Fondamenti di Programmazione

(con linguaggio di riferimento C++)

GIANFRANCO ROSSI, TATIANA ZOLO
Università di Parma
Dip. di Matematica
43100 Parma (Italy)
gianfranco.rossi@unipr.it

14 dicembre 2009

Indice

Capitolo 1

Introduzione alla programmazione

> "L'arte di programmare è l'arte di organizzare la complessità...",
> "...dobbiamo organizzare le computazioni in modo tale che le nostre limitate capacità siano sufficienti a garantire che gli effetti delle computazioni stesse siano quelli voluti."([5]).

1.1 Algoritmi

1.1.1 Problemi ed algoritmi

Consideriamo come punto di partenza un *problema*. I problemi possono essere di vario tipo: semplici operazioni aritmetiche (come moltiplicazione di due numeri interi, minimo comune multiplo, ecc.), operazioni aritmetiche più complesse (come estrazione della radice quadrata, calcolo delle radici di una equazione di secondo grado, ecc.), ma anche problemi in altri campi, non necessariamente matematici, come la ricerca di un nome in un elenco telefonico, la ricerca di un tragitto ferroviario tra due località, il prelievo di soldi con un bancomat, la preparazione di un piatto di cucina, la gestione di un magazzino (arrivo di prodotti già presenti / non presenti, calcolo giacenze, ...), e così via.[1]

Prefiggiamoci come scopo la risoluzione di un dato problema. L'approccio sarà il seguente: trovare (se esiste) un *metodo risolutivo* per il problema dato che possa essere capito ed applicato da un *esecutore* (umano, meccanico, ...). Questo ci porta ad introdurre la nozione di algoritmo.

[1] Per una trattazione più approfondita della nozione di problema nel contesto dello sviluppo di programmi si veda, ad esempio, il testo [3], cap. 15, in cui vengono introdotte, tra l'altro, la nozione di *specifica* di un problema, una classificazione dei problemi—di ricerca, di decisione, di ottimizzazione—, le nozioni di spazio di ricerca, di correttezza, ecc..

Algoritmo: sequenza finita di *istruzioni* che specificano come certe *operazioni elementari* debbano susseguirsi nel tempo per risolvere una data *classe di problemi*.

Per *operazioni elementari* si intendono le operazioni che si assume siano note all'esecutore, e che quindi quest'ultimo è in grado di eseguire. Le *istruzioni* indicano in genere richieste di azioni rivolte all'esecutore e che da questo devono essere capite ed eseguite. Per *classe di problemi* si intende una formulazione del problema indipendente dagli specifici dati su cui si opera (ad esempio, non la somma dei numeri 34 e 57, ma la somma di due numeri interi qualsiasi).

La Figura 1.1 mostra schematicamente la relazione intercorrente tra problema, algoritmo ed esecutore.

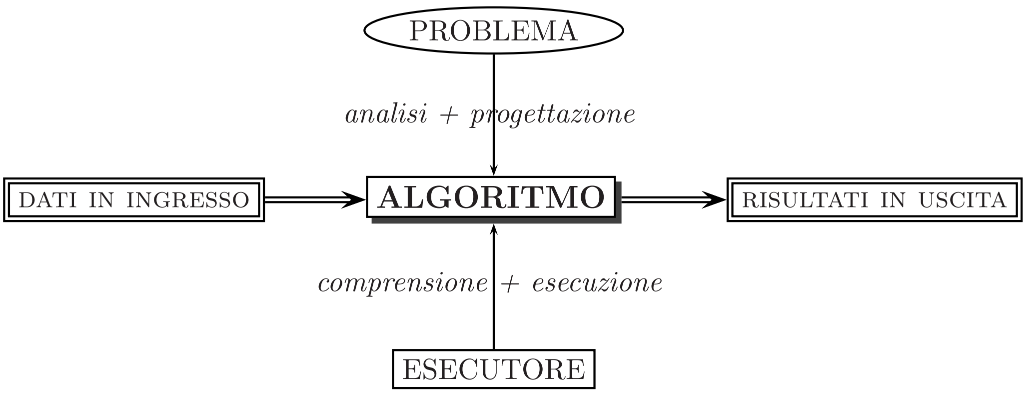


Figura 1.1: Dal problema all'algoritmo.

Alla formulazione dell'algoritmo si arriva attraverso una fase di *analisi* del problema e quindi di *progettazione* del procedimento risolutivo adeguato (in cui si passa dalla specifica alle scelte per la sua risoluzione e quindi alla definizione dell'algoritmo risolutivo). L'esecuzione dell'algoritmo comporta invece la capacità da parte dell'esecutore di capire le istruzioni e quindi di applicarle correttamente. L'esecuzione dell'algoritmo, inoltre, richiederà, in generale, che vengano forniti i dati specifici su cui operare (dati in ingresso o *input*) e produrrà altri dati come risultati della sua esecuzione (*output*).

Esempi di algoritmi si incontrano anche nella vita quotidiana (ed in tali casi l'esecutore è umano):

- somma di due numeri in colonna—un'operazione elementare in questo caso può essere "la somma di due cifre", mentre un'istruzione può essere "passa alla colonna immediatamente a sinistra della attuale", oppure "se il risultato è maggiore di 10 allora . . . ", ecc.;
- una ricetta di cucina;

- le istruzioni di montaggio di un pezzo meccanico.

L'esecutore esegue in modo acritico le istruzioni dell'algoritmo: non sa nulla del problema nella sua globalità; capisce soltanto le istruzioni e le operazioni elementari; è privo di "buon senso". Di conseguenza, la descrizione dell'algoritmo deve essere precisa, completa e non ambigua: le istruzioni devono poter essere interpretate in modo univoco dall'esecutore. Ad esempio, un'istruzione del tipo "scegli un ϵ piccolo a piacere" può essere ambigua: il significato di "piccolo a piacere" dipende in generale dal contesto in cui ci si trova. Analogamente, l'istruzione "aggiungi sale quanto basta" si basa chiaramente sul buon senso dell'esecutore e perciò non è adeguata per una formulazione algoritmica.

Oltre alle proprietà di eseguibilità e non ambiguità di un algoritmo, alcune altre proprietà fondamentali degli algoritmi sono:

- *correttezza*: l'esecuzione dell'algoritmo porta realmente alla soluzione del problema dato;
- *efficienza*: quanto "costa" l'esecuzione di un algoritmo in termini di risorse consumate (in particolare, se l'esecutore è un calcolatore, il tempo di CPU richiesto e lo spazio di memoria occupato);
- *finitezza*: in pratica normalmente si richiede che l'algoritmo termini in un tempo finito, e cioè che la sequenza di istruzioni che caratterizzano l'algoritmo sia una sequenza finita.[2]

Approfondimento (Studio degli algoritmi). Dato un problema (o, meglio, una classe di problemi), esiste un algoritmo che permetta di risolverlo? A questa e ad altre domande simili risponde la *Teoria della computabilità*. Questa branca dell'informatica utilizza opportuni modelli di calcolo (macchine di Turing, automi a stati finiti, ecc.) per dimostrare l'esistenza o meno di algoritmi per una data classe di problemi. Ad esempio, stabilire in modo algoritmico se, dato un programma ed i suoi dati di input, l'esecuzione del programma con questi dati termina o no in un tempo finito non è calcolabile (cioè non ammette un algoritmo). Due testi classici su questi argomenti sono [7] e [10].

Se un problema è risolvibile in modo algoritmico, quanto "costa" eseguire l'algoritmo? Esistono algoritmi più efficienti per risolvere lo stesso problema? Quale è il limite inferiore di efficienza per questi algoritmi? A queste e ad altre domande simili risponde la *Teoria della complessità*. Questa branca dell'informatica si occupa di determinare le risorse richieste dall'esecuzione di un algoritmo per risolvere un dato problema. Le risorse più frequentemente considerate sono il tempo (quanti passi sono richiesti per completare l'algoritmo) e lo spazio (quanta memoria è richiesta per immagazzinare i dati usati dall'algoritmo). Per approfondire l'argomento si possono consultare, ad esempio, i testi [2] e [4]. Per una trattazione più snella si veda [3], cap. 8.

[2] Per una discussione sulla presenza o meno del requisito di finitezza nella definizione di algoritmo si veda ad esempio `http://en.wikipedia.org/wiki/Algorithm/\#Termination`.

1.1.2 Descrizione di algoritmi

Un algoritmo deve essere comprensibile al suo esecutore. Per questo si deve utilizzare un *linguaggio di descrizione* dell'algoritmo adeguato alle caratteristiche dell'esecutore. Si può distinguere tra:

- esecutore umano: l'algoritmo viene descritto utilizzando il *linguaggio naturale* (eventualmente strutturato), o un linguaggio grafico come quello dei *diagrammi di flusso* (si veda 1.1.3);
- calcolatore: l'algoritmo viene descritto utilizzando un linguaggio di programmazione. La descrizione di un algoritmo in un linguaggio di programmazione costituisce un *programma*. Dunque il calcolatore è un esecutore (automatico) di algoritmi, descritti in un linguaggio di programmazione.

Vediamo un esempio di semplice algoritmo descritto in linguaggio naturale. Assumiamo per ora che il significato delle diverse istruzioni ed il modo in cui l'algoritmo viene eseguito sia quello "intuitivo" (aiutati in questo anche dall'uso del linguaggio naturale).

Esempio 1.1 (Moltiplicazione per somme)

Problema*: dati due numeri interi maggiori o uguali a* 0*,* `a` *e* `b`*, determinarne il prodotto* `p`*.*
Operazioni elementari*: somma, confronto (useremo l'operatore "="), decremento unitario.*
Procedimento risolutivo*—informale:*

$$p = a \cdot b = a + a + \cdots + a, \ b \text{ volte.}$$

Dati*:* `a`*,* `b`*,* `p`*: interi maggiori o uguali a* 0*;* `a` *e* `b` *sono dati di input,* `p` *è un dato di output (il risultato fornito dall'algoritmo).*
Descrizione algoritmo*: in linguaggio naturale strutturato (esecutore umano).*

```
1.  assegna 0 a p
2.  se b = 0 vai all'istruzione 6
3.  assegna p+a a p
4.  assegna b-1 a b
5.  vai all'istruzione 2
6.  fine
```

`a`*,* `b` *e* `p` *sono delle* variabili *(ritorneremo più avanti, nel capitolo* 2.2*, su questo fondamentale concetto dei linguaggi di programmazione). L'algoritmo è composto da* 6 *istruzioni che vengono eseguite in sequenza a partire dalla istruzione 1, a meno di modifica esplicita dell'ordine di esecuzione (come, ad esempio, nelle istruzioni 2 e 5). Le istruzioni del tipo "*`vai all'istruzione xxx`*" sono dette* istruzioni di salto *ed* `xxx` *è un'*etichetta*, cioè un nome simbolico che identifica in modo univoco un'istruzione.*

È possibile verificare "a mano" il funzionamento dell'algoritmo mostrato sopra, attribuendo dei valori numerici ai suoi dati di input. Ad esempio, se `a` vale 2 e `b` vale 3, i valori di `a`, `b` e `p` variano come mostrato nella tabella di Figura 1.2.

a	b	p
2	**3**	0
	2	2
	1	4
	0	**6**

Figura 1.2: Esecuzione di 2×3 con l'algoritmo di *moltiplicazione per somme.*

Si noti che il fatto che il numero di istruzioni presenti nella descrizione di un algoritmo sia finito non implica necessariamente che l'algoritmo termini in un tempo finito. Ad esempio, il seguente algoritmo, composto da solamente quattro istruzioni

```
1.  assegna 0 a r
2.  assegna r+1 a r
3.  vai all'istruzione 2
4.  fine
```

chiaramente non termina in un tempo finito. Questo è dovuto alla presenza di un "ciclo" senza una condizione di uscita esplicita.

Un *ciclo* è un gruppo di istruzioni eseguite ripetutamente. Ad esempio, nell'algoritmo di sopra le istruzioni *2* e *3* costituiscono un ciclo. Nell'algoritmo dell'Esempio 1.1 le istruzioni dalla *2* alla *5* costituiscono un ciclo.

L'utilizzo del linguaggio naturale, ancorché strutturato, mette presto in evidenza i suoi limiti non appena l'algoritmo da descrivere diventa più complesso. In questi casi è facile che la descrizione risulti contorta e ambigua; inoltre, la descrizione spesso non evidenzia, ma anzi nasconde, la "struttura" dell'algoritmo.

Risulta pertanto opportuno, in generale, utilizzare strumenti di descrizione degli algoritmi più precisi. Per questo si utilizzano opportuni *formalismi*, caratterizzati da un *linguaggio*, costituito dall'insieme delle istruzioni sintatticamente corrette che è possibile utilizzare, e da precise *regole d'esecuzione* delle istruzioni, che ne determinano il significato.

Nel prossimo paragrafo presenteremo sinteticamente un formalismo di descrizione di algoritmi molto intuitivo (e a dire il vero non completamente "formalizzato"): i *diagrammi di flusso.* Rivolgeremo quindi la nostra attenzione ai formalismi di descrizione di algoritmi più interessanti per i nostri scopi, i linguaggi di programmazione, concentrandoci, in particolare, sul linguaggio C++.

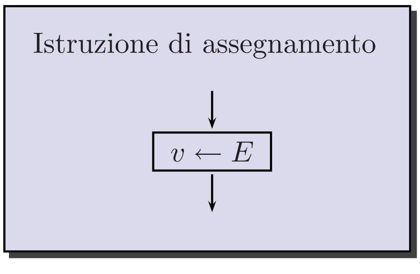
1.1.3 I diagrammi di flusso (concetti di base)

I diagrammi di flusso (o "flow chart") sono un formalismo grafico di descrizione degli algoritmi. I diversi tipi di istruzioni che caratterizzano questo formalismo sono rappresentati tramite *blocchi* di varia forma, connessi da frecce (per questo si parla anche di *schemi a blocchi*). Si tratta di una descrizione di algoritmi rivolta principalmente ad un esecutore umano che, in generale, ha il pregio di mettere ben in evidenza il "flusso del controllo" dell'algoritmo, cioè come si susseguono le sue diverse istruzioni, e quindi la "struttura" dell'algoritmo, cioè la presenza di cicli, di salti, di biforcazioni, ecc..

In questo paragrafo descriveremo brevemente gli elementi base del formalismo dei diagrammi di flusso, che possono essere schematicamente ridotti a tre tipi di blocchi—rettangolari, romboidali, ovali—e alle semplici regole con cui i diversi blocchi si possono comporre per formare strutture di controllo più complesse. La descrizione è volutamente non esauriente. Il suo scopo è essenzialmente quello di mostrare un formalismo di descrizione di algoritmi, alternativo ai linguaggi di programmazione, che sia semplice ed intuitivo e che ci permetta quindi di concentrare l'attenzione sulla nozione di algoritmo, piuttosto che sulle particolarità del formalismo stesso.

Blocchi rettangolari

I blocchi rettangolari vengono utilizzati principalmente per rappresentare istruzioni di assegnamento:[3]



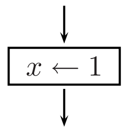
dove v è una variabile ed E è una espressione. Un'espressione, in generale, può essere una *costante* (ad es. 1), o una *variabile* (ad es. x, y), o un'*espressione composta*, cioè un'espressione ottenuta combinando costanti e variabili attraverso operatori (ad es., operatori aritmetici come in $x + 1$).

[3] Più in generale, i blocchi rettangolari possono rappresentare istruzioni che comportano una qualche modifica dello stato globale della computazione. Nel caso specifico dell'istruzione di assegnamento questa modifica riguarda il valore della variabile che compare nella parte sinistra dell'assegnamento.

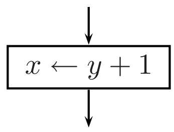
Il significato, ovvero la semantica (informale), di tale blocco è: *assegna alla variabile v il risultato della valutazione dell'espressione E*. Da notare che prima viene valutata E e poi avviene l'assegnamento; conseguentemente il vecchio valore di v viene perso (ritorneremo sulla nozione di variabile e di assegnamento nel capitolo successivo).

Esempi:

- assegna alla variabile x il valore costante 1:



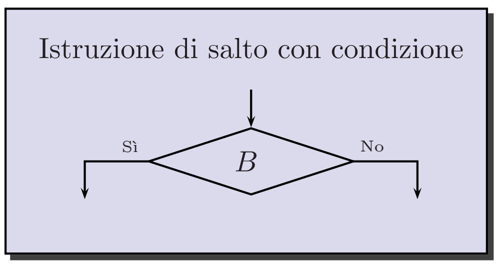
- assegna alla variabile x il risultato della valutazione dell'espressione $y + 1$:



  Si noti che se l'espressione alla destra di $\leftarrow$ fosse $x+1$, l'assegnamento avrebbe semplicemente l'effetto di incrementare di 1 il valore di x.

Blocchi romboidali

I blocchi romboidali vengono utilizzati per rappresentare istruzioni di salto con condizione. La loro forma generale è la seguente:

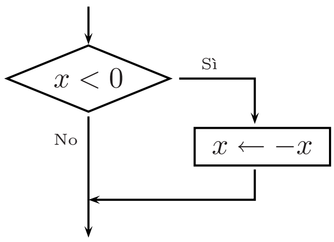


dove B è un'espressione booleana, cioè un'espressione costruita tramite operatori di relazione (quali "=", ">", "<", ...) e connettivi logici (*and*, *or*, *not*), il cui risultato può essere soltanto di tipo booleano (*vero* o *falso*).

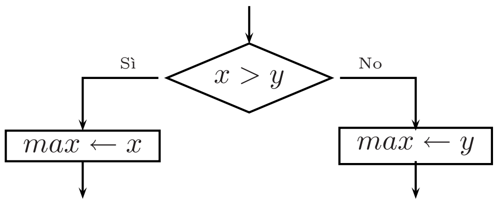
Il significato di tale blocco è: *valuta l'espressione B; se B ha valore "vero", continua dalla parte etichettata con* Sì*; altrimenti, cioè se B ha valore "falso", continua dalla parte etichettata con* No.

Esempi:

- (*esecuzione condizionale*) se il valore della variabile x è minore di 0 assegna ad x il suo opposto e passa all'istruzione successiva; altrimenti, passa direttamente all'istruzione successiva:



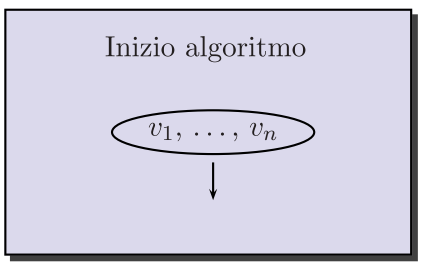
- (*biforcazione*) se il valore della variabile x è maggiore del valore della variabile y assegna alla variabile max il valore di x; altrimenti assegna alla variabile max il valore di y:



Blocchi ovali

I blocchi ovali vengono utilizzati per rappresentare l'inizio e la fine dell'algoritmo.

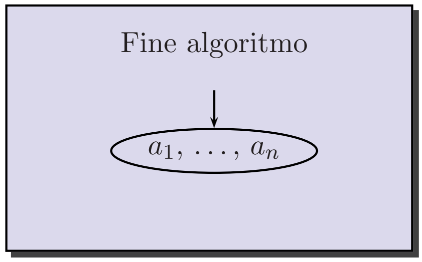
Blocco ovale iniziale



dove $v_1, \ldots, v_n$ sono variabili ($n \geq 0$).

$v_1, \ldots, v_n$ rappresentano i dati in ingresso dell'algoritmo; i loro valori si assume siano assegnati "dall'esterno" prima dell'inizio dell'esecuzione (come succedeva, ad esempio, per a e b nell'algoritmo di moltiplicazione per somma). Questo blocco indica l'inizio dell'algoritmo e quindi è unico.

Blocco ovale finale



dove $a_1, \ldots, a_n$ sono variabili o costanti ($n \geq 0$).

I valori di $a_1, \ldots, a_n$ costituiscono i risultati prodotti dall'esecuzione dell'algoritmo (allo stesso modo in cui, ad esempio, p indicava il risultato nell'algoritmo di moltiplicazione per somma). Questo blocco indica la fine dell'algoritmo e ne possono esistere più di uno.

Esempio 1.2 (Moltiplicazione per somme–continuazione)
Riprendiamo l'algoritmo di moltiplicazione per somme precedentemente descritto (cfr. Esempio 1.1), basato su

$$p = a \cdot b = a + a + \cdots + a$$

dove a, b e p sono tre interi non negativi e la somma ha esattamente b addendi. In Figura 1.3 *è rappresentato il diagramma di flusso che descrive questo algoritmo.*

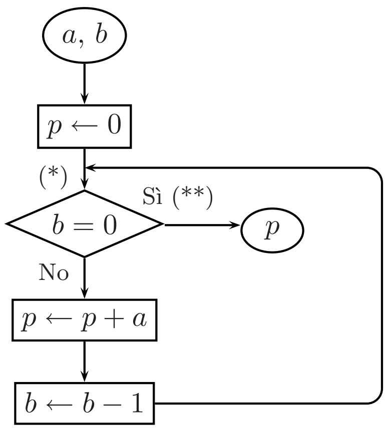


Figura 1.3: Diagramma di flusso dell'algoritmo di moltiplicazione per somme.

