell'*aritmetica dei puntatori*. L'aritmetica dei puntatori non è argomento di questo corso.

7.1.2 Allocazione dinamica degli array

Poiché, in linea di principio, in relazione alla propria dimensione, un array può occupare considerevoli quantità di memoria, è spesso preferibile optare per un metodo di allocazione dinamico. L'allocazione dinamica, a differenza della statica vista precedentemente, consente di richiedere la memoria necessaria all'array a *run time*, cioè durante l'esecuzione del programma, e di rilasciarla, sempre a run time, nel momento in cui l'elaborazione sull'array termina. In tal modo si può ottenere uno sfruttamento ottimale della memoria.

Un esempio di dichiarazione dinamica di un array è il seguente:

```
int dim_array = 8;
double* w = NULL;
w = new (nothrow) double[dim_array];
```

Si noti, innanzitutto, che, a differenza dell'allocazione statica, la dimensione dell'array è specificato tramite una variabile[2]. Il nome dell'array, `w`, è un oggetto di tipo `double*`, a riprova del fatto che il nome dell'array sia in realtà il puntatore al primo elemento della sequenza. Così come nell'esempio, è sempre buona norma inizializzare a `NULL` i puntatori. L'allocazione vera e propria dell'array avviene attraverso l'operatore `new`. Nell'esempio, l'operatore richiede un totale di `dim_array` elementi contigui e ritorna l'indirizzo del primo elemento della sequenza. Tale indirizzo è quindi assegnato a `w`, che diventa così, di fatto, il puntatore al primo elemento. L'opzione facoltativa `(nothrow)` fa si che, qualora il tentativo di allocazione della memoria fallisca, `new` ritorni il puntatore `NULL`[3]. Al fine di utilizzare

[2] È ancora possibile specificare la dimensione di un array dinamico attraverso una costante, ma la possibilità di adoperare una variabile conferisce al metodo una maggiore libertà d'azione. In tal modo è possibile, ad esempio, richiedere a run time la dimensione dell'array, immagazzinare tale valore in una variabile e utilizzare il valore acquisito per l'allocazione dell'array.

[3] Qualora `(nothrow)` venisse omesso, in caso di fallimento, l'espressione `new int[dim]` ritornerebbe un oggetto eccezione, la cui gestione esula dagli obiettivi di questo testo.

l'opzione `(nothrow)` è necessario includere nel programma l'header file `new`. Si noti, infine, che la precedente allocazione dell'array `w` può essere data in forma più compatta come segue:

```
int dim_array = 8;
double* w = new (nothrow) double[dim_array];
```

Una volta allocato, un array dinamico può essere utilizzato esattamente come un array statico. Quando un array dinamico non è più necessario, tuttavia, è possibile deallocarlo, liberando la memoria occupata, attraverso l'operatore `delete[]`, nel seguente modo:

```
delete[] w;
```

Si noti che non è necessario specificare la dimensione dell'array. Se il programmatore dimentica di deallocare l'array, la memoria torna comunque a disposizione del sistema operativo al termine dell'esecuzione del programma. Questo, tuttavia, non mette al riparo il programmatore da possibili malfunzionamenti. Non sono infrequenti, ad esempio, casi in cui la mancata deallocazione della memoria degli array determina un aumento eccessivo della memoria richiesta dal programma, con conseguente blocco dello stesso.

7.1.3 L'algoritmo della ricerca lineare

Il problema della ricerca di un elemento in un array consiste nel verificare se un dato elemento sia contenuto o meno all'interno dell'array. L'algoritmo più semplice (ma anche poco efficiente) è il così detto algoritmo della ricerca lineare in cui l'elemento da cercare viene confrontato con gli elementi dell'array sequenzialmente a partire dal primo. Il seguente esempio implementa l'algoritmo della ricerca lineare in un array.

Esempio

Scrivere un programma che legga un array di interi di dimensione arbitraria (letta anch'essa in input), legga un intero da cercare all'interno dell'array e implementi, a tal fine, l'algoritmo della ricerca lineare.

Soluzione

```
#include <cstdlib>     /* exit, EXIT_FAILURE, EXIT_SUCCESS */
#include <iostream>
using std::cin;
using std::cout;
```

```
using std::endl;
#include <new>
using std::nothrow;


int* alloca_array(const int dim = 1)
{
  return new (nothrow) int[dim];
}

void dealloca_array(int*& v)
{
  delete[] v;
  v = NULL;
}

void leggi_array(int* const v, const int dim)
{
  cout << "inserisci gli elementi dell'array...\n";
  for (int i=0; i<dim; i++){
    cout << "v[" << i << "]=";
    cin >> v[i];
  }
}

bool ricerca_lineare(const int* const w, const int dim, const int n)
{
  for (int i=0; i<dim; i++)
    if (n == w[i])
      return true;

  return false;
}


int main()
{
  // v è il nome dell'array nonché il puntatore al primo elemento
  int* v = NULL;
  int dim_array = 0;

  // a è l'elemento da cercare
  int a;

  cout << "Inserisci la dimensione dell'array: ";
  cin >> dim_array;

  //allocazione dinamica dell'array
  v = alloca_array(dim_array);
```

```
  if (v == NULL)
    exit (EXIT_FAILURE);

  leggi_array(v, dim_array);

  cout << "Inserisci un elemento da cercare: ";
  cin >> a;

  bool trovato = ricerca_lineare(v, dim_array, a);

  if (trovato)
    cout << "L'elemento " << a << " e' contenuto nell'array\n";
  else
    cout << "L'elemento " << a << " non e' contenuto nell'array\n";

  //deallocazione dell'array
  dealloca_array(v);

  return EXIT_SUCCESS;
}
```

Il programma definisce, innanzitutto, il puntatore a intero `v`, che diventerà, all'atto dell'allocazione della memoria, il puntatore al primo elemento dell'array. La dimensione dell'array è `dim_array`. L'intero `a` rappresenta, invece, l'elemento da cercare.

L'allocazione dell'array avviene invocando la funzione `alloca_array` che ha `const int dim = 1` come unico parametro, peraltro con valore di default 1. Questo implica che la funzione può essere invocata anche senza argomento, caso in cui sarà allocato un array di dimensione 1. La funzione ritorna un puntatore a intero, il cui valore è l'indirizzo del primo elemento dell'array. L'allocazione della memoria avviene tramite l'espressione `new (nothrow) int[dim]` che, in caso di successo, torna il puntatore al primo elemento della sequenza di `dim` interi allocata. Come già anticipato, l'opzione `(nothrow)` fa si che, in caso di fallimento, `new` ritorni il puntatore `NULL`. In quest'ultimo caso, la mancata allocazione può essere rilevata attraverso un banale controllo sul valore del puntatore. Nel `main`, infatti, nel caso `v == NULL`, il programma invoca la funzione `exit` con valore `EXIT_FAILURE`, che determina l'uscita dal programma segnalando al sistema operativo un fallimento dello stesso. Funzione e parametro, così come il parametro `EXIT_SUCCESS` utilizzato nell'istruzione `return` alla fine del `main`, sono definiti nell'header file `cstdlib`.

Nel caso in cui l'allocazione della memoria avvenga con successo, viene invocata la funzione `leggi_array`, il cui nome è autoesplicativo. Si noti il primo parametro della funzione: si tratta di `int* const v`. Trattandosi di un puntatore costante, la funzione non ha i permessi per modificare il valore di `v` facendolo puntare a un

indirizzo diverso da quello del primo elemento dell'array. In modo simile, il secondo parametro è `const int dim`. In tal modo si previene anche l'eventuale modifica di `dim`, che rappresenterebbe un errore logico.