Regole d'esecuzione (informale)

Oltre al modo di funzionare di ciascuna singola istruzione, per capire il significato (operazionale) di un intero algoritmo bisogna anche specificare come il controllo passa da un'istruzione all'altra, e come si inizia e si finisce.

Nel caso dei diagrammi di flusso la specificazione di queste regole risulta particolarmente semplice. Per eseguire un algoritmo descritto da un diagramma di flusso si deve: iniziare dal blocco ovale iniziale e quindi eseguire i blocchi successivi, seguendo le frecce, fino a raggiungere un blocco ovale finale.

Struttura degli algoritmi: cicli

Nel diagramma di Figura 1.3 risulta chiara la presenza di un ciclo. Nella figura è stato evidenziato con (*) il *punto di ingresso* nel ciclo e con (**) il *punto di uscita* dal ciclo. "$b = 0$" rappresenta la *condizione di uscita* dal ciclo: l'iterazione prosegue finché "$b = 0$" non diventa vera. In un ciclo deve comparire almeno una condizione d'uscita, come condizione necessaria (anche se, in generale, non sufficiente) a garantire che il ciclo possa terminare in un tempo finito.

Osserviamo inoltre che in questo ciclo, nel caso in cui la condizione d'uscita sia falsa già dalla prima iterazione, i blocchi all'interno del ciclo vengono saltati (mai eseguiti). Sono realizzabili ovviamente anche cicli in cui la condizione d'uscita appare al termine del ciclo, che hanno la caratteristica di permettere l'esecuzione dei blocchi al loro interno almeno una volta.

In generale, si potranno avere algoritmi, e quindi diagrammi di flusso, con più cicli, strutturati in vario modo: uno di seguito all'altro (ovvero, "in cascata"), uno dentro all'altro (ovvero, "annidati"), ecc..

Casi particolari, ottimizzazioni

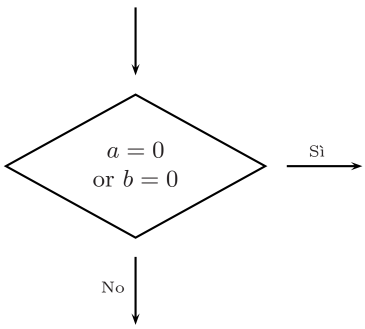
Uno stesso problema può essere risolto tramite più algoritmi, equivalenti dal punto di vista dei risultati prodotti, ma diversi dal punto di vista dell'efficienza di calcolo. Un'analisi (anche empirica) di un algoritmo prodotto può essere utile, in generale, per evidenziare casi particolari in cui l'algoritmo presenta un comportamento non soddisfacente (seppur corretto), e per i quali può essere opportuno prevedere una diversa formulazione.

Riferendoci ancora all'Esempio 1.1 analizziamo cosa accade nel caso particolare in cui uno dei due dati in ingresso sia `0`. La tabella in Figura 1.4 mostra che l'algoritmo funziona correttamente, ma nel secondo caso proposto risulta particolarmente inefficiente.

Per evitare che vengano ripetute somme con 0 si può modificare la condizione d'uscita dal ciclo utilizzando un'espressione booleana più complessa:

(i)			(ii)		
a	b	p	a	b	p
3	**0**	**0**	**0**	**3**	0
				2	0
				1	0
				0	**0**

Figura 1.4: Esecuzione di 3×0 (caso (i)) e 0×3 (caso (ii)) con l'algoritmo di *moltiplicazione per somme.*



In questo caso, se a o b sono 0, l'algoritmo termina immediatamente restituendo (correttamente) 0 come suo risultato.

In generale si osserva che si avrebbe un miglioramento di efficienza dell'algoritmo se si usasse come moltiplicatore il minore tra a e b (tanto più rilevante quanto più grande è la differenza tra a e b). Possiamo perciò pensare ad una variante dell'algoritmo di moltiplicazione per somme che provveda eventualmente a scambiare tra loro i due termini a e b in modo tale da avere come moltiplicatore sempre il minore tra i due (l'operazione di scambio non pregiudica la correttezza dell'algoritmo grazie alla proprietà di commutatività dell'operazione di moltiplicazione). In Figura 1.5 è mostrato il diagramma di flusso relativo a questa versione migliorata dell'algoritmo di moltiplicazione per somme. Lo scambio tra le due variabili a e b viene fatto (quando richiesto) utilizzando una variabile temporanea t.

Tra le ipotesi dell'algoritmo di moltiplicazione per somme vi era il fatto che i due valori di ingresso fossero non negativi. Proviamo a tralasciare questa ipotesi e a chiederci cosa succederebbe nel caso in cui fosse $b < 0$ (e $a > 0$). Non essendo verificata la condizione d'uscita dal ciclo, l'algoritmo proseguirebbe con le istruzioni contenute negli ultimi due blocchi; in particolare l'ultimo blocco comporta un decremento del valore di b. Risulta allora ovvio che in questo caso la condizione d'uscita non sarà mai verificata: si è creato quello che in gergo è chiamato *ciclo infinito*. L'algoritmo continuerà a ripetere "all'infinito" le istruzioni del ciclo, venendo così a mancare quella caratteristica fondamentale per un algoritmo che è il terminare in un tempo finito.

1.2 I linguaggi di programmazione

Un linguaggio di programmazione è un formalismo di descrizione di algoritmi eseguibile dal calcolatore. La descrizione di un algoritmo per un problema $\mathcal{P}$ in un linguaggio di programmazione $\mathcal{L}$, e cioè un *programma in* $\mathcal{L}$, è

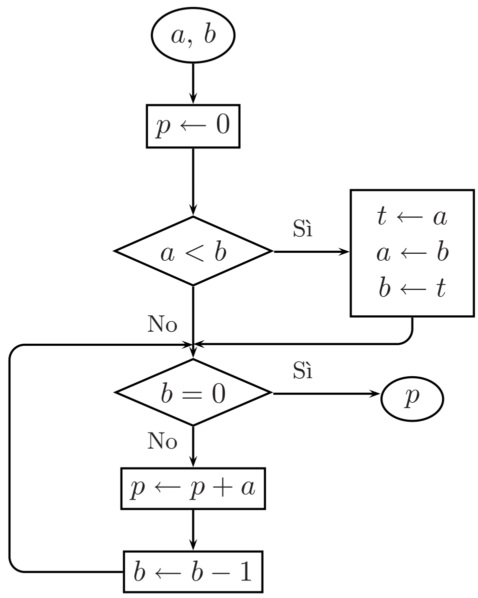


Figura 1.5: Algoritmo ottimizzato di moltiplicazione per somme.

costituito da una sequenza finita di istruzioni di $\mathcal{L}$ la cui esecuzione da parte del calcolatore porta (o dovrebbe portare) alla risoluzione di $\mathcal{P}$.

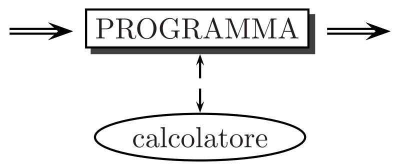
Un linguaggio di programmazione è caratterizzato formalmente da:

- una *sintassi*, la quale specifica la forma che possono avere le istruzioni e i programmi scrivibili con il linguaggio;
- una *semantica*, la quale specifica il significato e/o il funzionamento delle singole istruzioni e dei programmi realizzati con esse.

Per una trattazione approfondita delle problematiche inerenti la semantica dei linguaggi di programmazione si vedano, ad esempio, i testi [8] e [12].

Il fatto che un programma scritto in un linguaggio di programmazione sia *eseguibile* dal calcolatore significa che quest'ultimo sarà in grado di comprendere le istruzioni (sintatticamente corrette) e di eseguirle secondo la loro semantica.

Lo schema di Figura 1.1 assume perciò in questo caso la seguente forma:



Vedremo come questo schema generale possa essere ulteriormente precisato distinguendo tra vari tipi di linguaggi di programmazione e di modalitrà d'esecuzione di un programma da parte del calcolatore.

1.2.1 Linguaggi “a basso livello” e “ad alto livello”

Una prima (grossolana) classificazione dei linguaggi di programmazione si ottiene distinguendo tra linguaggi ad *alto livello* e linguaggi a *basso livello*, in base al livello di astrazione che essi forniscono rispetto al calcolatore.

La distinzione, in generale, non è così netta ed è possibile individuare varie gradazioni di “alto livello”. In questo testo indicheremo semplicemente con linguaggi “a basso livello” quelli che sono strettamente dipendenti da una specifica macchina hardware, e cioè il *linguaggio macchina* e il *linguaggio assembly*. Tutti gli altri linguaggi verranno genericamente indicati come linguaggi “ad alto livello”.[4]

Ogni diverso tipo di macchina hardware ha un proprio *linguaggio macchina*: le istruzioni dei programmi scritti in tale linguaggio sono lette ed eseguite (“interpretate”) direttamente dalla macchina hardware. Pertanto queste istruzioni sono scritte direttamente in notazione binaria (e cioè sequenze di 0 e 1) e quindi molto lontane dal linguaggio naturale e molto complesse da utilizzare per il programmatore.

Approfondimento (Linguaggio assembly). Generalmente per ogni linguaggio macchina esiste anche una versione più simbolica, detta *linguaggio assembly*. Essa utilizza nomi simbolici per codici di istruzioni ed indirizzi di memoria e la notazione decimale per rappresentare i numeri: questo la rende pertanto di più facile ed immediato utilizzo da parte del programmatore. Si tratta comunque ancora di un linguaggio a basso livello, dipendente dalla specifico tipo di macchina hardware. L’esecuzione di programmi scritti in linguaggio assembly richiede l’utilizzo di programmi traduttori, detti *assemblatori*, in grado di trasformare il programma scritto in linguaggio assembly nel corrispondente programma in linguaggio macchina (e quindi comprensibile per il calcolatore).

Per un approfondimento su linguaggio macchina e assembly, nonché, in generale, su tutto ciò che riguarda la macchina hardware, si veda ad esempio il testo [11].

I linguaggi ad alto livello nascondono, in modo più o meno completo, le caratteristiche proprie delle diverse macchine hardware ed offrono al programmatore una sintassi molto più vicina al linguaggio naturale rispetto a quanto possono fare i linguaggi macchina o assembly. Essi risultano pertanto più semplici da utilizzare ed indipendenti da una specifica macchina hardware.

L’esecuzione di un programma scritto in un linguaggio ad alto livello richiederà però la presenza di adeguati strumenti che rendano possibile la sua esecuzione sulla macchina hardware sottostante. Si distinguono solitamente due tecniche di realizzazione: la *compilazione* e l’*interpretazione* (si veda 1.2.3).

[4] Rispetto a questa classificazione il C ed il C++ risultano di livello “più basso” di altri linguaggi in quanto permettono, volendo, di accedere direttamente a certe caratteristiche della macchina hardware sottostante, rendendo, di fatto, i programmi che le utilizzano dipendenti da quest’ultima. In altri linguaggi, come ad esempio il Pascal, questa possibilità e del tutto impedita.

1.2.2 Linguaggi di programmazione esistenti

I linguaggi di programmazione ad alto livello esistenti attualmente sono moltissimi, sicuramente oltre il migliaio (per un elenco dettagliato si veda ad esempio la pagina Web dell'enciclopedia libera Wikipedia all'indirizzo `http://en.wikipedia.org/wiki/Timeline_of_programming_languages`). Qui di seguito diamo solo un brevissimo elenco indicando, in ordine cronologico, i nomi di alcuni dei linguaggi di programmazione più noti.

anni '50-'60:	Fortran
	Cobol
	LISP
	Basic
	PL/1
anni '70:	Pascal
	C
	Prolog
	Modula
anni '80:	C++
	Ada
	Perl
anni '90-2000:	Visual Basic
	Delphi
	Java
	C#

Rimandiamo alla letteratura specializzata sull'argomento (ad esempio [9]) qualsiasi descrizione ed analisi comparativa dei diversi linguaggi di programmazione oggi in uso.

Aggiungiamo soltanto che spesso si classificano i linguaggi in base al "paradigma di programmazione" da essi supportato. Si distingue principalmente tra quattro diversi paradigmi:

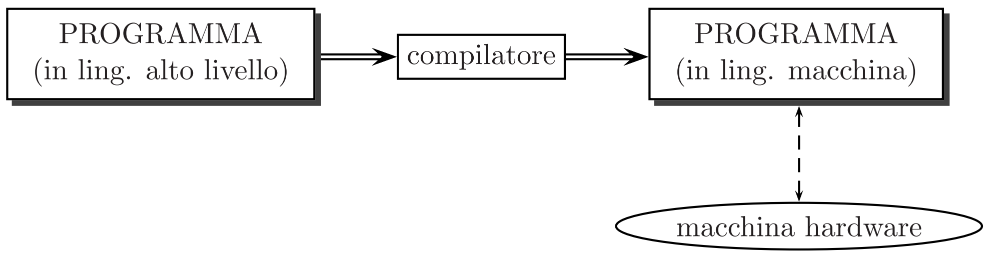
- *imperativo*
- *logico*
- *funzionale*
- *orientato agli oggetti.*

Anche per questo argomento rimandiamo alla letteratura specializzata.

In queste note utilizzeremo come linguaggio di riferimento il linguaggio C++, che può classificarsi come un linguaggio ad alto livello che supporta i paradigmi di programmazione imperativo e orientato agli oggetti. Precisamente, nella prima parte di queste note ci occuperemo soltanto degli aspetti del linguaggio relativi al paradigma imperativo, mentre nella seconda parte amplieremo il discorso anche a vari aspetti relativi alla programmazione orientata agli oggetti.

1.2.3 Modalità d'esecuzione: compilazione e interpretazione

La *compilazione* di un programma consiste nell'analisi delle istruzioni del programma che vengono lette e tradotte da un programma traduttore, il *compilatore*, e nella generazione delle istruzioni macchina in grado, quando verranno successivamente eseguite dalla macchina hardware, di realizzare il comportamento voluto. Sono solitamente realizzati in questo modo la maggior parte dei linguaggi di programmazione "convenzionali", come ad esempio il Pascal ed il C.

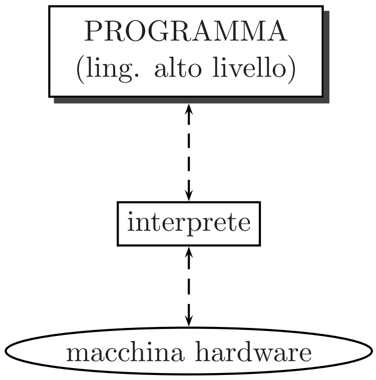


L'*interpretazione* consiste nell'analisi delle istruzioni del programma e nell'immediata esecuzione di esse tramite un apposito programma, l'*interprete*, in grado di realizzare per ciascuna di esse il comportamento previsto. L'interprete è dunque un programma che svolge ciclicamente le seguenti azioni, fino ad incontrare un'istruzione di terminazione dell'esecuzione:

- legge ed analizza l'istruzione corrente;
- esegue l'istruzione;
- passa all'istruzione successiva.[5]

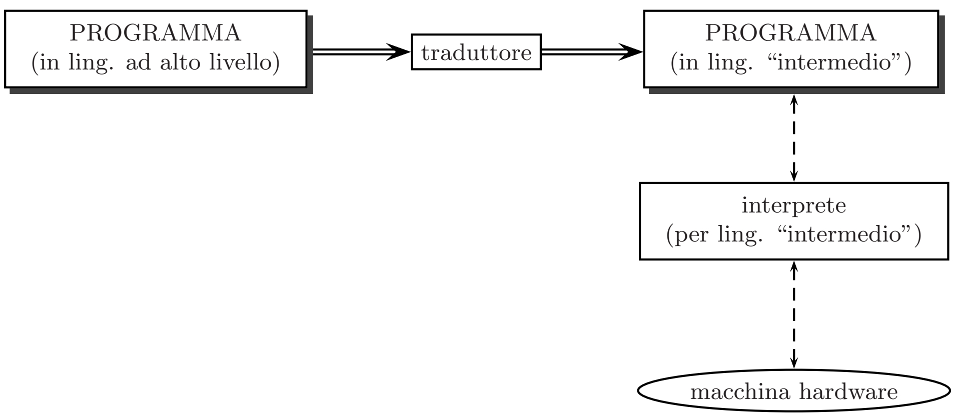
Sono solitamente realizzati in questo modo linguaggi di programmazione "non convenzionali", quali il Prolog o il LISP.

[5]Si osservi che anche la macchina hardware è di fatto un interprete, ma realizzato a livello hardware, piuttosto che software, ed in grado di eseguire istruzioni di un linguaggio molto semplice, il cosiddetto linguaggio macchina.



Se vogliamo fare un confronto tra queste due modalità d'esecuzione, seppur a livello molto superficiale, possiamo osservare che, poiché l'interpretazione richiede la presenza di un programma interprete che si frappone tra il programma da eseguire e l'esecutore vero e proprio, ovvero la macchina hardware, l'esecuzione di un programma tramite questa modalità risulterà in generale più lenta e comporterà una maggiore occupazione di memoria. D'altra parte, l'esecuzione tramite interpretazione rende il linguaggio di programmazione più adatto ad uno sviluppo dei programmi interattivo, proprio perché non vi è una fase di traduzione separata ed il debugging del programma utente può essere svolto direttamente dall'interprete sulle istruzioni del linguaggio ad alto livello.

Approfondimento (Tecniche miste). Le due modalità descritte sopra, e cioè interpretazione e compilazione, rappresentano due situazioni estreme. Spesso, vengono adottate soluzioni "miste" che combinano, a vari livelli, le due tecniche. In queste soluzioni è presente una prima fase di traduzione dal linguaggio ad alto livello ad un *linguaggio "intermedio"*, un linguaggio molto più vicino ai linguaggi macchina, ma ancora indipendente dalle caratteristiche della specifica macchina hardware. Segue poi una seconda fase in cui il programma in linguaggio intermedio viene eseguito tramite interpretazione, su una specifica macchina hardware.



Questo è ad esempio l'approccio tipicamente adottato per l'implementazione del linguaggio Java: un programma Java viene tradotto in un programma equivalente nel linguaggio intermedio *Byte-code*, che verrà poi interpretato tramite un programma interprete, denominato "Java Virtual Machine" (JVM) (più precisamente, l'interprete realizza la *macchina astratta* JVM, che offre il Byte-code come proprio "linguaggio macchina"). Analogamente, un programma in linguaggio Prolog viene normalmente prima tradotto in un programma in un apposito linguaggio intermedio le cui istruzioni verranno poi interpretate da un interprete denominato "Warren Abstract Machine" (WAM).

In queste note ci occuperemo soltanto di linguaggi ad alto livello compilati (specificatamente del C++), senza peraltro addentrarci nelle problematiche specifiche dell'implementazione di un linguaggio di programmazione, che esulano dai nostri scopi. Per una trattazione più completa e formale delle nozioni di interpretazione, macchina astratta ed, in generale, di implementazione di un linguaggio, si veda, ad esempio, il testo [6].

1.3 Il linguaggio C++

1.3.1 Dal C al C++

Il C venne sviluppato nel 1972 da Dennis Ritchie all'interno dei Laboratori Bell. Il suo progetto è inizialmente strettamente legato a quello del Sistema Operativo UNIX: buona parte di UNIX e delle sue utilities di sistema sono scritte in C. Il C nasce dunque principalmente come linguaggio per la *programmazione di sistema* e quindi ad ampio spettro di applicabilità. Requisiti fondamentali nella definizione del linguaggio sono l'efficienza e la flessibilità d'uso. Come conseguenza di queste scelte di fondo il C risulta essere un linguaggio piuttosto "permissivo".

Il C++ venne sviluppato da Bjarne Stroustrup sempre all'interno dei Laboratori Bell AT&T agli inizi degli anni '80. Il C++ nasce come estensione del C: include tutte le possibilità del C (è quindi possibile usare un compilatore C++ anche per sviluppare ed eseguire programmi in C), più molte altre. In particolare, il C++ introduce in più rispetto al C i concetti di *classe*, *oggetto* ed *ereditarietà* tra classi, che fanno del C++ un ottimo supporto per la programmazione orientata agli oggetti. Altre nuove possibilità presenti in C++, non strettamente correlate con la programmazione ad oggetti, sono l'*overloading di operatori e funzioni*, il passaggio parametri per riferimento, nuove forme di gestione dell'input/output, molte nuove librerie, nuove possibilità per la gestione delle stringhe, la gestione delle eccezioni, ... Tutte queste caratteristiche permettono una programmazione più semplice, più chiara e più affidabile.

Nella prima parte di queste dispense analizzeremo un sottoinsieme del C++ che contiene la maggior parte delle possibilità offerte dal C, con l'aggiunta di alcune possibilità proprie del C++ e non presenti, o meno eleganti, in C (come ad esempio il passaggio dei parametri per riferimento e l'input ed output tramite gli operatori ">>" e "<<"). Resteranno invece escluse dalla I Parte tutte quelle possibilità strettamente connesse con la programmazione orientata agli oggetti (come ad esempio le nozioni di classe e quella di ereditarietà), ma anche alcune nozioni non relative alla programmazione orientata agli oggetti (come ad esempio l'overloading di funzioni ed operatori ed i puntatori), per le quali preferiamo rimandare la trattazione ad un momento successivo, allo scopo di rendere la presentazione più semplice e graduale. Queste ulteriori possibilità del linguaggio, insieme ai principali costrutti per il supporto alla programmazione orientata agli oggetti, verranno introdotte nella II Parte.

Nella I come nella II Parte, comunque, utilizzeremo come linguaggio di riferimento il C++, senza ulteriori precisazioni.

Nota. Il linguaggio C, e di conseguenza anche il C++, è un linguaggio intenzionalmente molto "permissivo", che consente ad esempio di mescolare liberamente numeri interi con caratteri e booleani o addirittura con puntatori. Questo motivato, essenzialmente, dal desiderio di avere un linguaggio ad ampio spettro di applicabilità. Molto diverse sono le motivazioni che stanno alla base di un altro linguaggio di programmazione molto comune, "coetaneo" del C: il Pascal. Quest'ultimo, infatti, ha come suo obiettivo principale la scrittura di programmi "sicuri" e cerca di ottenere questo imponendo una programmazione disciplinata. Una conseguenza di tali diverse impostazioni di fondo tra i due linguaggi è, ad esempio, una filosofia profondamente diversa di trattamento e controllo dei tipi.

1.3.2 Un esempio di programma C++

In questo paragrafo mostriamo un primo semplice esempio di programma scritto in C++. Tutti i diversi costrutti del C++ qui introdotti in modo intuitivo verranno ripresi ed analizzati in modo più approfondito nei capitoli successivi.

Un programma C++—così come i programmi scritti nella maggior parte dei linguaggi di programmazione convenzionali—è costituito fondamentalmente da due tipi di costrutti linguistici: le *dichiarazioni* e gli *statement* (o comandi). Le prime servono a descrivere i dati utilizzati nel programma; gli statement invece costituiscono le istruzioni vere e proprie del linguaggio e permettono di specificare le azioni da compiere per giungere alla soluzione del problema considerato.

Si noti che nei diagrammi di flusso ci siamo basati su una descrizione dei dati informale, mentre nei linguaggi di programmazione questo aspetto è formalizzato in modo preciso e quindi anche la descrizione dei dati, oltre che quella delle azioni, dovrà rispettare precise regole sintattiche.

Il problema che vogliamo risolvere con questo primo programma C++ è un problema estremamente semplice, ma sufficiente a mettere già in evidenza alcune caratteristiche salienti di un tipico programma in un linguaggio di programmazione convenzionale.[6]

[6] Tutto il codice presentato in queste note è stato compilato ed eseguito con compilatore gcc 3.3.3 20040412 (Red Hat Linux 3.3.3-7) Copyright (C) 2003 Free Software Foundation, Inc.

Problema: dati in input 3 numeri interi, calcolare e dare in output la loro media.
Input: 3 numeri interi.
Output: un numero reale.

Programma:

```
#include <iostream>
using namespace std;
int main() {
  int x, y, z;
  cout << "Dammi 3 numeri interi" << endl;
  cin >> x >> y >> z;
  float m;
  m = (x + y + z) / 3.0;
  cout << "La media e' " << m << endl;
  return 0;
}
```

Osservazioni sul programma.

- `#include <iostream>` (direttiva *include*) indica che il programma ha bisogno di utilizzare ("includere") delle funzioni predefinite per la gestione dell'input/output le cui dichiarazioni sono contenute nel file `iostream` della libreria standard del C++.

- `using namespace std;` indica che il programma utilizza degli *identificatori*, non dichiarati nel programma stesso, che appartengono allo "spazio dei nomi" standard (`std`); in particolare, gli identificatori `cin` e `cout`, usati per identificare l'input e l'output del programma, nonché l'identificatore `endl`, usato per produrre un "a capo" in output, devono essere cercati nello spazio dei nomi standard.

- `int main()` indica che si tratta del *programma principale*; `int` e `main` sono due *parole chiave* e, come tali, fissate nel linguaggio; le parentesi tonde aperte e chiuse dopo `main()` indicano assenza di parametri dall'esterno.

- La *dichiarazione di variabile*

  ```
  int x, y, z;
  ```

  definisce tre nuove variabili, di nome `x`, `y` e `z`, tutte e tre di *tipo intero* (`int`). Il carattere "`;`" è usato in C++ come "terminatore" di dichiarazioni e statement.

  La dichiarazione di variabile

```
float m;
```

definisce una nuova variabile di nome `m` e *tipo reale* (`float`).

Si noti che qualsiasi variabile, prima di essere usata, deve essere dichiarata in una parte del programma che precede l'uso della variabile stessa.

- Lo statement

  ```
  cout << "Dammi 3 numeri interi" << endl;
  ```

  indica un'operazione di output: la stringa `"Dammi 3 numeri interi"` viene inviata sul dispositivo di output standard (normalmente il monitor), qui identificato dal nome `cout`, seguita dall'invio del carattere speciale di "a capo". "`<<`" è l'operatore di output (detto anche *operatore di inserimento*).

  L'esecuzione continua immediatamente.

  Lo statement

  ```
  cin >> x >> y >> z;
  ```

  indica un'operazione di input: si attende che tramite il dispositivo di input standard (solitamente la tastiera), qui identificato dal nome `cin`, vengano forniti al programma tre numeri interi, che verranno assegnati rispettivamente alle variabili `x`, `y` e `z`. "`>>`" è l'operatore di input (detto anche *operatore di estrazione*).

  L'esecuzione continua soltanto quando tutti i valori attesi (tre numeri interi in questo caso) sono stati forniti.

- Lo statement

  ```
  m = (x + y + z) / 3.0;
  ```

  indica un'operazione di assegnamento: l'espressione a destra dell'uguale, `(x + y + z) / 3.0`, viene valutata ed il suo risultato viene assegnato alla (ovvero diventa il nuovo valore della) variabile `m`.

  Se ad esempio `x` vale 4, `y` vale 6 e `z` vale 3, il valore di `m` diventa 4.33333.

- Lo statement

  ```
  return 0;
  ```

  indica che il programma termina restituendo all'*ambiente esterno* (cioè al sistema operativo, da cui il programma è stato inizialmente attivato) il valore 0 (il valore 0 viene tipicamente usato per indicare che il programma è terminato correttamente).

Nota. Alcuni compilatori C/C++ (in particolare Dev-C++ e Borland C++) prevedono che la finestra di output venga chiusa automaticamente al termine dell'esecuzione del programma. Per evitare che il programma termini senza dare la possibilità all'utente di prendere visione degli eventuali risultati visualizzati sulla finestra di output, è opportuno inserire al termine del programma un'istruzione che ne blocchi l'esecuzione. Questo si può ottenere, ad esempio, inserendo prima dell'istruzione `return 0`, le seguenti due istruzioni:

```
cout << "Digita un carattere qualsiasi per terminare ";
cin.get();
```

L'esecuzione dell'istruzione `cin.get()` pone il programma in attesa che l'utente digiti sulla tastiera un qualsiasi carattere (compresi il carattere "spazio" e "a capo"). Finché questo non avviene il programma non termina e quindi la finestra di output permane.

In alternativa alla `get()` si può utilizzare la funzione `getch()`, offerta da alcuni compilatori (tra cui i già citati Dev-C++ e Borland C++, ma non il `gcc` sotto Linux) come facente parte della libreria `conio.h`. Questa funzione si comporta essenzialmente come la `get()`, ma, a differenza di quest'ultima, evita di fare l'"echo" del carattere digitato sul monitor, risultando così, in questo specifico caso, più "naturale" per l'utente (in questo caso, infatti, non è necessario "vedere" quale carattere si è digitato, ma interessa soltanto averlo digitato). Volendo utilizzare questa alternativa il codice da inserire prima dell'istruzione `return 0`, diventa:

```
cout << "Digita un carattere qualsiasi per terminare ";
getch();
```

mentre, all'inizio del programma, andrà inserita la direttiva `#include <conio.h>`.

Per completezza, e per permettere un facile confronto, descriviamo l'algoritmo relativo al programma C++ appena mostrato anche con un diagramma di flusso (Figura 1.6). Per semplicità assumiamo qui di rappresentare le istruzioni di input ed output come blocchi rettangolari contenenti le corrispondenti istruzioni di input ed output del C++ (nell'uso comune queste istruzioni vengono spesso rappresentate da blocchi di forma speciale; noi preferiamo qui avvalerci dei blocchi rettangolari per non appesantire inutilmente la notazione utilizzata).

1.3.3 Convenzioni di programmazione

Nella stesura di un programma è opportuno, al fine di migliorarne la leggibilità, attenersi ad alcune convenzioni di scrittura.

- Evidenziazione delle "parole chiave".
  Le "parole chiave" sono elementi costitutivi del linguaggio di programmazione e come tali non modificabili. Ad esempio, `main`, `int`, `return` sono parole chiave del C++. Per aumentare la leggibilità di un programma si adotta solitamente la convenzione di evidenziare le parole chiave, ad esempio sottolineandole. Quando il programma viene scritto utilizzando un editor fornito da un ambiente di programmazione per uno specifico linguaggio solitamente l'editor riconosce le parole chiave e provvede automaticamente ad evidenziarle (ad esempio con

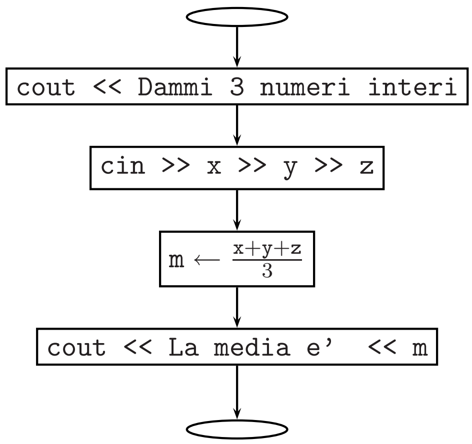


Figura 1.6: Diagramma di flusso per il calcolo della media di tre numeri interi.

il carattere "grassetto" oppure colorandole in modo diverso). Notare che l'evidenziazione delle parole chiave è utile all'utente umano, ma del tutto ignorata dal compilatore.

- "Indentatura".
  La separazione delle diverse parti di un programma su più righe ed il loro spostamento ed allineamento rispetto al margine sinistro (e cioè l'*"indentatura"*) sono essenziali per il programmatore, e per chiunque debba leggere il programma, per comprenderne la struttura, e quindi migliorarne la comprensibilità. Ad esempio è tipico in un programma C++ allineare sulla stessa colonna (e cioè alla stessa distanza dal margine sinistro) una parentesi graffa aperta e la corrispondente graffa chiusa, oppure spostare verso destra ed allineare sulla stessa colonna tutte le dichiarazioni e gli statement compresi tra due parentesi graffe. L'utilizzo di queste convenzioni è già visibile nel semplice programma mostrato in questo capitolo, ma diventerà via via più evidente man mano che verranno mostrati nuovi programmi d'esempio. Si noti comunque che, come per l'evidenziazione delle parole chiave, anche per l'"indentatura" si tratta di una convenzione utile all'utente umano ma del tutto ignorata dal compilatore (in linea di principio un programma potrebbe anche essere scritto tutto su un'unica riga).

- Commenti.
  I commenti sono frasi in formato libero che possono essere aggiunte in qualsiasi punto di un programma e che possono servire al programmatore per fornire informazioni aggiuntive sul significato e l'utilizzo

del programma o di parti di esso (ad esempio, l'utilizzo di una certa variabile). I commenti servono esclusivamente per rendere il programma più facile da capire sia per chi lo scrive che per chi deve leggerlo e comprenderlo, e sono del tutto ignorati dal compilatore.

In C++ sono possibili due modalità per aggiungere commenti ad un programma:

– `/* qualsiasi frase,`
  `anche su piu' righe */`

– `// qualsiasi frase` (fino a fine riga).

Ad esempio:

```
//programma di prova
#include <iostream>
using namespace std;
int main() { //inizio il programma
     ...
     cin >> x >> y >> z;
     /* A questo punto x, y, z contengono
     i tre numeri interi forniti tramite tastiera. */
     m = (x + y + z) / 3.0;
     ...
}
```

1.4 L'ambiente di programmazione

Per lo sviluppo e messa a punto di un programma in un certo linguaggio di programmazione si ha bisogno normalmente non soltanto di un compilatore per quel linguaggio, ma anche di un insieme di altri strumenti software che siano di ausilio alla scrittura e quindi all'esecuzione ed eventuale modifica di un programma. Questo insieme di strumenti—più o meno sofisticati, più o meno integrati fra loro, e più o meno correlati allo specifico linguaggio—è indicato come un *ambiente di programmazione* per un certo linguaggio. In Figura 1.7 è schematizzato un tipico semplice ambiente di programmazione per un linguaggio compilato, come ad esempio il C/C++.

Per poter essere eseguito un programma passa attraverso 5 fasi principali: *"editing"*, *compilazione*, *"linking"*, *caricamento* ed *esecuzione*.

La prima fase consiste nella scrittura in un linguaggio di programmazione $\mathcal{L}$ del programma (*programma sorgente*) e nel suo salvataggio in un file. Questa fase viene eseguita attraverso un programma chiamato *editor*, quale, ad esempio, `vi` o `emacs` in ambiente Unix/Linux, e `Blocco note` in ambiente Windows. Spesso l'editor è integrato con l'ambiente di programmazione e

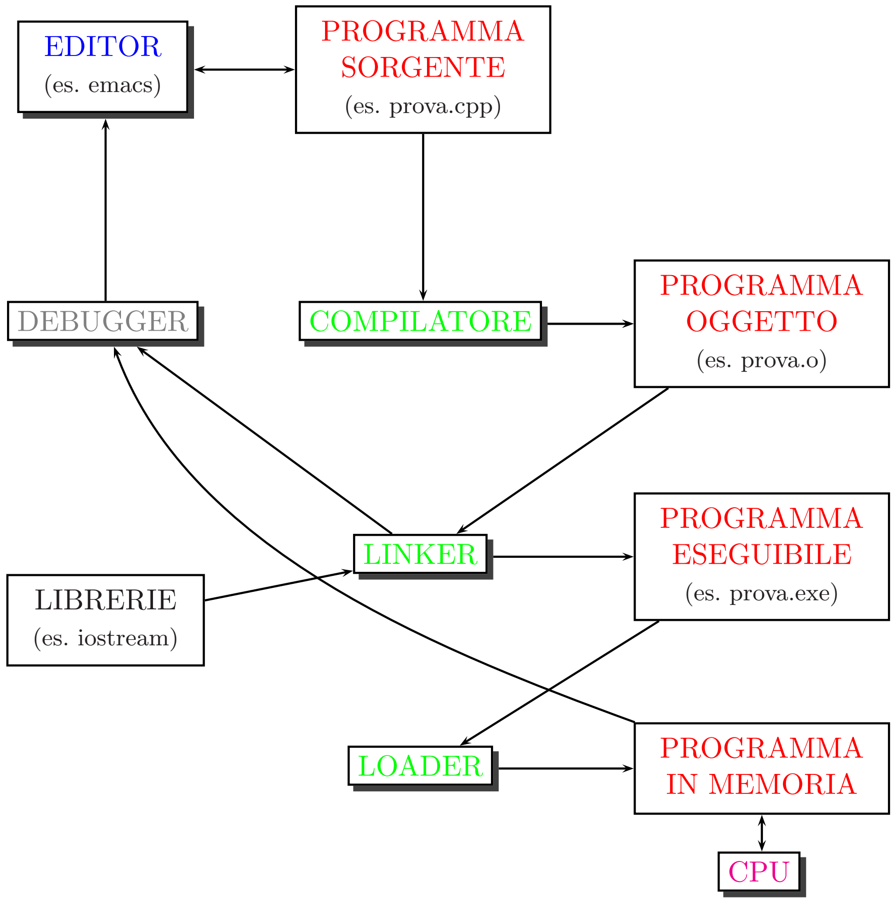


Figura 1.7: Schema di un ambiente di programmazione.

permette di riconoscere ed evidenziare alcuni elementi sintattici del linguaggio, come ad esempio le parole chiave, e di richiamare gli altri strumenti dell'ambiente in modo diretto.

Nella seconda fase, quella di *compilazione*, il programma in linguaggio ad alto livello viene trasformato nel corrispondente programma in linguaggio macchina (*programma oggetto*), tramite un programma traduttore, detto *compilatore*. La fase di compilazione è, in generale, molto complessa e si suddivide a sua volta in altre sotto-fasi che prevedono, tra l'altro, l'*analisi lessicale*, *sintattica* e *semantica* del codice da tradurre. Non ci soffermeremo su questi argomenti rimandando il lettore interessato ad approfondirli, ad esempio, in [1].

La terza fase è quella del *linking* (che in italiano si può tradurre con *collegamento*) e prevede l'utilizzo di uno strumento software detto *linker*.

Solitamente i programmi sorgenti contengono riferimenti a funzioni compilate separatamente, per esempio quelle fornite dalle *librerie standard*. Il linker provvede a "cucire" insieme il programma oggetto corrente, prodotto dal compilatore, con il codice oggetto delle altre funzioni esterne a cui il programma fa riferimento, andando anche a completare i riferimenti esterni presenti in quest'ultimo che erano stati necessariamente lasciati in sospeso durante la fase di compilazione. Il linker crea così un *programma eseguibile* completo (solitamente di dimensioni molto maggiori rispetto al codice oggetto del programma utente).

La quarta fase è chiamata *caricamento*. Un programma eseguibile, perché possa essere eseguito dalla macchina hardware sottostante, deve essere caricato nella memoria centrale. Questa operazione viene svolta da un programma di sistema detto *loader*. Il loader, una volta terminato il caricamento in memoria del programma utente, provvede anche ad attivarne l'esecuzione, facendo in modo che la CPU continui la sua esecuzione a partire dall'istruzione del programma indicata come punto iniziale di esecuzione (*entry point*).

Il comando *run* (o *execute*, o simili), tipicamente fornito dall'ambiente di programmazione, permette di eseguire in un colpo solo tutte le fasi che vanno dalla compilazione all'esecuzione del programma. L'ambiente fornisce tipicamente anche altri comandi che permettono di svolgere solo alcune delle fasi sopra indicate. Ad esempio, è possibile eseguire solo la compilazione di un programma sorgente (comando *compile*, o simili), ottenendo come risultato il corrispondente programma oggetto o la generazione di opportune segnalazioni nel caso siano presenti errori sintattici.

Approfondimento ("Preprocessing"). Il linguaggio C/C++ prevede anche una fase di trasformazione del programma sorgente, precedente a quella di compilazione, detta fase di *preprocessing*. Si tratta di una *trasformazione di programma*, effettuata da uno strumento detto *preprocessore*, che avviene all'interno dello stesso linguaggio: il programma in input è in linguaggio $\mathcal{L}$ e tale sarà anche il programma prodotto. Nel caso del C/C++, il preprocessor individua ed esegue comandi speciali, chiamati *direttive del preprocessore*, riconoscibili sintatticamente per il carattere iniziale "`#`", che indicano che sul programma devono essere eseguite alcune manipolazioni, come per esempio l'inclusione di codice sorgente contenuto in altri file (direttiva `#include`), o la sostituzione di stringhe con altre stringhe (direttiva `#define`).

I nomi dei file contenenti il codice del programma nelle sue varie fasi di traduzione terminano solitamente con opportune *estensioni* che indicano il tipo di informazione contenuta nel file. Ad esempio, nel caso del C++, i nomi dei file contenenti il codice sorgente terminano tipicamente con estensioni quali `.cc`, `.cpp`, `.c++` (ad esempio, `prova.cpp`), mentre i nomi dei file contenenti il codice oggetto terminano tipicamente con estensione `.o` oppure `.obj` (ad esempio, `prova.o`), e quelli contenenti codice eseguibile con l'estensione `.exe` (ad esempio, `prova.exe`).

In Figura 1.7 è evidenziata anche la presenza di un altro strumento tipico di un ambiente di programmazione, il *debugger*. Si tratta di un programma che è in grado di eseguire passo passo il programma utente caricato in memoria, comunicando con l'utente in modo interattivo, e mostrando l'evoluzione dei valori delle variabili presenti nel programma durante la sua esecuzione.