Viene quindi letto `a` e viene invocata la funzione `ricerca_lineare`. La funzione esegue banalmente un ciclo sugli elementi dell'array, confrontando ogni elemento con il valore da cercare. Nel caso di successo, la funzione ritorna immediatamente il valore booleano `true`; qualora non sia mai verificata l'uguaglianza tra gli elementi dell'array e l'elemento cercato, il ciclo `for` termina e la funzione ritorna `false`. Si noti che l'array `v` viene passato come primo argomento alla funzione `ricerca_lineare` e che il primo parametro della funzione è `const int* const w`, cioè un puntatore costante a valore puntato costante. Tale scelta è giustificata dal fatto che la funzione non debba avere nè i diritti per modificare il valore del puntatore, nè i diritti per modificare i valori puntati, cioè gli elementi dell'array. È sempre buona norma, al fine di ridurre il rischio di commettere errori logici, utilizzare parametri che concedano esclusivamente i diritti necessari e non altri.

Il programma termina con la stampa del risultato della ricerca e con la deallocazione dell'array, a carico della funzione `dealloca_array`. Quest'ultima ha come parametro un `int*&`, cioè un riferimento a un puntatore a intero. In tal modo, all'atto della chiamata, quando viene specificato come argomento il nome dell'array, la funzione può operare sul parametro come se operasse sull'array stesso e, per tale motivo, può impostare, dopo la deallocazione con `delete[]`, il valore del puntatore a `NULL`. Se, al contrario, la funzione avesse previsto come parametro un `int*`, l'assegnazione a `NULL` non sarebbe stata *trasmessa* all'argomento, che avrebbe continuato a puntare dove puntava in precedenza, la qual cosa può rappresentare fonte di potenziali errori logici. Infatti, se per errore, si continuasse a utilizzare ancora l'argomento, si correrebbe il rischio di accedere ad aree di memoria magari non più appannagio del proprio programma, con effetti del tutto imprevedibili. L'espressione `return EXIT_SUCCESS`, equivalente alla nota `return 0`, determina la fine del programma.

Esercizio

Scrivere un programma che legga in input un array di interi e un valore da cercare all'interno dell'array e determini il numero di volte che tale elemento compare nell'array.

Esercizio

Scrivere un programma che riceva in input un array di numeri razionali in doppia precisione, un ulteriore dato in doppia precisione e un valore di tolleranza `epsilon`

e verifichi se l'array contiene il dato con tolleranza ±`epsilon`[4].

Esercizio

Scrivere un programma che legga in input due array di interi e un determini se il più piccolo è contenuto nel più grande.

Esercizio

Scrivere un programma che legga in input due array di `double` e determini se il più piccolo è contenuto nel più grande a meno di un errore ±`epsilon` su ogni elemento.

7.1.4 L'algoritmo di ordinamento *bubble sort*

L'algoritmo *bubble sort* è uno dei più semplici algoritmi di ordinamento su array. Si supponga di voler ordinare un array in senso crescente, dall'elemento più piccolo all'elemento più grande. L'algoritmo consiste di più *passate*, in ognuna delle quali si confrontano gli elementi adiacenti ed eventualmente si scambiano se il primo è più grande del secondo. In tal modo, gli elementi più piccoli (più leggeri) tendono ad andare verso l'alto come le bolle, da cui il nome dell'algoritmo. Ovviamente un'unica passata non è, in generale, sufficiente. L'algoritmo effettua quindi di norma più passate e termina quando in una passata non viene effettuato alcuno scambio, il ché significa che l'array è ordinato. In figura 7.2 è illustrato un esempio di applicazione del *bubble sort*, mentre il seguente esempio implementa l'algoritmo.

Esempio

Scrivere un programma che legga un array di interi di dimensione arbitraria, letta anch'essa in input, e lo ordini in senso crescente applicando l'algoritmo *bubble sort*.

Soluzione

```
#include <cstdlib>     /* exit, EXIT_FAILURE, EXIT_SUCCESS */
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <new>
using std::nothrow;
```

[4]Si noti che, previa inclusione dell'header file `cmath`, è possibile utilizzare la funzione `fabs(x)` che torna il valore assoluto dell'argomento.

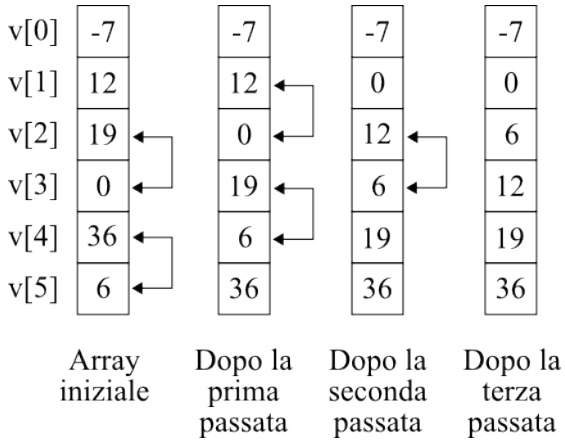

Figura 7.2: Esempio di applicazione dell'algoritmo di ordinamento *bubble sort* a un array di sei elementi.

```
int* alloca_array(const int dim = 1)
{
  return new (nothrow) int[dim];
}

void dealloca_array(int*& v)
{
  delete[] v;
  v = NULL;
}

void stampa_array (const int* const v, const int dim)
{
  cout << "Stampa dell'array...\n";
  for (int i=0; i<dim; i++)
    cout << "v[" << i << "]=" << v[i] << endl;
}

void leggi_array(int* const v, const int dim)
{
  cout << "inserisci gli elementi dell'array...\n";
  for (int i=0; i<dim; i++){
    cout << "v[" << i << "]=";
    cin >> v[i];
  }
}

void scambia (int& a, int& b)
{
```

```
  int temp = a;
  a = b;
  b = temp;
}

void bubble_sort (int* const v, const int d)
{
  bool scambio;

  do{
    scambio = false;
    for (int i=0; i<d-1; i++)
      if (v[i]>v[i+1]){
        scambia(v[i], v[i+1]);
        scambio = true;
      }
  } while (scambio);
}


int main()
{
  // v è il nome dell'array nonché il puntatore al primo elemento
  int* v = NULL;
  int dim_array = 0;

  cout << "Inserisci la dimensione dell'array: ";
  cin >> dim_array;

  //allocazione dinamica dell'array
  v = alloca_array(dim_array);

  if (v == NULL)
    exit (EXIT_FAILURE);

  leggi_array(v, dim_array);

  bubble_sort(v, dim_array);

  cout << "L'array ordinato:\n";
    stampa_array (v, dim_array);

  //deallocazione dell'array
  dealloca_array(v);

  return EXIT_SUCCESS;
}
```

Come nel caso della ricerca lineare, il programma definisce il puntatore a intero `v`, che diventerà, all'atto dell'allocazione della memoria, il puntatore al primo elemento dell'array. La dimensione di `v` è `dim_array`, dichiarato subito dopo `v`.

Dopo aver letto `dim_array`, segue l'allocazione dinamica della memoria. Anche in questo programma, la funzione `leggi_array` ritorna o l'indirizzo del primo elemento della sequenza allocata oppure, in caso di fallimento, il puntatore `NULL`. Il programma controlla quindi l'eventuale fallimento dell'operazione di allocazione confrontando `v` con `NULL` e, in caso di uguaglianza, termina il programma attraverso l'istruzione `exit` con argomento `EXIT_FAILURE`.