Capitolo 2

Elementi di base di un programma

2.1 Identificatori

Gli identificatori sono *nomi* utilizzati all'interno di un programma per riferirsi ai diversi componenti che lo costituiscono, come, ad esempio, variabili, funzioni, costanti, tipi, ecc.. Tecnicamente, tutti gli oggetti di un programma cui è possibile associare un identificatore sono detti *oggetti denotabili* del linguaggio.

Alcuni identificatori, come ad esempio `int`, `float`, `return`, in C++, sono *parole chiave* del linguaggio, e cioè nomi facenti parte della definizione del linguaggio stesso ed associati in modo prefissato ad oggetti del linguaggio come, ad esempio, tipi ed operazioni primitive . Come tali, questi identificatori, non potranno essere utilizzati per denotare oggetti definiti dall'utente (ad esempio, come nomi di variabili).

L'utente può introdurre i propri identificatori tramite opportune *dichiarazioni*. Una dichiarazione, oltre ad informare il compilatore che da quel punto in poi è possibile utilizzare quell'identificatore, serve anche a creare un'associazione (o *legame*, o *binding*) tra l'identificatore ed uno specifico oggetto denotabile (come ad esempio una variabile).

La forma degli identificatori definiti dall'utente è soggetta a precise regole sintattiche, che possono variare da linguaggio a linguaggio. La forma sintattica degli identificatori in C++ è definita nel modo seguente.

Identificatori C++: un *identificatore* è una sequenza di caratteri alfabetici e numerici, iniziante obbligatoriamente con un carattere alfabetico. Non sono ammessi caratteri speciali, come ad esempio "+", ">", "!", spazio, ..., ad eccezione del carattere "_" (underscore).

Esempio 2.1 (Identificatori)

`alfa1`, `ALFA`, `X1`, `y123`, `X_1`   *identificatori corretti*

`1A, X 1, X-1` *non sono identificatori.*

Alcuni linguaggi sono "case sensitive", cioè distinguono tra lettere maiuscole e minuscole, mentre altri non lo sono. Il C++ è "case sensitive": ad esempio, `i` e `I` sono due identificatori diversi in C++.

Nota. Conviene, in generale, scegliere nomi degli identificatori che ricordino il più possibile il significato che viene attribuito all'interno del programma all'oggetto da essi denotato. Ad esempio, se dobbiamo utilizzare una variabile per memorizzare il codice fiscale di una persona conviene chiamarla `codice_fiscale` piuttosto che semplicemente `x`. Questo contribuisce alla comprensibilità del codice scritto.

Nota. Meglio non iniziare un identificatore con carattere di sottolineatura. Questa possibilità è tipicamente utilizzata dal compilatore per identificatori che si riferiscono a collegamenti esterni. Inoltre, per riferirsi a variabili, costanti e funzioni si preferisce solitamente utilizzare identificatori inizianti con lettera minuscola; identificatori con lettera iniziale maiuscola vengono invece solitamente usati come nomi di tipo, mentre identificatori interamente in maiuscolo vengono tipicamente utilizzati per le direttive del preprocessore.

Ciascun identificatore può essere associato ad un solo oggetto all'interno di uno stesso "ambiente". Che cosa si intenda per "ambiente" e come si determinano le associazioni tra identificatori ed oggetti in ciascun punto di un programma verrà precisato in un paragrafo successivo, dedicato alle cosiddette *"regole di scope"*.

2.2 Variabili

Una variabile è un'astrazione di un "contenitore" (una cella di memoria, parlando di calcolatori) in grado di contenere, uno alla volta, dati di uno o più tipi prefissati. In particolare, una variabile in un linguaggio di programmazione convenzionale è caratterizzata da:

- il *tipo* dei dati che possono essere memorizzati nella variabile (per la definizione di tipo si veda il par. 2.3);
- il *valore* del dato contenuto nella variabile ad un certo istante;
- l'indirizzo della cella (o *locazione*) di memoria che costituisce la variabile.

Inoltre, ad una variabile è solitamente associato un *nome*, ovvero un identificatore, che permette di riferirsi in modo simbolico alla variabile stessa (operazione di *"referencing"*).[1]

Il valore di una variabile è modificabile, ad esempio tramite un'istruzione di assegnamento (vedi par. 2.4). Il nuovo valore sostituisce quello attuale, che quindi è definitivamente perso.

[1]Come vedremo meglio in seguito, non tutte le variabili hanno necessariamente un nome e, nel caso in cui ce l'abbiano, questo può non essere unico. Specificatamente, una variabile allocata dinamicamente (vedi par. **??**) non ha un nome, ma ad essa ci si può riferire direttamente tramite il suo indirizzo, memorizzato in un *puntatore*. Viceversa, una variabile passata come parametro per riferimento ad una funzione (vedi par. 5.3) avrà più di un nome associato ad essa.

Prima di poter essere utilizzata, una variabile deve essere creata. La creazione di una variabile può avvenire tramite una dichiarazione di variabile o tramite un'apposita istruzione di allocazione della memoria, come l'istruzione `new` in C++ (variabili dinamiche). Per ora ci limiteremo al caso di variabili create tramite dichiarazione, rimandando la creazione di variabili dinamiche ad un successivo capitolo.

La *creazione di una variabile* comporta, tra l'altro, l'allocazione dello spazio di memoria necessario ed eventualmente la sua inizializzazione. La quantità di memoria allocata per una variabile dipende dal tipo associato alla variabile stessa. Ad esempio, 4 byte per variabili di tipo intero, 1 byte per variabili di tipo carattere, ecc. L'operazione simmetrica alla creazione di una variabile è quella di *distruzione* e comporta la *deallocazione* della memoria principale ad essa riservata. Il tempo che intercorre tra la creazione di una variabile e la sua distruzione è detto *tempo di vita* della variabile. Ritorneremo più avanti anche sulle modalità di creazione e distruzione e sulla nozione di tempo di vita delle variabili.

Approfondimento (Variabili logiche). Le variabili utilizzate nei linguaggi di programmazione convenzionali sono profondamente diverse dalle *variabili matematiche* (o *logiche*). Il valore di una variabile dei linguaggi di programmazione è modificabile durante la computazione, mentre una variabile matematica (ad esempio l'argomento di una funzione) una volta assunto un valore, lo mantiene inalterato per tutta la computazione. Questo corrisponde al fatto che una variabile dei linguaggi di programmazione è in realtà un'astrazione di una cella di memoria, ovvero una "scatola" che può contenere un (solo) valore alla volta, modificabile. La presenza di variabili matematiche è una caratteristica fondamentale dei linguaggi di programmazione logica (ad esempio il Prolog) e dei linguaggi di programmazione funzionale (ad esempio il LISP). Per una trattazione più approfondita delle differenze tra variabili modificabili e variabili logiche si veda ad esempio [6].

2.2.1 Dichiarazione di variabile

Una dichiarazione di variabile è un costrutto linguistico che permette di specificare le caratteristiche essenziali di una variabile: nome, tipo ed un eventuale valore iniziale. La sua elaborazione comporta solitamente sia la creazione della variabile (ovvero dell'oggetto denotabile), con la conseguente allocazione dello spazio di memoria necessario, sia la creazione dell'associazione ("binding") tra questo ed il suo nome. Dalla dichiarazione in poi, la variabile può essere utilizzata, in lettura o in scrittura, tramite il suo nome. Nella maggior parte dei linguaggi di programmazione convenzionale (compreso il C++) utilizzare una variabile senza averla prima esplicitamente dichiarata viene considerato un errore, che viene normalmente individuato a tempo di compilazione.

Vediamo ora più in dettaglio le caratteristiche principali della dichiarazione di variabile in C++.

Dichiarazione di variabile in C++. La dichiarazione di variabile in C++ ha la seguente forma base:

$$t\ v_1 = i_1,\ v_2 = i_2,\ \ldots,\ v_n = i_n; \quad (n \geq 1)$$

dove:

- $v_1,\ldots,v_n$ sono identificatori (di variabili),
- t è un identificatore (di tipo),
- $i_1,\ldots,i_n$ sono espressioni di tipo t (facoltative).[2]

Il significato di questo costrutto è la dichiarazione di n variabili di nome $v_1,\ldots,v_n$, tutte di tipo t, con eventuali valori iniziali i valori risultanti dalla valutazione, rispettivamente, di $i_1,\ldots,i_n$.

Esempio 2.2 (Dichiarazioni di variabili)

```
int i;            // una variabile di tipo int e nome i
float x, y, z;    // tre variabili di tipo float
                  // e nome x, y e z
int i, j=1;       // due variabili di tipo int,
                  // di nome i e j, di cui la seconda
                  // con valore 1
```

Notare che

```
int i, j=1;
```

è del tutto equivalente a

```
int i;
int j=1;
```

Una variabile per la quale non si specifichi un valore iniziale nella sua dichiarazione (come la variabile `i` negli esempi di sopra) avrà comunque un valore al momento della sua creazione, che però è *indefinito* (precisamente, dipende dalla configurazione di bit presente in quel momento nella cella di memoria utilizzata per allocare la variabile). È quindi in generale un errore utilizzare il valore di una variabile prima di avergliene assegnato uno esplicitamente.

In molti linguaggi (ad esempio il Pascal) le dichiarazioni di variabile possono essere poste soltanto in punti precisi del programma, tipicamente all'inizio del programma principale o dei sottoprogrammi. In C++ invece una dichiarazione di variabile può essere posta in qualsiasi punto del programma dove possa stare uno statement. Ad esempio, nel programma introduttivo presentato nel paragrafo 1.3.2, la dichiarazione della variabile `m` è posta in mezzo al codice del programma principale, tra due statement qualsiasi:

[2]Ritorneremo in un capitolo successivo sulla nozione di espressione. Per ora assumiamo, in prima approssimazione, che un'espressione possa essere una costante, oppure una variabile, oppure un'espressione composta, costruita, ad esempio, tramite gli operatori aritmetici.

```
cin >> x >> y >> z;
float m;
m = (x + y + z) / 3.0;
```

Rimane comunque necessario che la dichiarazione preceda l'uso della variabile.

Si noti che in C++ l'espressione di inizializzazione di una variabile può essere un'espressione qualsiasi. Con riferimento all'esempio di sopra, si potrebbe allora scrivere:

```
float m = (x + y + z) / 3.0;
```

Una nozione importante associata alla dichiarazione di variabile è quella di *"campo d'azione"* (o *"scope"*) di una dichiarazione, definito come la porzione di programma in cui rimane attiva l'associazione nome–variabile stabilita dalla dichiarazione stessa. In presenza di regole di "scope" statiche, come nel caso del C++, lo "scope" di una dichiarazione di variabile è determinato dalla struttura (statica) del programma e solitamente, ma con varie, precise eccezioni, coincide con il tempo di vita della variabile stessa. Ritorneremo più approfonditamente sul concetto di "scope" di una variabile, con particolare riferimento al linguaggio C++, in un paragrafo successivo.

2.3 Tipi di dato primitivi

Tipo: il tipo di una variabile è costituito dall'insieme dei possibili *valori* assegnabili alla variabile e dall'insieme delle possibili *operazioni* applicabili ad essa.

In generale in un linguaggio di programmazione si distinguono tra *tipi semplici* e *tipi strutturati*, e tra *tipi primitivi* e *tipi definiti dall'utente*. In questo capitolo tratteremo dei tipi semplici primitivi, con particolare riferimento a quelli presenti in C++. In successivi capitoli tratteremo di tipi strutturati e di tipi definiti da utente.

La maggior parte dei linguaggi di programmazione convenzionali prevedono tipi semplici primitivi per i seguenti insiemi di valori: interi, reali, caratteri, e (non sempre) booleani. Il fatto che si tratti di tipi *semplici* indica che ciascun elemento del tipo è costituito da un valore singolo. Il fatto che siano *primitivi*, invece, indica che sono prefissati nel linguaggio.

Esaminiamo più in dettaglio i tipi semplici primitivi forniti dal C++.

2.3.1 Il tipo `int`

L'insieme dei valori del tipo `int` è costituito dall'insieme dei numeri interi con segno rappresentabili sulla specifica macchina hardware. Ad esempio, per interi rappresentati in complemento a 2 su 32 bit, l'insieme dei valori è l'intervallo chiuso $[-2^{31}, 2^{31} - 1]$.

Nota. Il minimo ed il massimo intero rappresentabili in una specifica installazione sono definiti da due costanti predefinite, `INT_MIN` e `INT_MAX`, che si trovano nell'header file `climits`.

I valori del tipo `int` sono denotati da *costanti intere* che possono avere la forma standard della rappresentazione decimale dei numeri (sequenza di cifre decimali con eventuale segno—ad esempio, `12`, `+34`, `-5`) oppure rappresentare un valore intero in base 8 o in base 16 (ad esempio, `012` per il numero 10 in base 8 e `0x1A` per il numero 26 in base 16).

L'insieme delle operazioni primitive su valori di tipo `int` prevede:

- operazioni aritmetiche di base: +, -, *, / (divisione intera con troncamento del risultato);
- resto della divisione intera: % — ad esempio, `5 / 2` restituisce come risultato 2, mentre `5 % 2` restituisce 1;
- operazioni di confronto: `==` (uguale), `!=` (diverso), `<`, `>`, `<=`, `>=`.

Le operazioni aritmetiche e quella di resto restituiscono un risultato di tipo `int`. Le operazioni di confronto invece restituiscono un risultato di tipo `bool`. Tutti gli operatori sopra indicati, così come la maggior parte di quelli sugli altri tipi di dato semplici primitivi che introdurremo fra breve, sono operatori *binari* (e cioè hanno due operandi), *infissi* (e cioè l'operatore è scritto in mezzo ai suoi operandi).

2.3.2 Il tipo `float`

L'insieme dei valori del tipo `float` è costituito dall'insieme (approssimato) dei numeri reali rappresentabili sulla specifica macchina hardware (tipicamente in virgola mobile, cioè "floating point").

Nota. Analogamente agli interi, nell'header file `cfloat` si trovano le due costanti predefinite, `FLT_MIN` e `FLT_MAX` che rappresentano il minimo ed il massimo numero in virgola mobile rappresentabile.

I valori del tipo `float` sono denotati da *costanti reali* che possono avere la forma standard della rappresentazione decimale dei numeri con virgola (ad esempio, `3.456`, `-23.00`, `.23`) oppure utilizzare la cosiddetta *notazione scientifica* con mantissa ed esponente (ad esempio, `-1.14e+3` ovvero `-1140.0`, `.12e-2` ovvero `0.0012`).

L'insieme delle operazioni primitive su valori di tipo `float` prevede:

- operazioni aritmetiche di base, +, -, *, /, che prendono due operandi di tipo `float` e restituiscono un risultato di tipo `float`;
- operazioni di confronto: come per il tipo `int`.

Accenniamo già ora al fatto che il C++ offre come tipi primitivi diversi sottotipi e supertipi dei tipi semplici primitivi qui presentati. In particolare, per quanto riguarda i numeri reali, viene spesso utilizzato, in alternativa al tipo `float`, il tipo `double`. L'unica differenza tra i due consiste, a livello astratto, nell'insieme di valori rappresentabili: l'insieme dei valori di `double` è infatti un soprainsieme dei valori di `float`. A livello concreto questo significa solitamente—ma non necessariamente—una quantità di memoria doppia per un numero `double` rispetto ad uno `float` (ad esempio, 32 bit per `float` e 64 bit per `double`). Cambia dunque il livello di precisione con cui si rappresentano i reali: *singola precisione* per i `float`, *doppia precisione* per i `double`. L'insieme delle operazioni applicabili nei due casi rimane invece esattamente lo stesso.

`float` può vedersi come un sottotipo di `double`. Questo implica tra l'altro che ogni oggetto di tipo `float` è anche un oggetto di tipo `double` e quindi un `float`, cioè un oggetto del sottotipo, può comparire ovunque possa trovarsi un `double`, cioè un oggetto del supertipo (*Principio dei sottotipi*). Il C++ permette anche di fare il viceversa e cioè di trattare un `double` come un `float`. In questo caso però è applicata una *conversione automatica* che porterà in generale ad una perdita di precisione.

Ritorneremo nella seconda parte del testo sul concetto di sottotipo, in connessione con la nozione di ereditarietà. Vedremo invece altri casi di sottotipo e supertipo per altri tipi semplici primitivi in un capitolo successivo.

Nota. L'utilizzo del nome `float` (invece del più significativo `real` utilizzato da altri linguaggi, quali il Pascal) è un'altra "concessione" fatta dai progettisti del C alla "visibilità" dell'implementazione sottostante. `float` si riferisce infatti alla tecnica "floating point" tipicamente adottata per rappresentare i numeri con virgola all'interno del calcolatore. Anche il termine `double` fa riferimento ad una tecnica di implementazione, quella della rappresentazione dei numeri reali in "doppia precisione".

Si noti che in C++ altre operazioni di uso comune sui numeri interi e reali, come ad esempio la radice quadrata, l'elevamento a potenza, le funzioni trigonometriche, ecc., non sono definite come operazioni primitive, ma piuttosto come funzioni della libreria standard. Per poterle utilizzare è necessario inserire nel programma la direttiva `#include <cmath>`. Tra le funzioni matematiche più comuni ricordiamo: `sin(x)`, `cos(x)`, `tan(x)` che calcolano rispettivamente, il seno, il coseno e la tangente trigonometrica di `x`; `log10(x)` e `sqrt(x)`, che calcolano rispettivamente, il logaritmo (in base 10) e la radice quadrata di `x`; `pow(x,y)` che calcola `x` elevato alla `y`-esima potenza.

2.3.3 Il tipo `char`

L'insieme dei valori del tipo `char` è costituito dall'insieme dei caratteri alfabetici, numerici e speciali rappresentabili tramite la codifica ASCII.

I valori del tipo `char` sono denotati da *costanti carattere* che in C++ hanno la forma '*c*' (racchiusi tra apici singoli), dove *c* può essere:

- un singolo carattere alfabetico (ad esempio, 'A', 'b'), numerico (ad esempio, '3'), o speciale (ad esempio, '+', ';', ' ' (carattere spazio), '!', ecc.);
- una coppia di due caratteri, della forma '*x*', per denotare i diversi *caratteri di controllo*: '\n' per il carattere di "new line" ("a capo"), '\b' per il carattere di "back space", '\t' per il carattere di "tabulazione", ecc..

L'insieme delle operazioni primitive su valori di tipo `char` prevede:

- operazioni aritmetiche di base: come per il tipo `int`;
- operazioni di confronto: come per il tipo `int`. L'ordinamento tra i caratteri è quello previsto dal codice ASCII. In particolare, l'ordinamento tra i caratteri alfabetici è quello standard: 'A' < 'B' < ... 'Z' < 'a' < ... < 'z'.

Mentre è naturale che le operazioni di confronto facciano parte della definizione del tipo `char`, risulta invece più "anomala" la presenza delle operazioni aritmetiche sui caratteri. Questa possibilità è, in realtà, un'immediata conseguenza della scelta del C++ (e prim'ancora del C) di rendere visibile a livello utente l'implementazione dei caratteri: i caratteri sono semplicemente interi senza segno, compresi tra 0 e 255. Dunque, in C/C++, i caratteri possono a tutti gli effetti essere trattati come interi "piccoli" e, viceversa, interi "piccoli" possono vedersi come caratteri.

Ad esempio, le seguenti sono istruzioni corrette in C++:

```
char c1 = 'A'; // c1 contiene 'A' (codice ASCII di 'A' = 65)
char c2 = 97;  // c2 contiene 'a' (codice ASCII di 'a' = 97)
c1 = c1 + 1;   // ora c1 contiene 'B'
c2 = 'a' - 'A' // ora c2 contiene  97 - 65 = 32
               // (codice ASCII del carattere "spazio")
```

Si tratta di una possibilità poco "pulita" dal punto di vista formale, in quanto permette di vedere lo stesso dato con due tipi diversi. In pratica però può risultare utile in varie situazioni. Ad esempio, per passare da un carattere minuscolo `c` al carattere maiuscolo corrispondente basta sottrarre 32 a `c` (32 è infatti la differenza tra i codici ASCII dei caratteri minuscoli e dei corrispondenti caratteri maiuscoli):

```
char c = 'a';    // c contiene il carattere 'a'
cout << c - 32;  // stampa il carattere 'A'
```

Approfondimento (Caratteri e interi). La possibilità di mescolare liberamente caratteri e interi non è presente, ad esempio, in Pascal. In Pascal sono invece presenti altre funzioni primitive per operare sui caratteri che permettono di sfruttare l'ordinamento dei caratteri (`succ` e `pred`) e di passare da un carattere al numero d'ordine corrispondente e viceversa (`ord` e `chr`).

2.3.4 Il tipo `bool`

L'insieme dei valori del tipo `bool` è costituito dall'insieme dei valori logici "vero" e "falso", denotati rispettivamente dalle costanti booleane `true` e `false`. Il seguente è un esempio di dichiarazione di variabile booleana:

```
bool b1 = false;
```

L'insieme delle operazioni primitive su valori di tipo `bool` prevede:

- operazioni logiche di base: `&&` (AND logico), `||` (OR logico), `!` (NOT logico);
- operazioni aritmetiche e di confronto: come per il tipo `int`.