Qualora l'allocazione vada a buon fine, viene invocata la funzione `leggi_array` e, subito dopo, la funzione `bubble_sort`, che implementa l'omonimo algoritmo di ordinamento. L'algoritmo è costituito da un ciclo `do-while` la cui condizione di continuazione è che sia avvenuto almeno uno scambio tra elementi adiacenti nell'array. La prima istruzione del ciclo imposta la variabile booleana `scambio` a `true` e definisce un ciclo `for` che rappresenta la singola *passata*. A ogni iterazione `i` del ciclo `for`, vengono infatti confrontati gli elementi `v[i]` e `v[i+1]` e, qualora risultasse `v[i] > v[i+1]`, i due elementi vengono scambiati. Lo scambio degli elementi è demandato alla funzione `scambia`.

Il programma stampa quindi l'array ordinato tramite la funzione `stampa_array` dealloca l'array tramite la funzione `dealloca_array` e ritorna `EXIT_SUCCESS`. Si noti il primo argomento di `stampa_array`, `const int* const v`. Si tratta di un puntatore costante a valore puntato costante. Questo trova giustificazione nel fatto che `stampa_array` deve limitarsi a stampare gli elementi di `v`, per cui non deve possedere i diritti di modifica dei suoi elementi né, tanto meno, di modifica del valore del puntatore al primo elemento.

Esercizio

Scrivere una versione del precedente programma che implementi l'algoritmo *bubble sort* per array dei tipi base `char`, `short`, `int`, `float` e `double`, facendo uso delle funzioni template.

Esercizio

Scrivere la funzione `bubble_sort_inv` che ordini un array di tipo double dall'elemento più grande all'elemento più piccolo. Si noti che è possibile procedere nel seguente modo: 1) si definisce una funzione `inverti_array` che inverte un array; 2) si definisce la funzione `bubble_sort_inv` che semplicemente invoca in sequenza la funzione `bubble_sort` e la funzione `inverti_array`.

Esercizio

Si escogiti e si implementi un algoritmo alternativo al *bubble sort* per l'ordinamento di un array.

Esercizio

Si escogiti e si implementi un altro algoritmo alternativo al *bubble sort* per l'ordinamento di un array.

7.1.5 L'algoritmo della ricerca binaria

L'algoritmo della ricerca binaria è una algoritmo di ricerca di un elemento in un array e risulta, in genere, sensibilmente più efficiente della ricerca lineare. A differenza della ricerca lineare, tuttavia, l'algoritmo della ricerca binaria presuppone un array ordinato.

L'algoritmo effettua il primo confronto dell'elemento da cercare con l'elemento medio dell'array. Se il confronto ha successo, l'algoritmo termina. Altrimenti si controlla se l'elemento da cercare è più piccolo dell'elemento medio. In tal caso, l'elemento da cercare, se esiste, si trova sicuramente nella prima metà dell'array. Pertanto, l'algoritmo scarta la parte destra dell'array, cioè quella che inizia con l'elemento medio e termina con l'ultimo elemento, e reitera il procedimento di ricerca sulla prima parte dell'array. Qualora l'elemento da cercare fosse più grande dell'elemento medio, si scarterebbe la parte destra dell'array e si reitererebbe il procedimento sulla parte sinistra.

Il seguente esempio implementa l'algoritmo della ricerca binaria in un array di interi.

```
#include <cstdlib>     /* exit, EXIT_FAILURE, EXIT_SUCCESS */
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <new>
using std::nothrow;


int* alloca_array(const int dim = 1)
{
  return new (nothrow) int[dim];
}

void dealloca_array(int*& v)
{
  delete[] v;
```

```
  v = NULL;
}

void stampa_array (const int* const v, const int dim)
{
  cout << "Stampa dell'array...\n";
  for (int i=0; i<dim; i++)
    cout << "v[" << i << "]=" << v[i] << endl;
}

void leggi_array(int* const v, const int dim)
{
  cout << "inserisci gli elementi dell'array...\n";
  for (int i=0; i<dim; i++){
    cout << "v[" << i << "]=";
    cin >> v[i];
  }
}

void scambia (int& a, int& b)
{
  int temp = a;
  a = b;
  b = temp;
}

void bubble_sort (int* const v, const int d)
{
  bool scambio;

  do{
    scambio = false;
    for (int i=0; i<d-1; i++)
      if (v[i]>v[i+1]){
        scambia(v[i], v[i+1]);
        scambio = true;
      }
  } while (scambio);
}

bool ricerca_binaria(const int* const v, const int d, const int n)
{
  int isx = 0;
  int idx = d-1;
  int iM = (d-1)/2;
  bool trovato = false;

  while (isx <= idx && trovato == false)
  {
```

```
    if (n == v[iM])
      trovato = true;
    else
      if (n < v[iM])
        idx = iM-1;
      else
        isx = iM+1;
    iM = (isx + idx) / 2;
  }
  return trovato;
}

int main()
{
  // v è il nome dell'array nonché il puntatore al primo elemento
  int* v = NULL;
  int dim_array = 0;
  // a è l'elemento da cercare
  int a;

  cout << "Inserisci la dimensione dell'array: ";
  cin >> dim_array;

  //allocazione dinamica dell'array
  v = alloca_array(dim_array);

  if (v == NULL)
    exit (EXIT_FAILURE);

  leggi_array(v, dim_array);

  cout << "Inserisci un elemento da cercare: ";
  cin >> a;

  // La ricerca binaria richiede un array ordinato
  bubble_sort(v, dim_array);

  cout << "L'array ordinato:\n";
  stampa_array (v, dim_array);

  bool trovato = ricerca_binaria(v, dim_array, a);

  if (trovato)
    cout << "L'elemento " << a << " e' contenuto nell'array\n";
  else
    cout << "L'elemento " << a << " non e' contenuto nell'array\n";

  //deallocazione dell'array
  dealloca_array(v);
```

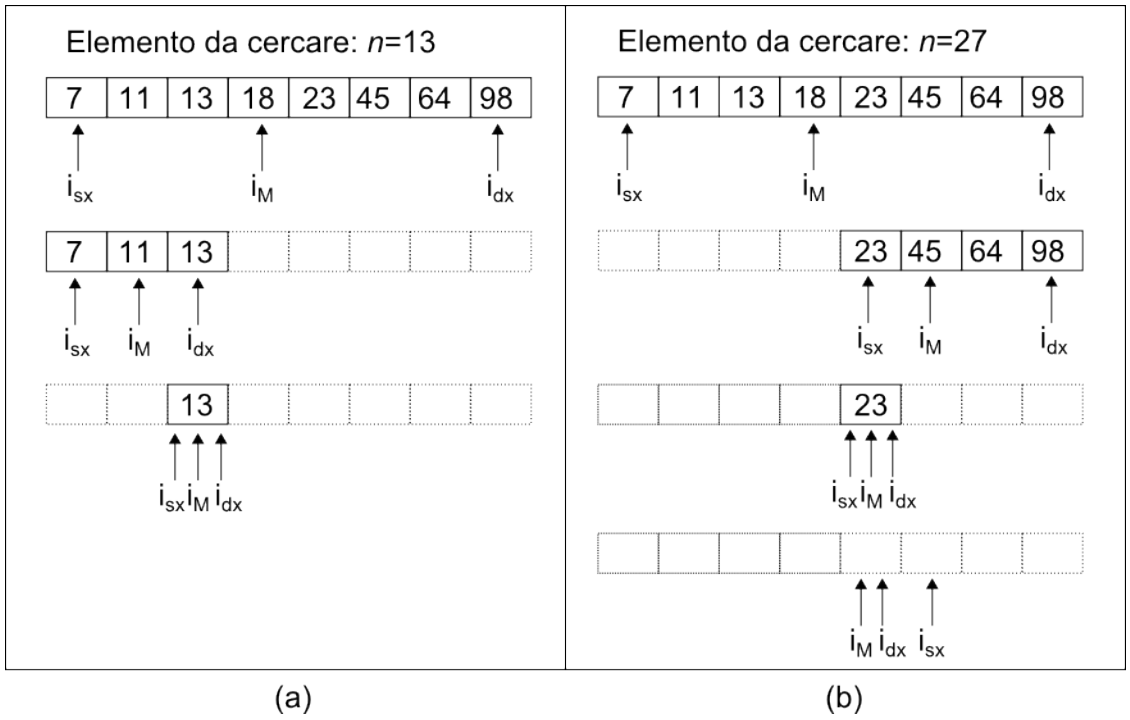

Figura 7.3: Esempi di applicazione dell'algoritmo della ricerca binaria a un array di 8 elementi. (a) Caso in cui l'elemento da cercare è contenuto nell'array. (b) Caso in cui l'elemento da cercare non è contenuto nell'array, caso in cui si verifica la situazione $i_{dx} < i_{sx}$.

```
  return EXIT_SUCCESS;
}
```

Si noti, innanzitutto, l'applicazione dell'algoritmo *bubble sort* al fine di ordinare l'array prima che venga invocata la funzione `ricerca_binaria`. Ai fini implementativi, l'algoritmo della ricerca binaria utilizza tre indici:

- `isx`, che rappresenta l'indice del primo elemento dell'array (quello più a sinistra);
- `idx`, che rappresenta l'indice dell'ultimo elemento dell'array (quello più a destra);
- `iM`, che rappresenta l'indice dell'elemento medio tra `isx` e `idx`.

Inizialmente `isx` e `idx` corrispondono rispettivamente al primo e all'ultimo indice dell'array e `iM` all'elemento medio. Se l'elemento da cercare `n` coincide con `v[iM]`,

l'algoritmo termina e ritorna una valore booleano che indica il successo della ricerca. Se, invece, risulta `n < v[iM]`, allora s'impone `idx = iM - 1` in modo da restringere la ricerca alla parte sinistra dell'array. In caso risulti `n > v[iM]`, s'impone `isx = iM + 1`, in modo da restringere la ricerca alla parte destra dell'array. Se l'elemento `n` non è contenuto nell'array, a un certo punto si verificherà la condizione `idx < isx`. In tal caso, ovviamente, l'algoritmo termina e ritorna una valore booleano che indica il fallimento della ricerca. La figura 7.3 illustra due esempi di applicazione dell'algoritmo, il primo in cui l'elemento viene trovato, il secondo un cui la ricerca dell'elemento fallisce.

Esercizio

Scrivere una funzione che implementi l'algoritmo della ricerca binaria e, in caso di successo, ritorni la posizione dell'elemento cercato all'interno dell'array. In caso di fallimento, la funzione deve ritornare il valore -1.

Esercizio

Scrivere una funzione che implementi l'algoritmo della ricerca binaria e che ritorni il numero di occorrenze dell'elemento cercato all'interno dell'array.

Esercizio

Scrivere una funzione che implementi l'algoritmo della ricerca binaria e ritorni il numero di occorrenze dell'elemento cercato all'interno dell'array e l'indice della prima occorrenza.

7.2 Matrici

Le matrici sono implementate in C++ come array bidimensionali. In questa sezione sono descritti i metodi di allocazione, statica e dinamica, e due semplici algoritmi: l'algoritmo della somma tra matrici e un algoritmo che consente di rappresentare una matrice attraverso un array unidimensionale.

7.2.1 Allocazione statica delle matrici

L'allocazione statica delle matrici, come nel caso degli array, richiede la specificazione delle due dimensioni (numero di righe e di colonne) come costanti. Inoltre, lo spazio necessario alla memorizzazione della matrice viene allocato in fase di caricamento e rimane allocato fino al termine del programma. Un esempio di dichiarazione statica di una matrice è il seguente:

```
const int righe = 3;
const int colonne = 4;
int M[righe][colonne];
```

L'accesso agli elementi della matrice avviene ancora tramite l'operatore di subscript. Ad esempio, per impostare al valore 7 la cella alla riga 1 e colonna 2, si scrive:

```
M[1][2] = 7;
```

mentre per azzerare tutti gli elementi della matrice è sufficiente un doppio ciclo `for`, come il seguente:

```
for (int i=0; i<righe; i++)
    for (int j=0; j<colonne; j++)
        M[i][j] = 0;
```

7.2.2 Allocazione dinamica delle matrici

Le considerazioni sullo sfruttamento della memoria per gli array unidimensionali valgono, a maggior ragione, per le matrici. All'allocazione statica delle matrici è quindi da preferire l'allocazione dinamica, poiché consente di liberare la memoria a *run time*.

Una matrice dinamica si dichiara come puntatore a puntatore al tipo di dato degli elementi. Ad esempio, una matrice dinamica di interi si dichiara nel seguente modo:

```
int** M;
```

In tal modo, leggendo sempre da destra verso sinistra, il nome della matrice è un puntatore a `int*`, che è a sua volta un puntatore a intero. L'allocazione della memoria di una matrice di `n` righe ed `m` colonne avviene nel seguente modo:

1. si alloca innanzitutto un array di `n` elementi di tipo `int*`; a tal fine si considera l'istruzione `M = new (nothrow) int* [n]`;

2. per ogni elemento di M (cioè, per ogni `M[i]`, con `i` che va da `0` a `n-1`), che è un puntatore a intero `int*`, si alloca un array di `m` elementi di tipo `int` attraverso la seguente istruzione: `M[i] = new (nothrow) int [m]`;

La figura 7.4 illustra un esempio di allocazione dinamica di una matrice di 3 righe e 4 colonne. Come mostra la figura, `M` è il puntatore al primo elemento di un array di puntatori. I puntatori di tale array, che si ottengono deferenziando M tramite l'operatore di subscript, sono, a loro volta, puntatori al primo elemento

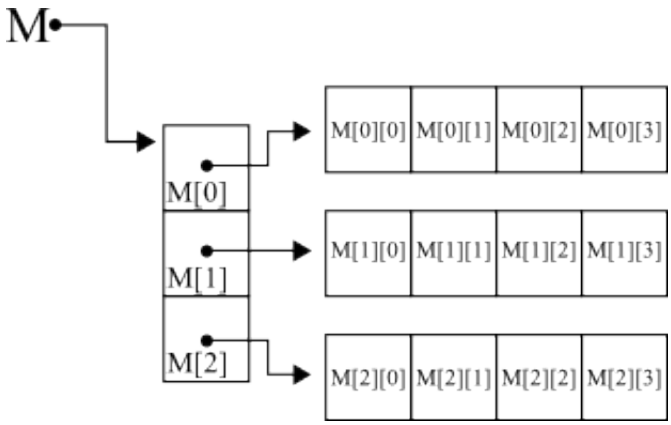

Figura 7.4: Esempio di allocazione dinamica di una matrice di 3 righe e 4 colonne come array bidimensionale. M, il nome della matrice, è il puntatore al primo elemento di un array di puntatori. I puntatori di tale array, che si ottengono deferenziando M tramite l'operatore di subscript, sono, a loro volta, puntatori al primo elemento di ulteriori array che rappresentano le righe della matrice.

di ulteriori array che rappresentano le righe della matrice. In altri termini, M[0] è il nome, o equivalentemente, il puntatore, della prima riga, M[1] della seconda e così via. L'accesso ai dati si ottiene deferenziando ulteriormente gli elementi riga. Ad esempio, M[0][1] è il secondo elemento della prima riga o, equivalentemente, l'elemento alla riga 0 e alla colonna 1.

Si noti che nel caso degli array bidimensionali non è garantita la contiguità degli elementi in memoria. La contiguità, tuttavia, continua a valere per i singoli elementi riga.

La deallocazione della memoria procede, invece, al contrario dell'allocazione:

1. si deallocano prima le righe `M[i]`;

2. si dealloca quindi l'array di puntatori `M`.