Le operazioni logiche prendono operandi di tipo `bool` e restituiscono un risultato di tipo `bool`. Gli operatori logici primitivi del C++ sono riassunti nella Tabella 2.1.

OPERATORE	FUNZIONE	USO
`!`	NOT logico	`!E`
`&&`	AND logico	`E1 && E2`
`||`	OR logico	`E1 || E2`

Tabella 2.1: Operatori logici.

L'operatore `&&` restituisce `true` se entrambi gli operandi sono `true`, mentre l'operatore `||` restituisce `true` se almeno uno dei suoi operandi è `true`. L'operatore `!` restituisce `true` se il suo operando ha un valore `false`, altrimenti restituisce `false`. Il significato dei diversi operatori logici è descritto sinteticamente nella Tabella 2.2, chiamata *tavola di verità*. b_1 e b_2 sono espressioni booleane (cioè espressioni il cui valore è di tipo booleano), mentre T ed F indicano rispettivamente i valori logici "true" e "false".

Tramite le tavole di verità è possibile ricavare il significato di espressioni booleane qualsiasi a partire da quello delle espressioni componenti. Ad esempio la tavola di verità per l'espressione a `|| !` b è la seguente:

a	b	a `|| !` b
T	T	T
T	F	T
F	T	F
F	F	T

b_1	b_2	`!`b_1	b_1 `&&` b_2	b_1 `||` b_2
T	T	F	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	F

Tabella 2.2: Tavole di verità degli operatori logici NOT, AND, OR.

Per quanto riguarda le operazioni aritmetiche e di confronto, anche per il tipo `bool`, così come per tipo `char`, il C++ fa la scelta di permettere che l'implementazione del tipo sia visibile a livello utente. Precisamente, il valore booleano `false` corrisponde al valore intero 0, mentre il valore booleano `true` corrisponde a qualsiasi valore intero diverso da 0. Dunque, in C++, i booleani possono a tutti gli effetti essere trattati come numeri interi e, viceversa, numeri interi possono vedersi come booleani.

Questo spiega anche perchè tra le operazioni primitive del tipo `bool` compaiano le operazioni aritmetiche. Pertanto risulta corretto (anche se poco "pulito") scrivere ad esempio:

```
char b1 = false;  // b1 contiene false
b1 = b1 + 1;      // ora b1 contiene true
```

2.4 Statement di assegnamento

Introduciamo in questo paragrafo uno statement fondamentale per tutti i linguaggi di programmazione convenzionali, lo statement di assegnamento. Come al solito faremo riferimento al linguaggio C++, ma le considerazioni che verranno fatte sono del tutto generali. Gli altri statement per realizzare strutture di controllo diverse (precisamente, gli statement di selezione, di iterazione e di salto) saranno esaminati nel capitolo successivo.

Assegnamento. Lo statement di assegnamento ha la seguente forma generale:

l-expr `=` *r-expr* `;`

dove:

- *l-expr* ("left expression") è un'espressione che denota una locazione di memoria (in particolare, *l-expr* può essere un identificatore di variabile)
- *r-expr* ("right expression") è un'espressione qualsiasi (vedremo una definizione precisa di espressione nel paragrafo successivo)
- `=` è l'operatore di assegnamento (che può differire da linguaggio a linguaggio: ad esempio, `=` in C++, Java e Fortran, `:=` in Pascal ed Ada, ecc.).

La semantica informale di questa istruzione è: valuta l'espressione *r-expr* e quindi assegna il valore risultante da questa valutazione alla locazione di memoria individuata dalla valutazione di *l-expr*.

Esempio 2.3 (Statement di assegnamento in C++)

```
int x, y, z;
x = 1;                  //assegna 1 a x
y = x + 3;              //assegna 4 a y
z = (x * y) - (2 / x); //assegna 2 a z
```

Si noti che la valutazione dell'espressione che compare nella parte destra dell'assegnamento restituisce come suo risultato un *valore*, mentre la valutazione dell'espressione a sinistra restituisce un *riferimento* (o, più concretamente, un indirizzo) ad una locazione di memoria. Dunque, in un semplice assegnamento come `x = y`, la variabile a destra denota un valore, mentre la variabile a sinistra denota una locazione di memoria: il valore di `y` viene copiato nella locazione di memoria individuata da `x`.

Per questo motivo, nella parte sinistra dell'assegnamento non può comparire un'espressione qualsiasi, ma soltanto espressioni che possano denotare locazioni di memoria. In particolare, la *l-expr* può essere un identificatore di variabile, come negli esempi mostrati sopra. Ma una *l-expr* può essere anche un'espressione più complessa, come ad esempio, `A[i]`, o `A.c`, o `*A`, dove `A` è a sua volta una *l-expr*. Non può invece essere, ad esempio, un'espressione aritmetica come `x + 1`, con `x` variabile intera, che chiaramente restituisce come suo risultato un valore intero. Esamineremo più avanti il significato delle diverse espressioni che possono comparire alla sinistra di un assegnamento, e ritorneremo su questo argomento più in generale anche nella II Parte di queste note, nel paragrafo dedicato alla "ridefinizione" dell'operatore di assegnamento.

E' importante osservare che lo statement di assegnamento non è un'uguaglianza (nonostante l'infelice scelta da parte dei progettisti del C di usare il simbolo = per rappresentare l'operatore di assegnamento). L'assegnamento modifica una locazione di memoria, quella denotata dalla *l-expr*, mentre un'uguaglianza semplicemente confronta due valori, senza apportare alcuna modifica.

Nell'assegnamento, inoltre, è importante l'ordine di valutazione delle sue due parti: prima si valuta la parte destra e poi si usa il risultato di questa valutazione per modificare la locazione denotata dalla parte sinistra. Questo è evidente, ad esempio, nel tipico statement di assegnamento,

```
x = x + 1;
```

il cui effetto è quello di incrementare di 1 il valore della variabile `x`.

Un semplice esempio che ben evidenzia le caratteristiche dello statement di assegnamento e delle variabili dei linguaggi di programmazione è quello dello scambio dei valori di due variabili. Supponiamo di avere due variabili intere `x` e `y` e di voler scambiare i loro valori, in modo cioè che il valore contenuto in `x` vada in `y` e viceversa. Non è difficile rendersi conto che la sequenza di istruzioni

```
x = y;
y = x;
```

non risolve correttamente il nostro problema. L'effetto di queste due istruzioni è piuttosto quello di assegnare sia ad `x` che ad `y` lo stesso valore iniziale di `y`. E a poco vale invertire l'ordine delle due istruzioni (l'effetto diventa quello di assegnare il valore di `x` ad entrambe le variabile). Una soluzione corretta a questo problema si ottiene utilizzando una variabile aggiuntiva (o ausiliaria), che chiameremo **`aux`**, nel modo seguente:

```
aux = x;
x = y;
y = aux;
```

Ritorneremo ancora sullo statement di assegnamento dopo aver visto alcune precisazioni sulla nozione di espressione nel paragrafo successivo.

2.5 Espressioni ed operatori

Una nozione fondamentale presente in qualsiasi linguaggio di programmazione è quella delle espressioni. La definizione sintattica delle espressioni è piuttosto articolata e può variare da linguaggio a linguaggio. Esiste però una definizione comune alla maggior parte dei linguaggi di programmazione convenzionali (compreso il C++) secondo la quale un'espressione può avere una delle seguenti forme:

- una costante;
- una variabile;
- un'espressione composta della forma
  $$expr_1 \; op \; expr_2$$
  dove $expr_1$, $expr_2$ sono (ricorsivamente) espressioni e *op* è un operatore (binario) infisso (ad esempio, un operatore aritmetico, o di confronto, o logico); $expr_1$ ed $expr_2$ sono detti gli *argomenti* (o *operandi*) dell'espressione, a cui viene *applicato* l'operatore *op*;
- un'espressione composta della forma
  $$op \; expr,$$
  dove *op* è un operatore (unario) prefisso (ad esempio l'operatore booleano *not*);
- un'espressione *expr* tra parentesi
  $$(expr);$$
- un'espressione *expr* preceduta da segno
  $$+expr \text{ oppure } -expr.$$

- una chiamata di funzione

$$\textit{nome\_funzione}\texttt{(}expr_1\texttt{,}\ldots\texttt{,}expr_n\texttt{)}.$$

Vedremo alcune altre forme di espressioni nel capitoli successivi, nel caso specifico del linguaggio C++.

Approfondimento (Posizione degli operatori). I termini prefisso, infisso e postfisso si riferiscono alla posizione dell'operatore rispetto agli operandi: *prefisso* quando l'operatore precede gli operandi, *infisso* quando l'operatore appare in mezzo agli operandi, e *postfisso* quando l'operatore segue gli operandi.

In C++ la notazione a livello utente per gli operatori primitivi del linguaggio è la seguente: infissa per gli operatori binari aritmetici, logici, di confronto (e cioè quella adottata normalmente in matematica) e per alcuni altri operatori particolari, come ad esempio l'operatore `.` (selettore di campi); prefissa per la maggior parte degli operatori unari, come, ad esempio, gli operatori `!` (not), `new`, `sizeof`, `*` (dereferenziazione), `++` e `--` (pre incremento e decremento); postfissa per alcuni altri operatori unari, come, ad esempio, `++` e `--` (post incremento e decremento).

Per la chiamata di funzioni definite da utente (si veda il Capitolo 5) si utilizza normalmente una notazione prefissa. Come vedremo meglio nella II Parte di queste note, il C++ ammette anche che, per le funzioni definite da utente che abbiano lo stesso nome di un operatore primitivo, si possa usare a livello utente la stessa notazione (in particolare quella infissa) che caratterizza l'operatore primitivo

Infine, notiamo che in C++ esistono alcuni operatori particolari che utilizzano una notazione, a livello utente, diversa dalle tre menzionate sopra. Ad esempio l'operatore binario `[]`, che permette l'accesso all'`i`-esimo elemento di un array `A`, prevede che il primo dei suoi argomenti (il nome dell'array) preceda l'operatore, mentre il secondo argomento (l'indice `i`) viene scritto tra le due quadre (e cioè, `A[i]`). Un altro caso particolare è costituito dall'operatore `?:`, usato per scrivere cosiddette *espressioni condizionali*, che è un operatore ternario, cioè prevede tre argomenti (il suo utilizzo è spiegato successivamente nel paragrafo 2.5.4).

2.5.1 Valutazione di una espressione

Valutare un'espressione significa determinarne il valore applicando gli operatori ai loro operandi, in accordo con la semantica prevista per i diversi operatori e secondo un opportuno ordine di valutazione tra essi.

La semantica degli operatori in C++ è nella maggior parte dei casi quella intuitiva. Per esempio, l'operatore `+` sugli interi e sui reali è interpretato come la somma, l'operatore `&&` come l'AND logico, l'operatore `==` come l'uguaglianza, ecc..

In un'espressione che preveda più operatori, l'ordine di valutazione delle diverse sottoespressioni è determinato in base a precise regole di precedenza ed associatività.

Precedenza tra operatori. I circa 60 operatori in C++ sono raggruppati in 18 classi di precedenza. Gli operatori con precedenza più alta vengono valutati per primi. La precedenza degli operatori più usati è mostrata in Tabella 2.3. Per gli operatori aritmetici si noti che le precedenze corrispondono a quelle comunemente usate in matematica (ad esempio, `*` e `/` hanno maggior precedenza rispetto a `+` e `-`).

Precedenza più alta	! ++ −−
	∗ / %
	+ −
	$<$ $>$ $<=$ $>=$
	&&
	$\vert\vert$
Precedenza più bassa	=

Tabella 2.3: Precedenza degli operatori in C++ (parziale).

Esempio 2.4 *Consideriamo l'espressione*

```
9 + 5 * 2.
```

Poichè il `*` *ha precedenza sul* `+`*, prima viene valutato* `5 * 2` *e poi* `9 + 10`*.*

Esempio 2.5 *Consideriamo l'espressione*

```
x > 1 && x <= 2.
```

Poichè il `>` *e il* `<=` *hanno precedenza sul* `&&`*, prima vengono valutate le espressioni* `x > 1` *e* `x <= 2`*, e poi successivamente si applica l'operatore* `&&`*.*

L'uso delle parentesi tonde permette comunque di controllare esplicitamente l'ordine di valutazione (come avviene comunemente in matematica): le espressioni tra parentesi assumono la massima precedenza e quindi vengono valutate per prime.

Esempio 2.6 *La valutazione dell'espressione*

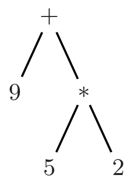
```
(9 + 5) * 2
```

comporta prima la valutazione di `9 + 5` *e poi di* `14 * 2`*.*

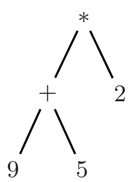
In generale, dunque, la presenza di opportune regole di precedenza ci permette di ridurre l'uso esplicito delle parentesi, semplificando la scrittura dei programmi. Ad esempio, nell'espressione dell'Esempio 2.5 abbiamo potuto evitare le parentesi attorno alle due sottoespressioni di confronto, grazie alle regole di precedenza dei tre operatori coinvolti. Nel dubbio, però, è consigliabile utilizzare le parentesi tonde per rendere esplicito l'ordine di valutazione.

Approfondimento (Alberi sintattici). Le espressioni possono essere convenientemente rappresentate sotto forma di alberi, detti *alberi sintattici*. Senza addentrarci in una definizione precisa di questa nozione, ne diamo qui un'idea intuitiva tramite alcuni esempi (per un trattamento preciso e completo dell'argomento si rimanda, ad esempio, a [1]).

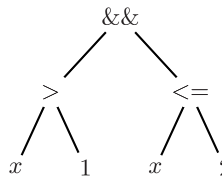
L'espressione dell'Esempio 2.4 può essere rappresentata con l'albero sintattico:



dove i nodi terminali (le *foglie*) sono gli operandi (costanti o variabili) mentre i nodi interni sono gli operatori. Se si assume di valutare per primi i sottoalberi più in profondità, questa rappresentazione permette facilmente di "codificare" anche l'ordine di valutazione previsto dalle regole di precedenza (e associatività) delle espressioni.



Così, ad esempio, l'espressione dell'Esempio 2.6 viene rappresentata dall'albero sintattico:



La rappresentazione tramite alberi sintattici permette dunque di rendere esplicita l'ordine di valutazione di una espressione, una volta stabilita una regola di visita dell'albero stesso. Gli alberi sintattici sono la forma interna spesso utilizzata dal compilatore per rappresentare le espressioni che appaiono nel programma sorgente. Questa rappresentazione viene usata all'interno del compilatore per guidare la generazione del codice macchina che realizza la valutazione dell'espressione e per individuare possibili ottimizzazioni del codice stesso.

Associatività. Le regole di associatività specificano l'ordine di valutazione tra operatori con la stessa precedenza. La maggior parte degli operatori in C++ (ma anche negli altri linguaggi di programmazione) sono *associativi a sinistra*, cioè richiedono che, tra operatori con la stessa precedenza, si applichi per primo l'operatore più a sinistra. Esistono però anche alcuni operatori, tra cui quello di assegnamento e gli operatori unari, che sono associativi a destra.

Esempio 2.7 (Associatività)

`9 + 2 - 5` *come se fosse* `(9 + 2) - 5`
`6 / 2 * 0` *come se fosse* `(6 / 2) * 0`
`x = y = z = 1` *come se fosse* `x = (y = (z = 1))`

Il significato della concatenazione di più operatori di assegnamento, mostrata nell'ultimo esempio di sopra, verrà chiarito nel Paragrafo 2.6*.*

Si noti che per gli operatori binari infissi della forma $expr_1$ *op* $expr_2$, la definizione del linguaggio C++ non specifica in modo esplicito quale delle due sottoespressioni, $expr_1$ o $expr_2$, venga valutata per prima.[3] Questa decisione è lasciata alla realizzazione dello specifico compilatore e potrebbe quindi portare a comportamenti diversi in diversi compilatori. Nella maggior parte dei casi, comunque, i compilatori scelgono di procedere sempre da sinistra verso destra, e cioè di valutare per prima la sottoespressione $expr_1$ e poi la sottoespressione $expr_2$.

2.5.2 Tipo di un'espressione

Il tipo del risultato prodotto dalla valutazione di un'espressione è detto *tipo dell'espressione*. Di fatto, il tipo di un'espressione è il tipo dell'operatore principale dell'espressione (quello valutato per ultimo, ovvero quello posto sul nodo radice dell'albero sintattico dell'espressione). Ad esempio, il tipo dell'espressione `x > 1 && x <= 2` è il tipo dell'operatore `&&` e quindi `bool`, mentre il tipo dell'espressione `sizeof(double) % 2` sarà il tipo dell'operatore `%` e quindi `int`.

Per operatori sovraccarichi, come per esempio "+", "∗", "−", il tipo dell'espressione dipende dal tipo degli operandi. Ad esempio:

```
float x = 2.0, z;  // due variabili di tipo float
z = x * 1.5;
```

Poichè gli operandi di `*` sono entrambi di tipo `float` si utilizzerà la moltiplicazione tra `float` che ha come risultato un `float`. Dunque il tipo dell'espressione alla destra dell'assegnamento (nella seconda istruzione) è `float`.

Approfondimento (Tipi non omogenei). Nel caso in cui i tipi degli operandi di un operatore primitivo sovraccarico non siano omogenei, il C++ risolve l'ambiguità applicando delle semplici *regole di conversione automatica*. Tali regole prevedono che il tipo di livello inferiore sia "promosso" (temporaneamente) a quello di livello superiore, secondo una gerarchia che prevede, per i tipi semplici primitivi più comuni, le seguenti relazioni:

$$\texttt{int} < \texttt{float} < \texttt{double}$$

mentre `char` e `bool` sono considerati allo stesso livello di `int`. Ad esempio, se il frammento di programma mostrato sopra è modificato come segue:

```
int x = 2;          // una variabile di tipo int
float z;            // una variabile di tipo float
z = x * 1.5;
```

in questo caso il tipo della variabile `x` è "promosso" a `float` e quindi il tipo dell'espressione è `float`.[4]

---

[3] A questo fa eccezione la valutazione delle espressioni booleane, per le quali, come vedremo nel prossimo paragrafo, è previsto esplicitamente di valutare le sottoespressioni da sinistra verso destra.

[4] Si noti che nel primo esempio di programma C++ mostrato nel paragrafo 1.3.2 abbiamo usato la costante reale `3.0` (invece che la costante intera `3`) nell'espressione `(x + y + z) / 3.0` proprio per "forzare" l'espressione ad essere di tipo `float`; in questo modo,

Per le funzioni definite dall'utente il tipo del risultato della funzione è quello specificato nella dichiarazione della funzione stessa.

Infine, osserviamo che espressioni che coinvolgono tipi non compatibili con quelli della loro definizione (e cioè non appartenenti al dominio della funzione realizzata dall'espressione) vengono individuate a tempo di compilazione (*compile time*) e segnalate come errate.

2.5.3 Espressioni booleane

Un tipo particolarmente interessante di espressioni sono le cosiddette espressioni booleane, cioè espressioni che restituiscono un risultato di tipo `bool`. In particolare, possiamo distinguere tra:

- espressioni booleane semplici, come per esempio

  ```
  x > y + 1
  ```

- espressioni booleane composte, in cui due o più espressioni booleane semplici sono messe insieme tramite connettivi logici, come per esempio

  ```
  (x > 3 && y < 15) || x < 0
  ```

  oppure

  ```
  x >= 0  &&  x < 10
  ```

  (ovvero, usando la notazione matematica standard, $0 \leq x < 10$).

Alle espressioni booleane composte si applicano le normali regole dell'algebra booleana. Ad esempio, l'espressione

```
(x == 1 && y < 1.5) || (x == 1 && z > 1)
```

può essere scritta equivalentemente come

```
x == 1 && (y < 1.5 || z > 1).
```

Abbiamo cioè applicato la *proprietà distributiva*:

$$p \text{ and } (q \text{ or } r) \equiv (p \text{ and } q) \text{ or } (p \text{ and } r).$$

infatti, l'espressione `(x + y + z)` che è di tipo `int` viene "promossa" a `float` e si applica l'operatore `/` per `float` (applicando `/` per `int` si sarebbe avuto un troncamento del risultato della divisione).

dove p, q ed r rappresentano generiche espressioni booleane.

Nella valutazione delle espressioni booleane in C++ si applica una forma di *valutazione "lazy"* (detta anche con *"corto circuito"*), nel senso che la valutazione dell'espressione (da sinistra a destra) termina non appena si è in grado di determinarne il suo valore logico, *true* o *false*

Nota. Questo non è vero in tutti i linguaggi di programmazione. Ad esempio, in Pascal, la definizione del linguaggio non permette di fare questa assunzione, perciò bisogna sempre prevedere che tutti gli operandi dell'espressione composta possano essere valutati.