Il seguente esempio implementa due funzioni per l'allocazione e la deallocazione di un array bidimensionale.

Esempio

Scrivere un programma che allochi dinamicamente un array bidimensionale di interi, acquisisca il numero di righe e di colonne, nonché i vari elementi da input, e lo stampi in output in forma matriciale.

Soluzione

```
#include <cstdlib>     /* exit, EXIT_FAILURE, EXIT_SUCCESS */
#include <iomanip>
using std::setw;
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <new>
using std::nothrow;


void dealloca_matrice(int**& M, const int n)
{
  for (int i=0; i<n; i++)
    delete[] M[i];

  delete[] M;
  M = NULL;
}

int** alloca_matrice(const int n = 1, const int m = 1)
{
  int** M;

  M = new (nothrow) int* [n];
  if (M == NULL)
    return NULL;

  for (int i=0; i<n; i++)
  {
    M[i] = new (nothrow) int [m];
    if (M[i] == NULL)
    {
      dealloca_matrice(M, i);
      return NULL;
    }
  }

  return M;
}

// leggendo da destra verso sinistra, M è un puntatore costante a
    puntatore costante a intero
void leggi_matrice(int *const *const M, const int n, const int m)
// si scrivere anche int *const*const M{
  for (int i=0; i<n; i++)
```

```
    for (int j=0; j<m; j++)
    {
      cout << "M[" << i << "][" << j << "] = ";
      cin >> M[i][j];
    }
}
// leggendo da destra verso sinistra,
// M è un puntatore costante a puntatore costante a intero costante
void stampa_matrice (const int *const *const M, const int n, const
    int m, const int width = 4)
// si scrivere anche int const *const *const M
// oppure anche      int const*const*const M
{
  for (int i=0; i<n; i++)
  {
    for (int j=0; j<m; j++)
      cout << setw(width) << M[i][j];
    cout << "\n";
  }
}


int main()
{
  int** A = NULL; // nome della matrice
  int n, m;       // numero di righe e colonne

  //allocazione e lettura della matrice
  cout << "Inserisci il numero di n e il numero di m: ";
  cin >> n >> m;

  A = alloca_matrice(n, m);
  if (A == NULL)
  {
    cout << "Errore fatale: allocazione della memoria non riuscita
        ...\n";
    exit (EXIT_FAILURE);
  }

  leggi_matrice(A, n, m);

  //stampa della matrice e deallocazione
  cout << "Stampa della matrice...\n";

  stampa_matrice(A, n, m, 4);
  dealloca_matrice(A, n);

  return EXIT_SUCCESS;
}
```

Si noti il primo parametro della funzione `leggi_matrice`: `int *const *const M`. Si tratta di (leggendo da destra verso sinistra) un puntatore costante a puntatore costante a intero. Questo implica che la funzione non possa modificare il puntatore `M`, né i puntatori `M[i]`, dove `i` è l'indice di riga. Per quanto riguarda la sintassi, posizionare il qualificatore `const` dopo il simbolo di puntatore significa proprio far sì che il puntatore sia un puntatore costante. Si noti, infine, che gli spazi sono facoltativi, motivo per cui la seguente definizione di parametro risulta equivalente alla precedente: `int *const*const M`.

La funzione `leggi_matrice` può comunque modificare i valori puntati, al fine di impostarli ai valori letti in input. Al contrario, la funzione `stampa_matrice` non deve possedere neanche questa facoltà, visto che il suo unico compito è quello di stampare i valori della matrice in output. Per tale motivo, i permessi sulla matrice devono essere assolutamente restrittivi e questo è reso possibile grazie al parametro `const int *const *const M`. Rispetto al parametro della funzione `leggi_matrice`, in questo caso compare il qualificatore `const` anche prima del tipo base `int`, il ché significa che anche i valori puntati sono costanti. Si noti che la precedente sintassi è equivalente alle seguenti:

```
int const *const *const  M"
int const*const*const M
```

In altri termini, posizionare a destra o a sinistra del tipo base il qualificatore `const` implica rendere gli elementi puntati di sola lettura.

7.2.3 L'algoritmo della somma tra matrici

L'algoritmo della somma è probabilmente l'algoritmo più semplice sugli array bidimensionali. Il seguente esempio lo implmenta.

Esempio

Scrivere un programma che legga in input due matrici e le relative dimensioni, calcoli la matrice somma e la stampi in output.

Soluzione

```
#include <cstdlib>     /* exit, EXIT_FAILURE, EXIT_SUCCESS */
#include <iomanip>
using std::setw;
#include <iostream>
using std::cin;
```

```
using std::cout;
using std::endl;
#include <new>
using std::nothrow;


void dealloca_matrice(int**& M, const int n)
{
  for (int i=0; i<n; i++)
    delete[] M[i];

  delete[] M;
  M = NULL;
}

int** alloca_matrice(const int n = 1, const int m = 1)
{
  int** M;

  M = new (nothrow) int* [n];
  if (M == NULL)
    return NULL;

  for (int i=0; i<n; i++)
  {
    M[i] = new (nothrow) int [m];
    if (M[i] == NULL)
    {
      dealloca_matrice(M, i);
      return NULL;
    }
  }

  return M;
}

// leggendo da destra verso sinistra, M è un puntatore costante a
   puntatore costante a intero
void leggi_matrice(int *const *const M, const int n, const int m)
// si scrivere anche int *const*const M
{
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
    {
      cout << "M[" << i << "][" << j << "] = ";
      cin >> M[i][j];
    }
}
```

```
// leggendo da destra verso sinistra, M è un puntatore costante a
    puntatore costante a intero costante
void stampa_matrice (const int *const *const M, const int n, const
    int m, const int width = 4)
// si scrivere anche int const *const *const M
// oppure anche       int const*const*const M
{
  //M = NULL;
  //M[0] = NULL;
  //M[0][1] = 0;

  for (int i=0; i<n; i++)
  {
    for (int j=0; j<m; j++)
      cout << setw(width) << M[i][j];
    cout << "\n";
  }
}

void somma_matrici(const int *const *const M1, const int *const *
    const M2, int *const *const S, const int n, const int m)
{
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
      S[i][j] = M1[i][j] + M2[i][j];
}

int main()
{
  int** A;       // nome della matrice
  int** B;       // nome seconda matrice
  int** S;       // matrice somma
  int n, m;      // numero di righe e colonne

  //allocazione e lettura delle matrici
  cout << "Inserisci il numero di righe (n) e il numero di colonne (m
      ): ";
  cin >> n >> m;

  A = alloca_matrice(n, m);
  if (A == NULL)
  {
    cout << "Errore fatale: allocazione della memoria non riuscita
        ...\n";
    exit (EXIT_FAILURE);
  }

  B = alloca_matrice(n, m);
  if (B == NULL)
```

```
  {
    cout << "Errore fatale: allocazione della memoria non riuscita
        ...\n";
    exit (EXIT_FAILURE);
  }

  S = alloca_matrice(n, m);
  if (S == NULL)
  {
    cout << "Errore fatale: allocazione della memoria non riuscita
        ...\n";
    exit (EXIT_FAILURE);
  }

  cout << "Inserisci la prima matrice:\n";
  leggi_matrice(A, n, m);

  cout << "Inserisci la seconda matrice:\n";
  leggi_matrice(B, n, m);

  somma_matrici(A, B, S, n, m);

  //stampa della matrice e deallocazione
  cout << "Stampa della matrice somma...\n";

  stampa_matrice(S, n, m, 4);

  dealloca_matrice(A, n);
  dealloca_matrice(B, n);
  dealloca_matrice(S, n);

  return EXIT_SUCCESS;
}
```

Esercizio

Scrivere un programma che calcoli il determinante di una matrice.

Esercizio

Scrivere un programma che implementi il prodotto tra matrici.

Esercizio

Scrivere un programma che legga in input due matrici di interi e determini se la più piccola è contenuta nella più grande.

7.2.4 Gestione delle matrici tramite array unidimensionali

L'uso degli array bidimensionali presenta l'inconveniente della non contiguità della memoria allocata. Il problema, che in applicazioni sequenziali è di fatto irrilevante, può essere significativo in contesti di calcolo parallelo, laddove si voglia distribuire il carico computazionale su più elementi di processazione al fine di ridurre i tempi di calcolo. In questi casi, la contiguità della mamoria consente una più semplice parallelizzazione del codice sorgente.

È possibile, in vista di eventuali future parallelizzazioni del codice, implementare le matrici attraverso array unidimensionali. Infatti, se si dispongono gli elementi di una matrice di n righe ed m colonne in un array per righe, cioè inserendo la prima riga, quindi la seconda e così via, esiste una semplice relazione biunivoca che lega la posizione degli elementi nella matrice specificati con gli indici di riga e colonna, i e j, a quella dello stesso elemento nell'array unidimensionale specificato col l'unico indice, k. Tale relazione è la seguente:

$$k = i * m + j$$

In virtù di questa relazione, il seguente esempio re-implementa il precedente algoritmo della somma di matrici facendo uso esclusivamente di array unidimensionali, senza variare in alcun modo il comportamento del programma a *run-time*.

Esempio

Scrivere un programma che legga in input due matrici e le relative dimensioni, calcoli la matrice somma e la stampi in output. Utilizzare array unidimensionali anzichè array bidimensionali per la gestione delle matrici.

```
#include <iomanip>
using std::setw;
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <new>
using std::nothrow;


int* alloca_matrice(const int n = 1, const int m = 1)
{
  const int dim = n * m;
  return new (nothrow) int[dim];
}
```

```
void dealloca_matrice(int*& M, const int n)
{
  delete[] M;
  M = NULL;
}

void leggi_matrice(int* const M, const int n, const int m)
{
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
    {
      cout << "M[" << i << "][" << j << "] = ";
      cin >> M[i*m + j];
    }
}

void stampa_matrice(const int* const M, const int n, const int m,
    const int width = 4)
{
  for (int i=0; i<n; i++)
  {
    for (int j=0; j<m; j++)
      cout << setw(width) << M[i*m + j];
    cout << "\n";
  }
}

void somma_matrici(const int* const M1, const int* const M2, int*
    const S, const int n, const int m)
{
  int dim = m*n;
  for (int k=0; k<dim; k++)
      S[k] = M1[k] + M2[k];
}

int main()
{
  int* A;         // nome della matrice implementata come array 1D
  int* B;         // nome seconda matrice implementata come array 1D
  int* S;         // matrice somma implementata come array 1D
  int n, m;       // numero di righe e colonne

  //allocazione e lettura delle matrici
  cout << "Inserisci il numero di righe (n) e il numero di colonne (m
      ): ";
  cin >> n >> m;

  A = alloca_matrice(n, m);
  if (A == NULL)
```

```
  {
    cout << "Errore fatale: allocazione della memoria non riuscita
        ...\n";
    exit (EXIT_FAILURE);
  }

  B = alloca_matrice(n, m);
  if (B == NULL)
  {
    cout << "Errore fatale: allocazione della memoria non riuscita
        ...\n";
    exit (EXIT_FAILURE);
  }

  S = alloca_matrice(n, m);
  if (S == NULL)
  {
    cout << "Errore fatale: allocazione della memoria non riuscita
        ...\n";
    exit (EXIT_FAILURE);
  }

  cout << "Inserisci la prima matrice:\n";
  leggi_matrice(A, n, m);

  cout << "Inserisci la seconda matrice:\n";
  leggi_matrice(B, n, m);

  somma_matrici(A, B, S, n, m);

  //stampa della matrice e deallocazione
  cout << "Stampa della matrice somma...\n";

  stampa_matrice(S, n, m, 4);

  dealloca_matrice(A, n);
  dealloca_matrice(B, n);
  dealloca_matrice(S, n);

  return EXIT_SUCCESS;
}
```

Esercizio

Scrivere un programma che calcoli il determinante di una matrice utilizzando la rappresentazione tra matrici tramite array 1D.

Esercizio

Scrivere un programma che implementi il prodotto tra matrici utilizzando la rappresentazione tra matrici tramite array 1D.

Esercizio

Scrivere un programma che legga in input due matrici di interi e determini se la più piccola è contenuta nella più grande utilizzando la rappresentazione delle matrici tramite array 1D.

Capitolo 8

Definizione di nuovi tipi di dato

Come visto nel capitolo 7, il C++ consente di aggregare dati dello stesso tipo tramite il tipo array. Tuttavia, nelle applicazioni pratiche, spesso accade che i dati coinvolti si presentino naturalmente come aggregati ma di tipi differenti. Come vedremo nelle prossime sezioni, il C++ permette la definizione di nuovi tipi di dato, eventualmente anche molto complessi. Tutto ciò rende il linguaggio estensibile, nonché estremamente espressivo.

8.1 Definizione di nuovi tipi strutturati con struct

In C++ consente di definire nuovi tipi di dato i cui membri possono essere di tipi e dimensioni differenti attraverso il costrutto `struct`, come segue:

```
struct type_name {
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    .
    .
    .
} object_names;
```

`type_name` è il nome della struttura, cioè del nuovo tipo di dato, `object_names` (opzionale), è un insieme di identificatori per le variabili del nuovo tipo di dato appena definito. All'interno del blocco è definita la lista dei membri, ognuno caratterizzato dal tipo e dal nome. Si consideri, per esempio, la seguente struttura:

```
struct elemento {
  string nome;  //nome dell'elemento
```

```
  int elettroni;//numero di elettroni per livello
  int Z;        //numero atomico
  double A;     //massa atomica
} ;

elemento H;
elemento Li, Na;
```

definisce un nuovo tipo di dato struttrato chiamato `elemento` composto da quattro membri: il nome dell'elemento, il numero di elettroni per livello, il numero e la massa atomica. Il nuovo tipo di dato è quindi utilizzato per la dichiarazione delle tre variabili `H`, `Li` e `Na`. Si noti che il precedente esempio è del tutto equivalente al seguente:

```
struct elemento {
  string nome;  //nome dell'elemento
  int elettroni;//numero di elettroni per livello
  int Z;        //numero atomico
  double A;     //massa atomica
} H, Li, Na;
```

dove le variabili sono dichiarate contestualmente alla definizione del nuovo tipo di dato. In questo caso il nome della struttura è opzionale. È buona norma, tuttavia, non omettere il nome della struttura in vista di eventuali successive dichiarazioni di nuove variabili.

L'accesso ai membri della struttura avviene tramite l'operatore `.` specificando, nell'ordine, il nome della variabile, il punto e il nome del dato membro. As esempio, per settare i membri della variabile `H` si procede nel seguente modo:

```
H.nome = "idrogeno";
H.elettroni = 1;
H.Z = 1;
H.A = 1.008;
```

Esempio

Scrivere un programma che definisca il nuovo tipo di dato `elemento_chimico` come nell'esempio precedente, escluso il nome, e un array di elementi chimici di lunghezza `l`. Il programma inizializzi da input gli elementi dell'array, legga in input un ulteriore elemento e verifichi se tale elemento è contenuto nell'array tramite l'algoritmo della ricerca lineare.