Ad esempio, nella valutazione dell'espressione

```
-3 > 0 && x < 3
```

è sufficiente prendere in considerazione soltanto la prima condizione, `-3 > 0`, per determinare che il risultato dell'intera espressione è *false*. Analogamente, nella valutazione dell'espressione

```
3 > 0 || x < 3
```

viene considerata soltanto la prima condizione ed il risultato restituito sarà *true*.

Nei capitoli successivi, vedremo con alcuni esempi come si può trarre vantaggio (oltre all'evidente vantaggio in termini di efficienza) da questa caratteristica del C++.

2.5.4 Espressioni condizionali

Il C++ mette a disposizione una forma particolare di espressione, detta *espressione condizionale*, per la cui valutazione viene adottata una strategia di valutazione *lazy* che permette di valutare una soltanto tra due sottoespressioni in base al valore logico di una terza sottoespressione booleana. Precisamente, un'espressione condizionale ha la seguente forma generale:

$$cond \ \texttt{?} \ expr_1 \ \texttt{:} \ expr_2$$

dove

cond :	espressione booleana;
$expr_1$, $expr_2$:	espressioni qualsiasi;
`?:` :	operatore ternario (l'unico in C++).

La semantica di questa espressione è: se *Cond* ha valore "vero" il risultato dell'espressione condizionale è il risultato della valutazione di $expr_1$; altrimenti è il risultato della valutazione di $expr_2$.

Il seguente esempio, che calcola il valore assoluto di un numero immesso da utente tramite standard input, mostra l'utilizzo di un'espressione condizionale:

```
#include <iostream>
using namespace std;
int main() {
  cout << "Inserisci un numero" << endl;
```

```
  int x;
  cin >> x;
  int abs_x = (x >= 0) ? x : -x;
  cout << "Il valore assoluto di " << x << " e' "
       << abs_x << "." << endl;
  return 0;
}
```

Si osservi che un'espressione condizionale, proprio per il fatto di essere un'espressione, può stare per esempio a destra di un assegnamento.

Nota. La sintassi particolarmente sintetica suggerisce l'utilizzo di questa forma di espressione in casi in cui le espressioni $expr_1$ e $expr_2$ risultino relativamente semplici, per evitare possibili problemi di leggibilità e comprensione del testo.

2.6 Ancora sullo statement di assegnamento

Lo statement di assegnamento agisce, di base, tramite un *effetto collaterale* (o *side effect*): la sua esecuzione modifica in modo permanente il contenuto di una locazione di memoria, precisamente il valore della variabile che appare alla sua destra[5]. In C++ (ma non in altri linguaggi, come ad esempio il Pascal) l'assegnamento non si limita a produrre un effetto collaterale, ma può essere usato anche come un'espressione che restituisce un risultato esplicito. Precisamente, lo statement di assegnamento *var* = *expr* restituisce come suo risultato il valore dell'espressione *expr* assegnato alla variabile *var*.

Questa possibilità può avere alcune interessanti applicazioni (oltre che alcuni "rischi", come vedremo in alcuni esempi nei capitoli successivi). Ad esempio, in C++ è possibile scrivere lo statement

```
x = y = 1;
```

in cui l'assegnamento più a destra è usato come espressione all'interno dell'assegnamento più a sinistra. Ricordando che l'operatore `=` è associativo a destra, la valutazione di questo statement procede nel modo seguente: valuta l'espressione `y = 1`, assegnando 1 a `y` (effetto collaterale) e restituendo come risultato il valore 1; esegue l'assegnamento più esterno, assegnando a `x` il valore 1 restituito al passo precedente (mentre il risultato esplicito dell'assegnamento, in questo caso, può essere ignorato). Il risultato finale è di aver assegnato il valore 1 sia ad `x` che ad `y` con un'unico statement.

Un'ultima osservazione sull'operatore di assegnamento riguarda i tipi dei suoi argomenti. Dato un assegnamento *var* = *expr*, il tipo di *var* e il tipo di *expr* devono essere *compatibili*. Questo è senz'altro vero se il tipo di *var* e quello di *expr* sono coincidenti. Ad esempio, entrambi `int` o entrambi `float`. Nel caso dei tipi semplici primitivi, però, è possibile anche che i tipi delle

[5] La presenza di meccanismi che operano per effetti collaterali è un aspetto caratterizzante di tutti i linguaggi di programmazione cosiddetti imperativi. Per un approfondimento sull'argomento si veda ad esempio [6].

due parti dell'assegnamento non siano coincidenti: in questo caso si applica una regola di *conversione automatica* che prevede che il tipo dell'espressione a destra dell'"`=`" sia convertito in quello della variabile a sinistra dell'"`=`". Ad esempio, il codice seguente è corretto:

```
int i;
float f;
f = 10;  // int --> float
i = 1.7;  // float --> int
```

Nel secondo assegnamento, però, la conversione dal tipo `double` della costante `1.7` al tipo `int` della variabile `i` comporta un troncamento del valore assegnato e quindi in generale una perdita di informazione solitamente segnalata dai compilatori con un *warning* (in questo esempio, il valore assegnato ad `i` sarà 1).

2.6.1 Altri operatori di assegnamento

Il C++ offre altri operatori di assegnamento che permettono di scrivere in modo più sintetico alcune forme di assegnamento di uso comune.

- Operatori della forma

  op`=`

  dove *op* può essere uno dei seguenti operatori: `+`, `-`, `*`, `%`, `/`, `<<`, `>>`, `&`, `|`, `^`.
  Il significato dello statement:

  l-expr op`=` *expr*`;`

  dove *expr* è un'espressione compatibile con il tipo di *op*, è:

  l-expr `=` *l-expr op expr*`;`

  Ad esempio:

  ```
  x += 2;        //equivale a  x = x + 2
  x *= 10;       //equivale a  x = x * 10
  ```

  Come per la forma base dello statement di assegnamento, anche le forme contratte di assegnamento possono vedersi come espressioni, il cui risultato è il valore assegnato alla *l-expr*. È pertanto possibile scrivere, ad esempio:

  ```
  int x = 1, y;
  y = x += 5;
  ```

  in cui `x += 5` è un'espressione la cui valutazione modifica (come effetto collaterale) il valore di `x`, incrementandolo di `5`, e quindi restituisce come risultato il nuovo valore di `x` (e cioè `6`), che viene assegnato a `y`.

- Operatori di autoincremento e autodecremento:

  `++` e `--`.

  Questi due operatori possono essere usati sia prefissi che postfissi. Il significato degli statement:

  l-expr`++;`
  `++`*l-expr*`;`

  è:

$$l\text{-}expr \texttt{ = } l\text{-}expr \texttt{ + 1;}$$

e, analogamente, il significato degli statement:

$$l\text{-}expr\texttt{--;}$$
$$\texttt{--}l\text{-}expr\texttt{;}$$

è:

$$l\text{-}expr \texttt{ = } l\text{-}expr \texttt{ - 1;}$$

Ad esempio:

```
x++;        //equivale a  x = x + 1
x--;        //equivale a  x = x - 1
```

La differenza fra la forma prefissa e quella postfissa degli operatori di autoincremento e autodecremento si evidenzia quando i relativi statement vengono utilizzati come espressioni.

Ad esempio è possibile scrivere:

```
y = ++x;
```

ma anche

```
y = x++;
```

Nel primo caso, la valutazione di `++x`, prima modifica il valore di `x`, incrementandolo di `1`, e quindi restituisce come risultato il valore di `x` modificato; nel secondo caso, invece, la valutazione di `x++`, prima restituisce come risultato il valore di `x` (non modificato) e poi modifica `x` incrementandolo di `1`. Quindi, se il valore di `x` è originariamente `1`, il valore assegnato a `y` dal primo statement è `2`, mentre quello assegnato dal secondo statement è 1; in entrambi i casi, il valore di `x` diventa `2`.

Come altro esempio, il seguente frammento di programma

```
int x = 10;
cout << ++x;
cout << x;
```

stampa due volte `11`; mentre il frammento di programma

```
int x = 10;
cout << x++;
cout << x;
```

stampa `10` e poi `11`.

Si noti che, in ogni caso, le diverse forme contratte di assegnamento costituiscono delle semplici estensioni sintattiche, comode da usare, ma di cui si può fare a meno, dato che sono sempre rimpiazzabili dagli equivalenti statement che usano l'assegnamento di base. L'utilizzo delle forme abbreviate può servire a scrivere codice più sintetico ed, eventualmente, a permettere al compilatore di individuare più facilmente forme particolari di assegnamento e quindi di generare codice macchina più efficiente.

Infine si noti che i diversi operatori di assegnamento si applicano su *l-expr* qualsiasi e non soltanto a variabili, come mostrato negli esempi sopra. Sarà quindi possibile scrivere, ad esempio, `A[i]++` o `A.c += 5` dove `A[i]` e `A.c` sono forme di *l-expr* più complesse che introdurremo nei capitoli successivi.

2.7 Input/output di base

Abbiamo già visto nel Capitolo 1.1 che, tipicamente, un algoritmo riceve dei dati su cui operare (dati in ingresso o *input*) e produce dei risultati (dati in uscita o *output*). Tutti i linguaggi di programmazione forniscono perciò opportuni strumenti per esprimere operazioni di input/output (o I/O). Questi strumenti possono essere istruzioni primitive del linguaggio (come la `read` e la `write` del Pascal) oppure sottoprogrammi predefiniti facenti parte dell'ambiente di programmazione standard del linguaggio (si veda il Capitolo 1.4).

Il C++ adotta la seconda soluzione, offrendo un insieme piuttosto ampio di funzioni di libreria ed astrazioni varie atte a supportare diverse forme di input/output. L'utilizzo di astrazioni definite dall'ambiente piuttosto che di istruzioni primitive del linguaggio risulta senz'altro più flessibile, non vincolando la definizione del linguaggio a specifiche modalità di input/output e quindi permettendo un loro più facile adattamento alle mutevoli esigenze delle diverse applicazioni. Questa flessibilità, come già sottolineato, è uno dei motivi principali del successo del C/C++.

La forma standard di input/output utilizzata in C++ é quella basata su *stream*, un tipo di dato astratto predefinito che rappresenta un *canale di comunicazione* tra un programma ed una sorgente/destinazione di dati. Quest'ultima puó essere sia un dispositivo esterno di I/O (ad esempio la tastiera o il monitor), sia un generico file. In questo capitolo esamineremo soltanto le possibilità di base offerte dal C++ per l'input/output su stream, in particolare per la comunicazione con i dispositivi standard di I/O (tastiera e monitor). Ritorneremo, in modo piú approfondito e generale, sulla nozione di stream nel Capitolo 7 dove esamineremo l'input/output su file.

Dal momento che l'input/output in C++ è realizzato tramite funzioni di libreria è necessario prima di tutto includere nel programma le dichiarazioni richieste per il loro utilizzo. Nel caso dell'input/output basato su stream, questo è ottenuto inserendo—tipicamente all'inizio del programma—la direttiva di preprocessore

```
#include <iostream>
```

che permette di aggiungere al programma tutte le dichiarazioni che servono a realizzare la nozione di stream e le funzioni che operano su essi. In particolare, l'inclusione del file `iostream`, congiuntamente all'utilizzo dell'istruzione `using namespace std`, rende disponibile all'interno del programma due oggetti di tipo stream che consentono la comunicazione tra il programma ed i dispositivi di I/O standard:

- `cin` (standard input): stream di input, associato di default alla tastiera;
- `cout` (standard output): stream di output, associato di default al monitor.

.

Il C++ offre varie funzioni ed operatori predefiniti da applicare agli stream, in particolare agli stream `cin` e `cout`, per realizzare varie forme di input/output. In questo capitolo esamineremo in dettaglio il funzionamento dei due operatori `>>` e `<<`, detti, rispettivamente, di *estrazione* e di *inserimento*, che costituiscono i due strumenti base per l'input/output su stream in C++. In un capitolo successivo ritorneremo su altre problematiche legate all'utilizzo di tali operatori ed introdurremo altre funzioni di input/output (in particolare, le funzioni `get` e `put`).

2.7.1 Lettura da stream tramite >>

Un'operazione di lettura (input) da stream tramite l'operatore di estrazione `>>` è un'espressione della forma:

`s >> v`

dove `s` è uno stream di input (in particolare `cin`) e `v` è una variabile di tipo t. La libreria `iostream` fornisce la definizione di `>>` per i tipi semplici primitivi `int`, `char`, `float` e `double`, oltre che per le stringhe (array di caratteri). Inoltre, come vedremo meglio nella II Parte del testo, l'operatore `>>` può essere ridefinito anche per altri tipi di dato definiti da utente o definiti nelle librerie standard (come ad esempio il tipo `string`), sfruttando il meccanismo di "overloading" degli operatori fornito dal C++.

Il significato dell'espressione `s >> v` è il seguente: legge (= estrae) dallo stream `s` un dato di tipo t, se presente, e lo assegna a `v`; altrimenti, se il dato non è ancora presente, attende.

Si noti che il dato letto deve essere una costante di tipo t sintatticamente corretta. Se, ad esempio, t è `int`, l'esecuzione di `>>` cercherà di leggere dallo stream di input una costante intera (sintatticamente corretta), ovvero una sequenza di caratteri numerici, eventualmente preceduti da un segno (caratteri `'+'` o `'-'`). La lettura del numero intero termina non appena si incontra il primo carattere non numerico (ad esempio un carattere spazio o a capo) successivo alla sequenza di caratteri numerici che compongono il numero. Il numero letto viene rimosso dallo stream di input, mentre il carattere non numerico che delimita il numero rimane sullo stream di input.

Esempio 2.8 *Se si esegue*

```
int x;
cin >> x;
```

e lo stream di input `cin` *è* `32\n` *(*`'\n'` *indica il carattere a capo generato tramite il tasto di 'invio'), allora al termine dell'esecuzione dell'operazione di estrazione la variabile* `x` *conterrà l'intero 32 (ovviamente, rappresentato in binario, ad esempio in complemento a 2) e lo stream di input conterrà il solo carattere* `'\n'`*.*

Se sullo stream di input nel momento in cui si esegue l'operazione di lettura è presente un dato non del tipo previsto il dato viene rifiutato (precisamente, lo stream viene impostato allo stato `failed` e alla variabile `v` non viene assegnato alcun valore). Di conseguenza il comportamento del programma risulterà in generale non corretto. Ad esempio, se lo stream di input contiene `12*5\n`, l'esecuzione dell'istruzione `cin >> x`, con `x` di tipo `int`, memorizzerà in `x` il numero intero `12`, ma l'esecuzione di una successiva istruzione `cin >> y`, con `y` di tipo `int`, trovando in input il carattere `'*'`, porterà ad una situazione di errore.

Spesso i dati (numerici e non) sullo stream di input sono separati tra loro da uno o più caratteri "spazio" o "a capo". In questo caso si tenga presente che l'operatore `>>` ignora eventuali caratteri "spazio" o "a capo" precedenti un dato da leggere: i caratteri "spazio" e "a capo" vengono rimossi dallo stream di input, ma non considerati parte del dato letto. Nell'esempio di sopra, se lo stream di input fosse `12  5\n` (e cioè i due numeri fossero separati da uno o più caratteri spazio), l'esecuzione in sequenza delle due istruzioni `cin >> x; cin >> y` verrebbe portata a termine correttamente, assegnando a `x` il numero `12` e a `y` il numero `5`.

Come ulteriore esempio per chiarire la natura "tipata" dell'operatore `>>`, mostriamo come la stessa sequenza di caratteri presente sullo stream di input possa venir interpretata e memorizzata in modo diverso a seconda del tipo della variabile che compare nell'operazione di estrazione.

Esempio 2.9 *Si supponga di avere:*
Stream `cin`*:* `-12\n`

Se si esegue:

```
int x;
cin >> x;
```

allora sullo stream di input viene cercata e prelevata una costante di intero (sintatticamente corretta) che viene poi memorizzata nella variabile `x`*; dunque, al termine dell'esecuzione dell'operazione di estrazione,* `x` *conterrà l'intero* -12.

Se invece si esegue:

```
char x;
cin >> x;
```

allora sullo stream di input viene cercata e prelevata una costante di carattere (sintatticamente corretta) che viene poi memorizzata nella variabile `x`*; dunque, in questo caso, al termine dell'esecuzione dell'operazione di estrazione,* `x` *conterrà il carattere* `'-'` *(rappresentato in binario secondo la codifica ASCII). Una eventuale successiva operazione di lettura troverà sullo stream di input la sequenza di caratteri* `12\n`.

Dal momento che l'operazione di estrazione è un'espressione e che restituisce come suo risultato proprio (un riferimento al)lo stream di input modificato, risulta semplice combinare più operatori di estrazione "in cascata":

`s >> v1 >> v2 >> ... >> vn`

dove `v1`, `v2`, ..., `vn` sono variabili, di tipo, rispettivamente, `t1`, `t2`, ..., `tn`. Essendo l'operatore `>>` associativo a sinistra, la valutazione procederà da sinistra a destra, valutando per prima l'espressione `s >> v1`, che toglierà dallo stream di input `s` il primo dato presente (che deve essere di tipo `t1`), e quindi restituirà lo stream di input così modificato a cui verrà applicato il secondo operatore di estrazione che leggerà il secondo dato assegnandolo alla variabile `v2`, e così via.

Esempio 2.10 *Se si esegue*

```
int x, y;
cin >> x >> y;
```

e lo stream di input `cin` *è*

`3 5 1\n`,

allora: dopo il primo `>>`*,* `x` *vale* `3`*, il valore di* `y` *é (ancora) indeterminato, lo stream* `cin` *è*

`5 1\n`

(il `3` *è stato rimosso dallo stream); dopo il secondo* `>>`*,* `x` *vale* `3`*,* `y` *vale* `5`*, lo stream* `cin` *è*

`1\n`

(anche il `5` *è stato rimosso dallo stream).*

Il contenuto dello stream rimasto potrá eventualmente essere letto (e rimosso) tramite successive operazioni di estrazione.

Si noti che nell'Esempio 2.10 non è necessario eseguire alcuna operazione di lettura per rimuovere i caratteri "spazio" che separano i numeri tra loro.

Nota. È importante tener presente che l'input da tastiera è "bufferizzato" e quindi i dati digitati non vengono effettivamente presi in considerazione fintantochè l'utente non preme il tasto di 'invio' ('enter'). Con riferimento all'Esempio 2.10, assumendo di dare l''invio' solo dopo aver digitato tutti e tre i numeri (e cioè che lo stream di input sia `3 5 1\n`), il numero `3` non viene prelevato dallo stream, e quindi assegnato alla variabile `x`, finchè l'utente non ha digitato anche gli altri due numeri e premuto 'invio'. Viceversa se dopo ognuno dei primi due numeri si premesse il tasto di 'invio' (e cioè se lo stream di input fosse `3\n5\n1\n`), allora i numeri `3` e `5` verrebbero immediatamente letti ed assegnati alle relative variabili. Nel resto del testo preferiamo comunque ignorare questo tipo di dettagli lasciando al lettore di tenerne eventualmente conto nelle prove pratiche che vorrà effettuare.

2.7.2 Scrittura su stream tramite <<

Un'operazione di scrittura (output) su stream tramite l'operatore di inserimento `<<` è un'espressione della forma:

```
s << e
```
dove `s` è uno stream di output (in particolare `cout`) e `e` è un'espressione di tipo t. Come per l'operatore `>>`, la libreria `iostream` fornisce la definizione di `<<` sia per i tipi primitivi che per le stringhe (array di caratteri). Il programmatore può comunque ridefinire `<<` anche per tipi di dato definiti da utente, sfruttando il meccanismo di "overloading" degli operatori.

Il significato dell'espressione `s << e` è il seguente: immette sullo stream `s` il valore ottenuto dalla valutazione dell'espressione `e` (n.b., prima valuta `e` e quindi immette).

Esempio 2.11 *Se si esegue*

```
int x = 10;
cout << x;
```

sullo stream di output `cout` *verrà inserita la costante di intero* `10`*. Se successivamente si esegue*

```
int y = 20;
cout << x + y + 1;
```

allo stream di output `cout` *verrà aggiunta (di seguito alla costante* `10` *inserita in precedenza) la costante di intero* `31`*. Infine, se si esegue*

```
cout << "Termine prova";
```

allo stream di output `cout` *verrà aggiunta la costante di stringa* `Termine prova`*. Dunque al termine dell'esecuzione delle tre istruzioni di output lo stream* `cout` *sarà:* `1031Termine prova`*.*

Si noti che, di default, non vengono inseriti spazi aggiuntivi prima o dopo i dati inviati sull'output (in particolare prima o dopo le costanti numeriche). Nell'esempio di sopra, quindi, le tre costanti inserite in `cout` vengono visualizzate sul monitor una di seguito all'altra senza alcuna separazione.