Soluzione

```
#include <cstdlib>     /* exit, EXIT_FAILURE, EXIT_SUCCESS */
#include <iomanip>
using std::setw;
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
#include <new>
using std::nothrow;


//Definizione del nuovo tipo di dato elemento_chimico
//Il nuovo tipo ha visibilità globele poiché definito
//al difuori di ogni blocco di codiche
struct elemento_chimico {
  int e;          //numero di elettroni per livello
  int Z;          //numero atomico
  double A;       //massa atomica
};



elemento_chimico* alloca_array(const int dim = 1)
{
  return new (nothrow) elemento_chimico[dim];
}

void dealloca_array(elemento_chimico*& E)
{
  delete[] E;
  E = NULL;
}

void stampa_array(const elemento_chimico* const E, const int dim,
    const int width = 4)
{
  for (int i=0; i<dim; i++)
    cout  << setw(width) << "Elemento " << i << " (e Z A): "
          << setw(width) << E[i].e
          << setw(width) << E[i].Z
          << setw(width) << E[i].A
          << endl;
}

void leggi_array(elemento_chimico* const E, const int dim)
{
```

```
  cout << "inserisci gli elementi dell'array...\n";
  for (int i=0; i<dim; i++){
    cout << "e Z A = ";
    cin >> E[i].e >> E[i].Z >> E[i].A;
  }
}

bool ricerca_lineare(const elemento_chimico* const E, const int dim,
   const elemento_chimico elem_chim)
{
  for (int i=0; i<dim; i++)
    if (elem_chim.e == E[i].e && elem_chim.Z == E[i].Z && elem_chim.A
        == E[i].A)
      return true;

  return false;
}


int main()
{
  // v è il nome dell'array nonché il puntatore al primo elemento
  elemento_chimico* sequenza_elementi = NULL;
  int dim_array = 0;

  // elem è l'elemento da cercare
  elemento_chimico elem;

  cout << "Inserisci la dimensione dell'array di elementi chimici: ";
  cin >> dim_array;

  //allocazione dinamica dell'array
  sequenza_elementi = alloca_array(dim_array);

  if (sequenza_elementi == NULL)
    exit (EXIT_FAILURE);

  leggi_array(sequenza_elementi, dim_array);

  cout << "Inserisci l'elemento da cercare: ";
  cout << "e Z A = ";
  cin >> elem.e >> elem.Z >> elem.A;

  bool trovato = ricerca_lineare(sequenza_elementi, dim_array, elem);

  if (trovato)
    cout << "L'elemento e' contenuto nell'array\n";
  else
    cout << "L'elemento non e' contenuto nell'array\n";
```

```
  cout << "L'array inserito e' infatti...\n";
  stampa_array (sequenza_elementi, dim_array, 4);

  //deallocazione dell'array
  dealloca_array(sequenza_elementi);

  return EXIT_SUCCESS;
}
```

Il programma definisce il nuovo tipo di dato `elemento_chimico` con i tre dati membro `e`, `Z` ed `A` di cui, i primi due di tipo `int`, l'ultimo di tipo `double`. Le varie funzioni sono un adattamento del programma della ricerca lineare di un elemento in un array di interi visto nel capitolo 7 al nuovo tipo di dato `elemento_chimico`. Così, ad esempio, la funzione `alloca_array` torna un puntatore a `elemento_chimico`, mentre la funzione `ricerca_lineare` effettua il controllo tra elementi accedendo ai dati membro della struttura.

Esercizio

Scrivere un programma che definisca il nuovo tipo di dato `elemento_chimico` come nell'esempio precedente e un array di elementi chimici di lunghezza `l`. Il programma inizializzi in modo random gli elementi dell'array e li stampi in output. Il programma ordini quindi in senso crescente in base al peso atomico gli elementi dell'array sfruttando l'algoritmo di ordinamento a bolle, studiato nel capitolo 7, e stampi nuovamente la sequenza.

8.1.1 Puntatori a struttute

Come per i tipi di dato predefiniti, anche per i nuovi tipi di dato creati con `struct` è possibile definire puntatori. Ad esempio, il seguente codice definisce un puntatore all'oggetto `H` di tipo `elemento_chimico`, definito nell'esempio precedente:

```
elemento_chimico H;
elemento_chimico* pH;
```

L'unica particolarità da tenere in considerazione quando si accede ai dati membro di una struttura attraverso un puntatore è che bisogna usare l'operatore `->` (anziché l'operatore `.`). Così, ad esempio, l'accesso ai membri della struttura `elemento_chimico` attraverso il puntatore `pH` può avviene nel seguente modo:

```
pH->e = 1;
pH->Z = 1;
pH->A = 1.008;
```

8.2 Definizione di *tipi alias* con typedef e using

In C++, per comodità, è possibile definire *tipi alias* al fine di identificare un tipo di dato con un altro nome. Esistono due modi per creare un alias. Il primo usa la parola riservata `typedef`:

```
typedef existing_type new_type_name;
```

dove `existing_type` è un qualsiasi tipo base o strutturato, mentre `new_type_name` è il nuovo nome, cioè l'alias, per `existing_type`. Per esempio, il seguente esempio

```
typedef char byte;
typedef double real;
typedef int* pInt;
typedef float vec3 [3];
```

definisce quattro alias: `byte`, `real`, `pInt` e `vec3`, rispettivamente per i tipi `char`, `double`, `int*` e `float[3]`. Si noti la particolarità della definizione dell'alias `vec3`, che rappresenta un array di tre `float`. In tale definizione, il numero di elementi dell'array è specificato, in modo forse anti-intuitivo, vicino al nome dell'alias anziché vicino al nome del tipo esistente.

Rispetto al linguaggio C, oltre al `typedef`, il C++ offre una seconda sintassi per definire tipi alias:

```
using new_type_name = existing_type;
```

L'esempio precedente può essere quindi riscritto in modo del tutto equivalente come segue:

```
using byte = char;
using real = double;
using pInt = int*;
using vec3 = float [3];
```

Si noti, in particolare, la definizione dell'alias `vec3`. A differenza della definizione con `typedef`, la definizione che fa uso della direttiva `using` permette di specificare la dimensione dell'array indicando la dimensione vicino al tipo esistente (`float` nel caso specifico), il ché rende probabilmente la definizione dell'alias più intuitiva.

8.3 Il tipo enum

I tipi enumerativi sono tipi di dato particolari che definiscono un insieme finito di possibili valori per le variabili. Tali valori, specificati all'atto della definizione del tipo, sono detti enumeratori. La sintassi è la seguente:

```
enum type_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

dove `object_names` è una lista, opzionale, di variabili di tipo `type_name`.

Si consideri il seguente esempio in cui viene definito il tipo `colors_t`:

```
enum colors_t {
    black,
    blue,
    green,
    cyan,
    red,
    purple,
    yellow,
    white
};
```

È quindi possibile instanziare e utilizzare variabili del tipo appena creato, ad esempio come nel seguente esempio di codice:

```
colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

Una caratteristica interessante dei tipi `enum` è che i valori definiti corrispondono a valori nimerici interi e possono essere usati come costanti. Se non diversamente specificato, il valore intero equivalente al primo valore è 0 ,l' equivalente al secondo è 1, e così via. Pertanto, nel tipo `colors_t`, `nero` equivale a 0, `blu` equivale a 1 e così via.

Nel seguente esempio:

```
enum months_t {
    january=1, february, march,
    april, may, june,
    july, august, september,
    october, november, december
};
```

poiché il primo valore è forzato a essere 1, i valori assunti dagli enumeratori variano tra 1 e 12 (anziché tra 0 e 11).

Capitolo 9

Input/Output su FILE

Il linguaggio C++ associa a ogni dispositivo fisico sul quale è possibile effettuare operazioni di input/output un dispositivo logico detto *stream*. Le operazioni di input/output si effettuano chiamando in causa lo stream, il quale offre un'interfaccia per il dispositivo fisico indipendente dal dispositivo stesso. In tal modo è possibile operare su dispositivi differenti nello stesso modo, senza preoccuparsi dei dettagli fisici dei dispositivi coinvolti nelle operazioni.

Un *file* è una sequenza di bit, organizzati in blocchi minimi di un byte, che è possibile scrivere su un dispositivo di output, ad esempio un hard disk. Esistono essenzialmente due tipi di file: file di testo, come i listati dei programmi C++, e file binari, come gli eseguibili dei programmi.

Il C++ fornisce i seguenti stream per l'input/output su file:

- ofstream: stream per scrivere su file
- ifstream: stream per leggere da file
- fstream : stream per leggere e scrivere da/su file.

Possiamo usare il file allo stesso modo in cui siamo già abituati a usare `cin` e `cout` utilizzando gli operatori di prelievo (`>>`) e inserimento (`<<`) dallo stream, con la sola differenza che dobbiamo associare questi stream di input e di output con i file fisici. L'esempio seguente chiarirà ogni dubbio.

Esempio

Scrivere un programma crei un file di testo e vi inserisca la stringa "Prova di scrittura su file di testo".

stream	valore di default
`ios::out`	Apre il file per operazioni di output
`ios::binary`	Apre il file in modalità binaria
`ios::ate`	Imposta il cursore in coda anziché in testa
`ios::app`	Non sovrascrive gli elementi già presenti nel file
`ios::trunc`	Cancella il contenuto del file

Tabella 9.1: Opzioni per l'apertura di un file. È ammissibile una qualsiasi combinazione in OR bit a bit.

Soluzione

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

int main () {
  ofstream myfile;
  myfile.open ("esempio.txt");
  myfile << "Prova di scrittura su file di testo";
  myfile.close();
  return 0;
}
```

Questo programma crea uno stream di output chiamato `myfile` e lo associa a un file su disco grazie all'istruzione `myfile.open`. Quest'ultima istruzione crea un file su disco di nome `esempio.txt` nella cartella corrente, cioè in quella cartella da cui è stato eseguito il programma. Se si vuole posizionare il file in un'altra cartella, è sufficiente specificare il percorso completo. L'istruzione successiva utilizza l'operatore di inserimento nello stream, `<<`, per inserire la stringa "Prova di scrittura su file di testo" nello stream `myfile`, che è associato al file `prova.txt`. In tal modo, la stringa viene effettivamente aggiunta al file. La funzione `myfile.close()` chiude il file di modo che non sia più possibile effettuare operazioni di input o di output su di esso.

9.1 Aprire e chiudere un file

Come abbiamo visto nel precedente esempio, per aprire un file è sufficiente usare l'istruzione

```
stream_associato_al_file.open (nome_file);
```

In realtà, la funzione `open`, accetta un ulteriore parametro opzionale:

Stream	Valore di default	In caso di ulteriori valori
`ofstream`	`ios::out`	I valori sono combinati in OR bit a bit
`ifstream`	`ios::in`	I valori sono combinati in OR bit a bit
`fstream`	`ios::in | ios::out`	Il valore di default viene sovrascritto

Tabella 9.2: Valori di default per gli stream di input/output.

```
stream_associato_al_file.open (nome_file, modo);
```

il cui valore può essere inizializzato con una qualsiasi combinazione (in OR bit a bit) delle opzioni elencate in tabella 9.1. Per esempio, se si vuole aprire il file `esempio.bin` in modalità binaria per aggiungere dati, si può usare la seguente combinazione:

```
ofstream myfile;
myfile.open ("esempio.bin", ios::out | ios::app | ios::binary);
```

I valori di default per il parametro `modo` sono elencati in tabella 9.2.

Dichiarazione dello stream e apertura del file possono essere eseguite in forma compatta in un'unica istruzione, come nel seguente esempio:

```
ofstream myfile ("esempio.bin", ios::out | ios::app | ios::binary);
```

Per verificare se un file è stato aperto correttamente, è possibile utilizzare la funzione `is_open`, come nel seguente esempio:

```
if (myfile.is_open())
{
    /* ok, proceed with output */
}
```

Infine, quando le operazioni di input/output sono state completate, è necessario chiudere il file tramite la funzione `close`:

```
myfile.close();
```

9.2 File di testo

I file di testo sono quei file in cui non è stato considerato il valore `ios::binary` al momento dell'apertura. Le operazioni di scrittura su file di testo sono effettuate in modo analogo a come avveniva con lo stream di output `cout`.

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("esempio.txt");
  if (myfile.is_open())
  {
    myfile << "Questa e' una linea di testo.\n";
    myfile << "Questa e' un'altra linea.\n";
    myfile.close();
  }
  else cout << "Impossibile aprire il file";
  return 0;
}
```

Le operazioni di lettura su un file di testo sono effettuate in modo analogo a come avveniva con lo stream di input `cin`.

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string linea;
  ifstream myfile ("esempio.txt");
  if (myfile.is_open())
  {
    while ( getline (myfile, linea) )
      cout << linea << '\n';

    myfile.close();
  }

  else cout << "Impossibile aprire il file";

  return 0;
}
```

In quest'ultimo esempio viene utilizzata la funzione `getline` che legge un'intera riga per volta dal file. In caso di successo, ritorna `true`. Quando il ciclo raggiunge la fine del file, la funzione `getline` ritorna `false` e il ciclo `wile` termina.

La tabella 9.3 lista altre funzioni che possono essere invocate su stream.

Funzione	Significato
`bad()`	Torna `true` se l'operazione di input/output fallisce
`fail()`	Simile a `bad()` ma con controllo errori di formattazione[1]
`eof()`	Ritorna `true` se un file aperto in lettura ha raggiunto la fine
`good()`	Ritorna `false` nei casi in cui le funzioni precedenti tornano `true`

Tabella 9.3: Funzioni di controllo su stream.

9.3 File binari

Quando si lavora con i file binari si utilizzano le funzioni `write` e `read`. La loro sintassi è la seguente:

```
write ( memory_block, size );
read ( memory_block, size );
```

dove:

- `memory_block` è un `char*` (puntatore a `char`) e rappresenta l'indirizzo dell'array di byte[2] nel quale sono memorizzati i dati;
- `size` è un valore intero che specifica il numero di byte che deve essere letto o scritto.

Si consideri il seguente esempio:

```
// reading an entire binary file
#include <iostream>
#include <fstream>
using namespace std;

int main ()
{
  fstream file;
  char nome_file[] = "file.bin";
  int n = 100;

  file.open(nome_file, ios::out|ios::binary);
  file.write( (char*) &n, sizeof(n) );
  file.close();

  file.open(nome_file, ios::in|ios::binary);
  if (file.is_open())
  {
```

[2]Si rammenta che il byte è l'elemento atomico poiché non è possibile memorizzare o leggere un dato di dimensione più piccole.

```
    file.read( (char*) &n, sizeof(n) );
    cout << "Il valore letto e' " << n << endl;
    file.close();
  }
  else
    cout << "Impossibile aprire il file!\n";

  return 0;
}
```

Il programma dichiara un file di tipo `fstream`. L'array `nome_file` contiene il percorso del file su disco. In questo caso, il file su disco si chiama `file.bin`. Il file viene aperto in modalità `out` e `binary`. L'istruzione `file.write` scrive un l'intero `n` sul file. Il primo parametro, infatti, è l'indirizzo (convertito in `char*`) della variabile intera `n`, mentre `sizeof(n)` restituisce il numero di byte che occupa `n`. Il file viene quindi chiuso. Successivamente, il file viene riaperto in input e il dato contenuto nel file viene letto con l'istruzione `file.read`.

Elenco delle figure

Elenco delle tabelle