Anche l'espressione di inserimento, come giá notato per quella di estrazione, restituisce come suo risultato lo stream di output modificato, e quindi risulta immediato combinare più operatori di inserimento "in cascata":
```
s << e1 << e2 << ... << en
```
dove `ve1`, `e2`, ..., `en` sono variabili, di tipo, rispettivamente, `t1`, `t2`, ..., `tn`. Anche l'operatore `<<` é associativo a sinistra e quindi la valutazione procederà da sinistra a destra, valutando per prima l'espressione `s << e1`, che immetterà sullo stream di output `s` il valore di `e1` e quindi restituirà lo stream di output così modificato a cui verrà applicato il secondo operatore di inserimento, e così via.

Esempio 2.12 *Se si esegue*

```
int x = 1;
int y = 2;
cout << x << ' ' << y << "\nSomma: " << x + y;
```

al termine dell'esecuzione lo stream `cout` *sarà:* `1 2\nSomma: 3`*. Il carattere di controllo* `'\n'` *("nuova linea") non viene visualizzato sul monitor, ma provoca il posizionamento del cursore del monitor sulla riga successiva. Dunque l'output sul monitor nell'esempio di sopra sarà:*

```
1 2
Somma: 3
```

Si presti molta attenzione alla differenza che esiste tra le due espressioni `cout << x` e `cout << "x"`. Nella prima, `x` é una variabile la cui valutazione restituisce il valore ad essa associato; nella seconda, `"x"` é una *costante* la cui valutazione restituisce la costante stessa. Dunque l'esecuzione di `cout << "x"` nell'esempio di sopra provocherebbe la scrittura sul monitor del carattere `x` (invece che del valore `1`).

Per forzare il posizionamento su una nuova linea di output si puó utilizzare anche il manipolatore di stream `endl` ("end line"), presente nella libreria `iostream`. L'esecuzione di

```
cout << endl;
```

provoca l'inserimento nello stream di output `cout` del carattere di controllo `'\n'`, esattamente come se si eseguisse `cout << '\n'`. Inoltre l'`endl` provvede anche a svuotare l'eventuale buffer di output, ovvero forza l'invio dei dati di output fino a quel punto accumulati al dispositivo di output associato allo stream. Dunque, spesso si preferisce inserire un `endl` come ultimo operando di un'istruzione di stampa, eventualmente composta da piú `<<` concatenati, piuttosto che un semplice `'\n'`.

Nota. Oltre a `cin` e `cout`, la libreria `iostream` mette a disposizione anche un altro oggetto di tipo stream (di output) denominato `cerr` ("standard error"). Questo stream é associato di default al monitor (ma potrebbe facilmente venir associato anche ad un file). `cerr` puó venir utilizzato, in alternativa a `cout`, per inviare all'utente tutti i messaggi di errore o, piú in generale, di controllo del funzionamento del programma, generati durante l'esecuzione del programma stesso. Il fatto di distinguere `cerr` da `cout` consente al programma di tenere separati i messaggi di errore da quelli di interazione "normale" con l'utente. Se sia `cerr` che `cout` sono associati al monitor, di fatto l'utente non si accorge di questa separazione; ma se uno dei due viene ridiretto su un file (ad esempio, per creare un file di "log" in cui tenere tutti i messaggi sullo stato d'avanzamento del programma) allora la separazione diventa piú significativa ed evidente.

Capitolo 3

Costrutti per il controllo di sequenza

Nella descrizione di un algoritmo è fondamentale poter specificare l'ordine in cui si susseguono le diverse istruzioni (ovvero, il *controllo di sequenza*). Nei diagrammi di flusso, abbiamo visto, l'ordine è specificato molto semplicemente tramite frecce che connettono le diverse istruzioni. Nei linguaggi di basso livello, tipicamente, sono presenti a questo scopo *istruzioni di salto* che permettono di modificare, quando opportuno, il comportamento di "default" della macchina hardware che è quello di eseguire le singole istruzioni una dopo l'altra, in sequenza, così come appaiono memorizzate nella memoria principale del calcolatore. I linguaggi di programmazione ad alto livello, di tipo convenzionale, offrono un più ricco insieme di costrutti sintattici per specificare, in termini più astratti, diverse tipiche forme di controllo di sequenza. In tutti questi linguaggi sono presenti statement per specificare (almeno) le seguenti forme di controllo del flusso d'esecuzione:

- sequenzializzazione
- selezione
- iterazione
- salto.

Per quanto riguarda la *sequenzializzazione* i linguaggi ad alto livello adottano solitamente soluzioni molto semplici come, ad esempio, l'uso esplicito di un operatore di sequenzializzazione (ad esempio il **;** in Pascal) o l'assunzione che la sequenzializzazione sia la regola implicita di default. Ad esempio, in C++ gli statement sono eseguiti in sequenza, uno dopo l'altro, come appaiono nel testo, a meno che non si trovino all'interno di uno statement che modifica esplicitamente il flusso d'esecuzione.

Nota. In C++ il **;** è semplicemente un "terminatore" di statement (e di dichiarazioni), e non un operatore che indica la sequenzializzazione di due statement, come accade ad esempio in Pascal. Per questo in C++, qualsiasi statement (o dichiarazione, tranne lo statement composto) deve essere terminata da un `;`, indipendentemente da ciò che segue (ad esempio, in C++, l'ultimo statement del programma, prima della parentesi graffa finale, va comunque terminato da un `;`).

Nei paragrafi successivi descriveremo in dettaglio gli statement offerti dal C++ per modellare le diverse forme di controllo di sequenza sopra citate. In particolare analizzeremo:

- gli statement `if` e `switch` per le strutture di controllo selettive;
- gli statement `while`, `do-while` e `for` per le strutture di controllo iterative;
- gli statement `break`, `continue` e `goto` per i salti espliciti.

Prima di descrivere in dettaglio i singoli statement, introdurremo, nel paragrafo 3.1, la nozione di *blocco*, mostrando il costrutto offerto dal C++ per realizzare tale nozione. In un capitolo successivo, invece, prenderemo in esame altri costrutti e meccanismi che permettono di realizzare altre forme di astrazione sul controllo, quali quelli per la definizione e la chiamata di sottoprogrammi ed il meccanismo della ricorsione.

Nota. Si osservi che le *espressioni*, introdotte nel capitolo precedente, costituiscono già una forma di controllo del flusso d'esecuzione. Nella valutazione di un'espressione, infatti, si procede seguendo un ben preciso ordine tre le varie parti costituenti l'espressione: tipicamente, valutando prima gli operandi, da sinistra a destra, e poi applicando l'operatore e, nel caso in cui un'espressione sia composta da più operatori, seguendo precise regole di precedenza ed associatività.

3.1 Statement composto

I linguaggi di programmazione convenzionali offrono normalmente un costrutto—detto *statement composto*—che permette di raggruppare più statement per formare un unico statement. Lo statement composto in C++ ha la seguente forma:.

$$\begin{array}{l} \{\ \textit{dichiarazioni} \\ \qquad\quad + \\ \quad\ \textit{statement} \\ \} \end{array}$$

e cioè un insieme di dichiarazioni (eventualmente vuoto) unito ad un insieme di statement (eventualmente vuoto), il tutto racchiuso tra una coppia di parentesi graffe.

Esempio 3.1 (Statement composto) *Il seguente frammento di codice C++ costituisce un esempio di statement composto:*

```
{ int x, y;
  x = y + 1;
  float z;
  z = x;
}
```

La regione di testo racchiusa tra le due graffe costituisce un *blocco*, nozione fondamentale per la strutturazione dei programmi nei linguaggi di programmazione moderni che incontreremo nuovamente e preciseremo in un paragrafo successivo dedicato ai problemi di visibilità degli identificatori. Per ora anticipiamo soltanto che tutti gli identificatori eventualmente dichiarati all'interno di un blocco sono *locali* al blocco e cioè utilizzabili soltanto all'interno di quel blocco o di eventuali altri blocchi in esso contenuti.

Si osservi che un programma principale, come quello mostrato nel paragrafo 1.3.2, è costituito da una *testata*, che di base ha la forma `int main()`, seguita dal *corpo* del programma che di fatto è uno statement composto.

3.2 Statement `if`

Tra le strutture di controllo selettive più comuni figurano senz'altro quelle che permettono l'"esecuzione condizionale" di un'istruzione e la "biforcazione", già incontrate nel paragrafo 1.1.3 relativo ai diagrammi di flusso. Queste strutture di controllo sono realizzate nella maggior parte dei linguaggi di programmazione attuali da una o più forme di statement `if`. Vediamo sintassi e semantica di questo statement nel caso del linguaggio C++.

3.2.1 Caso base

Sintassi

$$\texttt{if (}E\texttt{)} \quad S\texttt{;}$$

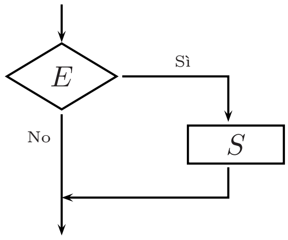
dove

E: espressione booleana;
S: statement qualsiasi.

Semantica (informale)

Calcola il valore dell'espressione E; se E ha valore "vero" allora esegue lo statement S e quindi termina l'esecuzione dello statement; altrimenti, termina immediatamente.

Il caso descritto è rappresentabile con un diagramma di flusso nel seguente modo (*selezione singola*):



Si tratta di una forma di *esecuzione condizionale*: lo statement S viene eseguito solo se la condizione E è vera.

Esempio 3.2 *Il seguente semplice frammento di programma calcola il valore assoluto di un numero intero letto da standard input:*

```
int x;
cin >> x;
if (x < 0)
  x = -x;
cout << "Il valore assoluto e' " << x;
```

Lo statement `x = -x` *viene eseguito soltanto se la condizione* `x < 0` *è vera. In ogni caso, terminata l'esecuzione dell'*`if`*, si passa allo statement successivo, che, in questo esempio, provvederà a stampare il valore corrente della variabile* `x`.

Qualora le istruzioni da eseguire quando la condizione dell'`if` è vera siano più di una, è necessario racchiuderle all'interno di uno statement composto, in modo che possano essere viste come un unico statement (si osservi che la sintassi dell'`if` prevede che S sia un singolo statement, che potrebbe comunque essere anche lo statement composto). Come esempio, si supponga di voler modificare il frammento di programma mostrato nell'Esempio 3.2 in modo che, quando `x < 0` è vero, allora venga stampato anche un messaggio che informa che il numero dato è negativo. Lo statement `if` dell'esempio 3.2 viene sostituito da:

```
if (x < 0) {
  cout << "Il numero dato e' negativo" << endl;
  x = -x;
}
```

Nota. Si osservi che senza racchiudere tra graffe i due statement da eseguire quando la condizione dell'`if` è vera il programma risultante

```
int x;
cin >> x;
if (x < 0)
  cout << "Il numero dato e' negativo" << endl;
  x = -x;
cout << "Il valore assoluto e' " << x;
```

avrebbe, in generale, un comportamento totalmente diverso (e non corretto rispetto alle nostre intenzioni originarie). Infatti, in questo caso, il compilatore considera come parte del costrutto `if` l'istruzione di stampa `cout << ... << endl`, ma non l'istruzione di assegnamento successiva, che pertanto è eseguita sempre, indipendentemente dal valore dell'espressione `x < 0`. Dunque, se, ad esempio, il dato in input è `-3`, il programma stampa correttamente

```
Il numero dato e' negativo
Il valore assoluto e' 3
```

ma, se il dato in input è `3`, il programma stampa

```
Il valore assoluto e' -3
```

che evidentemente non è il risultato desiderato.

3.2.2 Caso `if-else`

Lo statement `if` ammette anche un'altra forma, in cui è prevista una parte introdotta dalla parola chiave `else`.

Sintassi

`if (`E`)` S_1`;`
`else` S_2`;`

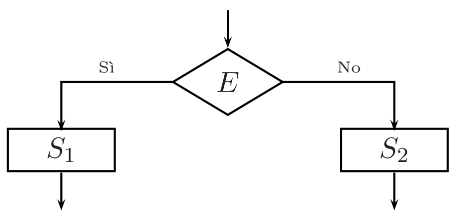
dove

E: espressione booleana;
S_1, S_2: statement qualsiasi.

Semantica (informale)

Calcola il valore dell'espressione E; se E ha valore "vero" allora esegue lo statement S_1, altrimenti esegue lo statement S_2; quindi, in entrambi i casi, termina l'esecuzione dello statement.

Anche in questo caso la situazione è rappresentabile con un diagramma di flusso (*selezione doppia*):



Si ha dunque una *biforcazione* nel flusso di esecuzione: si esegue uno tra gli statement S_1 o S_2 in base al verificarsi o meno della condizione E.

Esempio 3.3 *Il seguente semplice frammento di programma determina il maggiore tra due numeri interi letti da standard input e lo scrive sullo standard output:*

```
int x, y, max;
cin >> x >> y;
if (x < y)
  max = x;
else
  max = y;
cout << "Il maggiore e’ " << max;
```

(per il diagramma di flusso relativo allo statement `if-else` *di questo esempio si veda il paragrafo 1.1.3).*

Anche per lo statement che compare nella parte `else` valgono le stesse considerazioni fatte nel caso dell'`if` semplice: qualora le istruzioni da eseguire siano più di una è necessario che siano racchiuse tra le parentesi graffe di uno statement composto, come mostra il seguente frammento di codice:

```
if (x >= y) {
  max = x;
  min = y;
}
else {
  max = y;
  min = x;
}
```

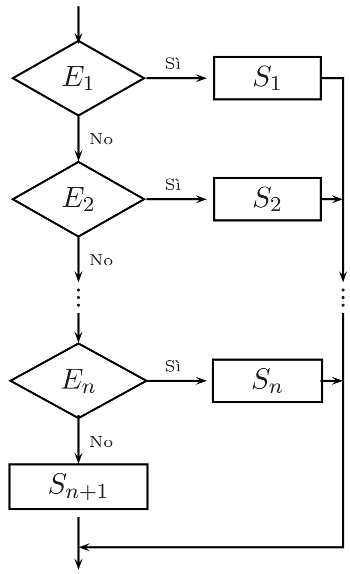
Si osservi che lo statement composto non vuole il "`;`" dopo la "`}`".

3.2.3 Statement `if-else` annidati

Gli statement S, S_1 ed S_2 delle diverse forme di `if` possono essere statement qualsiasi. In particolare possono essere a loro volta statement `if`, nel qual caso si parla di `if` "annidati" (o "nidificati"). Un caso particolarmente interessante di `if` annidati è quello che realizza una struttura di controllo del tipo *"test multiplo"* (o *selezione multipla*):

```
if (E1)  S1;
else
  if (E2)  S2;
  else
    ...
      else
        if (En)  Sn;
        else Sn+1;
```

Come mostra l'indentazione, il ramo `else` relativo al primo `if` contiene a sua volta un `if`; il ramo `else` relativo al secondo `if` contiene ancora un `if` e così via. La struttura di controllo realizzata da queste istruzioni è quella descritta dal seguente diagramma di flusso:



Si osservi che le espressioni booleane E_1, E_2, ..., vengono valutate nell'ordine in cui compaiono. Pertanto lo statement S_i eseguito sarà quello associato alla prima condizione S_i che risulta vera. In particolare, lo statement S_n viene eseguito solamente nel caso in cui tutte le precedenti espressioni booleane abbiano valore falso. In ogni caso, soltanto uno degli statement S_i viene eseguito, mentre tutti gli altri vengono ignorati.

Nota. Per evidenziare meglio la struttura del test multiplo può essere conveniente utilizzare una diversa indentatura degli statement `if` annidati, come mostrato qui di seguito:

```
if (E1)  S1;
else if (E2)  S2;
...
else if (En)  Sn;
else  Sn+1;
```

Il seguente esempio di programma completo mostra l'utilizzo di più `if` annidati per realizzare un test multiplo.

Esempio 3.4 (Conversione *voti* $\rightarrow$ *giudizi*)
Problema*. Scrivere un programma in grado di convertire un voto numerico tra 0 e 10 in un giudizio, secondo il seguente schema:*

voto $\leq$ *5,* *giudizio: insufficiente*
5 $<$ *voto* $\leq$ *6.5,* *giudizio: sufficiente*
6.5 $<$ *voto* $\leq$ *7.5,* *giudizio: buono*
voto $>$ *7.5,* *giudizio: ottimo*

Input*: un numero reale.*
Output*: una stringa di caratteri (cioè il giudizio) oppure un messaggio di errore.*

Programma*:*

```
#include <iostream>
using namespace std;
int main() {
  float voto;
  cout << "Dammi il voto numerico (tra 0 e 10)" << endl;
  cin >> voto;
  if (voto < 0 || voto > 10)
    cout << "voto non valido" << endl;
  else if (voto <= 5)
    cout << "insufficiente" << endl;
  else if (voto <= 6.5)
    cout << "sufficiente" << endl;
  else if (voto <= 7.5)
    cout << "buono" << endl;
  else
    cout << "ottimo" << endl;
  cout << "arrivederci" << endl;
  return 0;
}
```

Esempio d'esecuzione *(le parti sottolineate rappresentano input):*

```
Dammi il voto numerico (tra 0 e 10)
6
sufficiente
arrivederci
```

Nota*. Si noti che nei test relativi ai giudizi "sufficiente" e "buono" non è necessario specificare anche l'estremo inferiore del voto. Ad esempio, per il giudizio buono basta dare la condizione* `(voto <= 7.5)` *e non necessariamente* `(voto > 6.5 && voto <= 7.5)` *dato che, nel momento in cui si esegue il test, sicuramente sono già stati effettuati i test precedenti e quindi sono già stati esclusi i valori di voto inferiori o uguali a* `5`*.*

Esercizio 3.5 *Scrivere un programma C++ che legge da standard input due numeri interi e determina se i due numeri sono uguali o se uno è maggiore dell'altro e, nel primo caso, stampa il messaggio "*`I due numeri` *`sono uguali`" mentre, nel secondo caso, stampa il maggiore tra i due numeri (SUGG.: completare il frammento di programma dell'Esempio 3.3, utilizzando due `if` annidati).*

Variante #1 per l'esempio 3.4.
Il programma dell'esempio 3.4 può essere riscritto utilizzando una sequenza di statement `if` semplici, piuttosto che più `if-else` annidati. Mostriamo qui soltanto le parti del programma modificate:

```
...
cin >> voto;
if (voto < 0 || voto > 10)
  cout << "voto non valido" << endl;
if (voto > 0 && voto <= 5)
  cout << "insufficiente" << endl;
if (voto > 5 && voto <= 6.5)
  cout << "sufficiente" << endl;
if (voto > 6.5 && voto <= 7.5)
  cout << "buono" << endl;
if (voto > 7.5 && voto <= 10)
  cout << "ottimo" << endl;
cout << "arrivederci" << endl;
...
```

I risultati prodotti dalle due versioni del programma sono gli stessi, ma la nuova versione (variante #1) risulta meno soddisfacente per almeno due motivi:

- è più pesante da scrivere, in quanto richiede di esplicitare tutte le condizioni, indicando sia estremo inferiore che superiore dei diversi intervalli;
- è più inefficiente, in quanto richiede di eseguire sempre comunque tutti i test, anche se uno soltanto avrà effettivamente esito positivo (e cioè la condizione risulterà vera).

Esercizio 3.6 *Scrivere i diagrammi di flusso delle due versioni dei programmi per la conversione di voti in giudizi e confrontarli tra loro.*

Approfondimento (Ambiguità tra `if` annidati). L'utilizzo di `if-else` annidati può portare a situazioni di possibile ambiguità. Si consideri ad esempio il seguente frammento di programma:

```
if (trovato == true)
if (a == 0)
cout << "a e' nullo" << endl;
else
cout << "a non e' nullo" << endl;
```

A quale `if` corrisponde il ramo `else`? L'ambiguità è risolta in C++—come nella maggior parte dei linguaggi convenzionali—assumendo che un `else` faccia riferimento sempre all'`if` più vicino che non abbia già un `else` associato. Quindi, nell'esempio di sopra, l'`else` si riferisce al secondo `if` (se ci fosse anche un altro `else`, successivo a quello già presente, questo ulteriore `else` farebbe chiaramente riferimento al primo `if`, in quanto il secondo sarebbe già "completo"). L'indentatura del programma può facilitare la comprensione dell'annidamento degli `if`. Conviene quindi scrivere l'esempio di sopra come:

```
if (trovato == true)
  if (a == 0)
    cout << "a e’ nullo" << endl;
  else
    cout << "a non e’ nullo" << endl;
```

Se vogliamo che il ramo `else` si riferisca al primo `if` è necessario inserire il secondo `if` all’interno di uno statement strutturato nel modo seguente:

```
if (trovato == true) {
  if (a == 0)
    cout << "a e’ nullo" << endl;
}
else
  cout << "Non e’ stato trovato l’elemento cercato" << endl;
```

3.3 Statement `while`

Un altro tipo di struttura di controllo che si incontra con grande frequenza nella progettazione di algoritmi è la struttura di controllo iterativa, caratterizzata dall’esecuzione ripetuta di una sequenza di istruzioni (un ciclo). Nella pratica, si individuano diverse tipologie di strutture iterative, distinte, ad esempio, in base alla posizione in cui compare la condizione d’uscita dal ciclo o al tipo di azioni che si compiono all’interno del ciclo. Tutti i linguaggi di programmazione convenzionali forniscono uno o più costrutti linguistici che permettono di realizzare, in modo più o meno naturale, le diverse tipologie di strutture di controllo iterative. Il C++, in particolare, offre tre diversi statement, `while`, `do-while` e `for`, specificatamente rivolti alla realizzazione di cicli, oltre alle istruzioni di salto esplicito che possono comunque essere utilizzate per la realizzazione di cicli.

In questo paragrafo descriveremo lo statement `while`, mentre gli altri statement iterativi e quelli di salto verranno descritti nei paragrafi successivi di questo capitolo.

Sintassi

$$\texttt{while}\ (E)\\ \quad S\texttt{;}$$

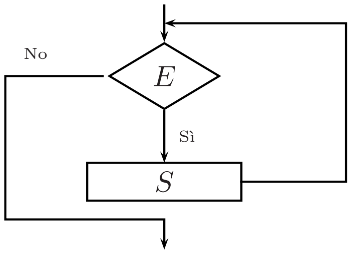
dove

E: espressione booleana;
S: statement qualsiasi.

Semantica (informale)

Calcola il valore dell'espressione E; se E ha valore "vero" allora esegue lo statement S e ripete dall'inizio; se E ha valore "falso" allora termina l'esecuzione dello statement.

La situazione è descrivibile con un diagramma di flusso nel seguente modo:



Lo statement `while` realizza in modo naturale una struttura di controllo iterativa in cui la condizione d'uscita dal ciclo è posta all'inizio ed in cui le azioni da ripetere (il "corpo del ciclo") possono essere qualsiasi. Di fatto, il ciclo continua finchè la condizione E non diventa falsa.

Ogni ripetizione dello statement S viene detta *iterazione* del ciclo. In generale, non è possibile stabilire a priori il numero di iterazioni, ma si osservi che, nel caso in cui l'espressione booleana E risulti subito falsa, allora non vi è alcuna iterazione (lo statement `S` non viene quindi eseguito, e si passa subito allo statement successivo al `while`).

Come nel caso dello statement `if`, qualora le istruzioni all'interno del ciclo siano più di una, è necessario che vengano racchiuse in uno statement composto, ossia tra parentesi graffe.

Un semplice esempio di utilizzo del `while` è mostrato nel seguente frammento di programma.

Esempio 3.7 *Stampa i quadrati dei primi 10 numeri interi positivi:*

```
int i = 1;
while (i <= 10){
   cout << i * i << endl;
   i = i + 1;
}
cout << "terminato!" << endl;
```

L'esecuzione di questi statement produrrà il seguente output:

```
1
4
9
...
100
terminato!
```

Il corpo del `while` può contenere anche altri statement strutturati, in particolare altri statement `while` (cicli annidati, di cui vedremo esempi più avanti) o statement `if`. Il seguente programma completo mostra un esempio di `while` contenente uno statement `if`.

Esempio 3.8 (Conta numero totale di vocali)
Problema. *Scrivere un programma che legge da standard input una sequenza di caratteri terminata da un punto, determina il numero di vocali minuscole presenti nella sequenza e quindi scrive il numero calcolato sullo standard output.*

Programma:

```
#include <iostream>
using namespace std;
int main() {
  char c;
  int vocali_minusc = 0;
  cout << "Inserisci sequenza di caratteri terminata da ."
       << endl;
  cin >> c;   //legge un carattere e lo assegna a c
  while (c != '.'){
    if (c=='a' || c=='e' || c=='i' || c=='o' || c=='u')
       ++vocali_minusc;
   cin >> c;
  }
  cout << "La sequenza data contiene " << vocali_minusc
       << " vocali minuscole" << endl;
  return 0;
}
```

L'esecuzione del programma, con la sequenza di input
   `Una frase di prova.`
produce il seguente output:
   `La sequenza data contiene 6 vocali minuscole.`

Esercizio 3.9 *Scrivere un programma C++ che legge da standard input una sequenza di numeri interi terminata da un numero negativo, calcola la media aritmetica dei numeri (non negativi) letti, e scrive il risultato sullo standard output (il programma controlla anche che la sequenza non sia vuota, nel qual caso informa l'utente con opportuno messaggio in output). (SUGG.: la struttura del programma è essenzialmente la stessa del programma dell'Esempio 3.8; si utilizza una variabile* `somma` *in cui "accumulare", man mano, la somma parziale dei numeri letti da input, inserendo all'interno del ciclo uno statement* `somma = somma + x`*, dove* `x` *è la variabile che contiene l'ultimo numero letto da input, ...).*

Esercizio 3.10 *Scrivere un programma che legge da standard input una sequenza di caratteri terminata da un punto e determina e stampa il numero di "doppie" presenti nella sequenza, dove per "doppie" si intende una sequenza di due caratteri consecutivi qualsiasi (ma diversi da spazio e a capo) identici. Ad esempio, data in input la sequenza*

```
Arrivero'   appena   possibile.
```

il programma deve indicare che sono presenti 3 doppie.

Nota (Cicli senza fine). Un aspetto critico delle strutture di controllo iterative, come quelle realizzate tramite il `while`, è dato dalla possibilità di scrivere cicli non terminanti. In generale, è compito di chi progetta l'algoritmo assicurarsi che i cicli presenti terminino in un tempo finito.

Un primo semplice controllo è assicurarsi che il valore dell'espressione che costituisce la condizione d'uscita dal ciclo venga in qualche modo modificata all'interno del ciclo (altrimenti la condizione, se inizialmente vera, rimarrebbe tale per sempre e quindi non porterebbe mai all'uscita dal ciclo). Si consideri, ad esempio, il seguente frammento di programma C++:

```
int i = 0;
while (i < 10)
  somma = somma + 1;
```

Poichè il valore della variabile che funge da variabile di controllo del ciclo (in questo caso la variabile `i`) non viene mai modificata all'interno del corpo del `while` e la condizione del `while` è inizialmente vera, si è sicuramente in presenza di un ciclo infinito.

Un altro semplice controllo da effettuare è di verificare che l'espressione booleana che funge da condizione d'uscita non sia sempre banalmente vera, indipendentemente dal valore delle variabili che eventualmente appaiono in essa. Ad esempio, il ciclo

```
int i = 1;
while (i > 0 || i <= 10)
  i = i + 1;
```

è chiaramente senza fine (probabilmente il programmatore, inesperto, voleva dire che `i` può variare tra `1` e `10` ma ha sbagliato connettivo logico ...).

Non sempre è così immediato individuare la presenza di un ciclo senza fine. I tre frammenti di programmi C++ che seguono sono altrettanti esempi di situazioni di ciclo senza fine. Probabilmente sono tutti frutto di errori di programmazione che, purtroppo, non è così infrequente commettere ....[1] Lasciamo al lettore attento scoprire perchè si ha un ciclo senza fine e qual'è la probabile versione corretta del codice.

```
int i = 1, j = 1;
while (i = 1){
  j = j + 1;
  if (j > 10) i = 0;
}
```

[1] Si noti che si tratta di errori semantici non sintattici e quindi non rilevabili dal compilatore; in pratica, il programma non fa quello che si vorrebbe, ma è sintatticamente corretto.

(SUGG.: si ricordi che l'assegnamento può essere usato anche come espressione, che il valore `1` è interpretato come valore "vero" in un'espressione booleana, che la relazione di uguaglianza in C++ si scrive `==` e non `=`, ...).

```
int i = 1;
while (i <= 10);
  {cout << i * i << endl;
   i = i + 1;
  }
cout << "terminato!" << endl;
```

(SUGG.: si consideri che il C++ ammette anche lo *statement vuoto*, e che lo statement del `while` può essere uno statement qualsiasi (anche vuoto) ...).

```
int i = 1;
while (i <= 10)
   cout << i * i << endl;
   i = i + 1;
cout << "terminato!" << endl;
```

(SUGG.: si ricordi che il corpo del `while` è costituito da un solo statement (che però può essere anche lo statement composto) ...).

Approfondimento (Il problema della terminazione). Il problema della terminazione dei programmi è senz'altro molto complesso: a partire dagli anni '70 hanno iniziato a nascere *metodi formali* per la verifica (statica) di correttezza dei programmi che comprendono tecniche per la dimostrazione formale di proprietà di terminazione dei cicli. Per un approfondimento si veda ad esempio il testo [7].

Gli statement visti finora, in particolare gli statement `if` e `while`, sono sufficienti a scrivere qualsiasi programma (ovvero a descrivere la soluzione di qualsiasi problema computabile). In altri termini, non è strettamente necessario introdurre altri costrutti; in particolare, non è richiesta la presenza nel linguaggio di programmazione di istruzioni di salto esplicito, quali il `goto`.

Normalmente, però, i linguaggi di programmazione convenzionali offrono anche altri statement per il controllo di sequenza, quali lo statement `for`, lo `switch` (o `case`), ecc. Questi ulteriori costrutti sono introdotti, in generale, non per aumentare le capacità computazionali del linguaggio (che, come si diceva, sono già complete con i pochi tipi di statement visti fin qui), ma per motivi essenzialmente metodologici, quali:

- facilitare la scrittura dei programmi, offrendo all'utente statement che modellano in modo più naturale le strutture di controllo che si vogliono realizzare;

- migliorare la comprensibilità del programma, permettendo l'utilizzo di diverse forme di astrazione sul controllo

- aumentare l'affidabilità dei programmi, riducendo le possibilità di commettere errori grazie all'utilizzo di costrutti più specializzati, ad esempio per realizzare cicli limitati, test multipli, ecc.

Approfondimento (Programmazione strutturata). BOZZA - Teorema di Böhm-Jacopini: fondamentale per chiudere la disputa sulla necessità o meno del `goto`. Sufficienti solo statement di controllo *strutturati*: realizzano strutture di controllo con un solo punto di ingresso ed un solo punto di uscita. Utili per la verifica (formale) di correttezza di un programma; ciascun statement strutturato può essere visto come una "scatola nera" (non ci interessa come è fatto dentro), con un'entrata ed un'uscita, tramite cui viene collegata in sequenza alle "scatole nere" degli altri statement.

3.4 Statement `do-while`

Spesso si ha a che fare con strutture di controllo iterative in cui il test d'uscita è posto alla fine del corpo del ciclo. Per modellare nel modo più naturale possibile queste situazioni, molti linguaggi di programmazione, tra cui il C++, offrono uno statement apposito. Nel caso del C++ si tratta dello statement `do-while`.

Sintassi

```
do
  S;
while (E);
```

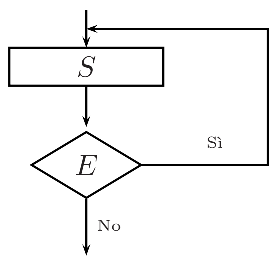
dove

E: espressione booleana;
S: statement qualsiasi.

Semantica (informale)

Esegue lo statement S; quindi valuta l'espressione E: se E ha valore "vero" allora ripete dall'inizio; altrimenti termina l'esecuzione dello statement.

La situazione è descrivibile con un diagramma di flusso nel seguente modo:



In altri termini, si esegue ripetutamente lo statement S per tutto il tempo che la condizione E rimane "vera" (si esce quando E assume valore "falso").

Si osservi che, a differenza dello statement `while`, nel `do-while` prima si esegue S e poi si valuta E. Dunque, con il `do-while`, si esegue sempre almeno un'iterazione.

Come già sottolineato, il `do-while` non è strettamente necessario e può essere sempre sostituito da un `while`. Precisamente, il generico costrutto `do-while` mostrato sopra è equivalente al frammento di programma:

```
S;
while (E)
   S;
```

Esercizio 3.11 *Realizzare il comportamento del costrutto* `while (E) S;` *tramite* `do-while`*. (SUGG.: inserire uno statement* `if` *all'inizio del corpo del* `do-while` *. . .).*

Due tipiche situazioni in cui l'utilizzo dello statement `do-while` risulta particolarmente comodo sono le seguenti:

Controllo dell'input: ripeti la lettura di un dato di input finchè il dato letto non soddisfa certe condizioni prestabilite. Ad esempio, se si deve leggere un numero intero `x` da standard input e si vuol imporre che il numero sia non negativo, si può sostituire la semplice istruzione `cin >> x;` con l'istruzione `do-while` seguente:

```
do
   cin >> x;
while (x < 0);
```

che impone al programma di ripetere la lettura se il dato letto è negativo (scartando di fatto tutti gli eventuali dati negativi letti). Dunque, all'uscita dal ciclo il dato contenuto in `x` sarà sicuramente non negativo.

Ripetizione programma: ripetere l'esecuzione di un programma fornendo ogni volta nuovi dati di input fintantochè l'utente non indichi esplicitamente di voler smettere.

Come esempio per illustrare questo modo di utilizzare il `do-while`, mostriamo come modificare il programma dell'Esempio 3.4 per permetterne l'esecuzione ripetuta.

Esempio 3.12 (Conversione voti → giudizi ripetuta)
Problema. *Scrivere un programma in grado di eseguire ripetutamente la conversione di un voto numerico tra 0 e 10 in un giudizio secondo lo schema indicato nell'*Esempio 3.4. *Al termine di ogni operazione di conversione, il programma dovrà richiedere all'utente se vuole continuare o no ed in caso di risposta positiva (carattere* `'s'`*) dovrà ripetere dall'inizio.*

Programma:

```
#include <iostream>
using namespace std;
int main() {
  float x;
  do {
     cout << "Dammi il voto numerico (tra 0 e 10)" << endl;
     cin >> voto;
     if (voto < 0 || voto > 10)
       cout << "voto non valido" << endl;
     else if (voto <= 5)
       cout << "insufficiente" << endl;
     else if (voto <= 6.5)
       cout << "sufficiente" << endl;
     else if (voto <= 7.5)
       cout << "buono" << endl;
     else
       cout << "ottimo" << endl;

     char ripeti;
     cout << "Altra conversione? ('s' per continuare): ";
     cin >> ripeti;
  }
  while (ripeti == 's');
  cout << "Arrivederci" << endl;
}
```

Esercizio 3.13 *Riscrivere il programma dell'Esempio 3.8 usando il costrutto* `do-while` *invece che* `while`*.*

3.5 Statement `for`

Un'altra forma molto comune di struttura di controllo iterativa è quella del cosiddetto *ciclo limitato* (o *controllato*, o *iterazione determinata*) che può essere schematicamente espressa nel modo seguente: "ripeti una certa azione per tutti i valori di x appartenenti ad un dato insieme finito di valori". Ad esempio: "stampa tutti i quadrati dei numeri interi da 1 a 100" (ovvero, "per tutti gli x tra 1 e 100 calcola e stampa $x * x$")

Il C++, come la maggior parte dei linguaggi di programmazione ad alto livello, offre uno specifico costrutto, lo statement `for`, che permette di realizzare (tra l'altro) questo tipo di cicli.

Sintassi

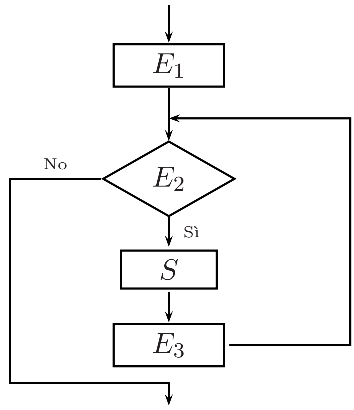
`for` $(E_1$; E_2; $E_3)$
  S;

dove

E_1, E_2, E_3:	espressioni qualsiasi;
S:	statement qualsiasi.

Semantica (informale)

(*i*) Valuta l'espressione E_1; (*ii*) valuta l'espressione E_2: se E_2 ha valore "falso" termina l'esecuzione del `for`; altrimenti, se E_2 ha valore "vero", esegue lo statement S e quindi valuta l'espressione E_3 e ripete da (*ii*).
La situazione è descrivibile con un diagramma di flusso nel seguente modo:



Nota. Il ciclo realizzato tramite `for` può sempre essere realizzato, in alternativa, utilizzando uno statement `while`. Precisamente, il generico costrutto `for` mostrato sopra è equivalente al seguente frammento di programma:

```
E1;
while (E2) {
  S;
  E3;
}
```

da cui risulta evidente, tra l'altro, che, come nel caso del `while`, potrebbe non esserci alcuna esecuzione dello statement S. Si osservi anche che le espressioni E_1 ed E_3 sono di fatto utilizzate come statement e quindi saremo verosimilmente interessati al possibile side-effect da esse provocato piuttosto che al valore ritornato dalla loro valutazione.

Questa forma molto generale di `for` può essere opportunamente specializzata per ottenere diverse forme di ciclo limitato o di struttura iterativa in genere.

Analizziamo dapprima in dettaglio come realizzare la forma base del ciclo limitato sopra menzionata; poi discuteremo brevemente le altre forme ed usi del `for`.

3.5.1 Ciclo limitato: caso base

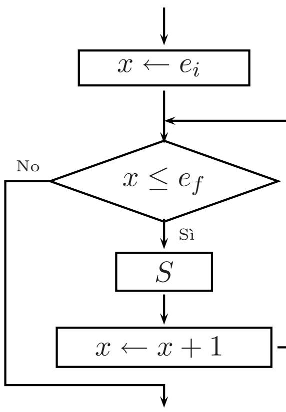
Vogliamo eseguire lo statement S per tutti i valori di una variabile x compresi tra un valore iniziale e_i ed un valore finale e_f. Per ottenere questo comportamento è sufficiente specializzare lo statement `for` nel modo seguente:

```
for (x = e_i; x <= e_f; x++)
  S;
```

dove

x:	variabile di tipo compatibile con `int`;
e_i, e_f:	espressioni di tipo compatibile con `int`;
S:	statement qualsiasi.

il cui significato è esprimibile con il seguente diagramma di flusso (in pratica si tratta di sostituire in modo opportuno le espressioni generali E_1, E_2 ed E_3 con le espressioni più specifiche che appaiono in questo uso particolare del `for`):



x costituisce la *variabile di controllo* del ciclo, mentre e_i ed e_f rappresentano, rispettivamente, il valore iniziale e finale di x. La prima espressione del `for` permette di inizializzare la variabile di controllo; la seconda costituisce la *condizione d'uscita* dal ciclo; la terza specifica il *passo* usato per l'incremento della variabile di controllo.

Il seguente frammento di programma C++ mostra un semplice esempio di utilizzo del `for` per la realizzazione di un ciclo limitato.

Esempio 3.14 *Stampa tutti i numeri interi compresi tra* 0 *e* 9 *in ordine crescente:*

```
int i;
for (i = 0; i <= 9; i++)
    cout << i << ' ';
```

Il tipo della variabile di controllo è normalmente `int`, ma può essere anche un qualsiasi tipo compatibile con `int` (ovvero un tipo *scalare*), come `char`, `bool`, un tipo definito per enumerazione, o un sottotipo di `int`, come `unsigned int`, `short`, ecc. In realtà il compilatore C++ non esegue alcun controllo sul tipo della variabile di controllo (nè tantomeno su quello delle espressioni e_i ed e_f) dato che in generale le espressioni del `for` possono essere di tipo qualsiasi. È perciò compito del programmatore fare in modo che tutte le espressioni coinvolte in un `for` usato come ciclo limitato abbiano tipo `int` o compatibile con `int`; il non rispetto di questo vincolo (utilizzando ad esempio tipi `float`) potrebbe portare alla non terminazione del ciclo.

In C++ la variabile di controllo del `for` può essere dichiarata direttamente all'interno del `for` stesso.[2] Ad esempio, il frammento di programma mostrato sopra può essere scritto alternativamente nel modo seguente:

```
for (int i = 0; i <= 9; i++) cout << i << ' ';
```

In questo caso, la variabile di controllo è *locale* al corpo dello statement `for` e quindi <u>non</u> può essere utilizzata al suo esterno.

Nota. Si osservi che le espressioni x `=` e_i e x`++` usate nello statement `for` sono in realtà statement di assegnamento che, come detto in precedenza, il C++ permette di usare anche come espressioni. Si osservi anche l'uso—non strettamente necessario, ma assai frequente—della forma contratta dell'assegnamento (x`++`) per realizzare l'incremento della variabile di controllo. In questo caso si potrebbe utilizzare in modo del tutto equivalente anche la forma prefissa (`++`x) in quanto qui si è interessati alla modifica della variabile x piuttosto che al valore ritornato dall'assegnamento. Dunque usare (x`++`) o (`++`x) in questo contesto è solamente una questione di stile e noi abbiamo scelto di utilizzare in questo testo sempre la forma postfissa.

Anche nel caso del `for` qualora le istruzioni all'interno del ciclo siano più di una è necessario racchiuderle in un blocco, ossia tra parentesi graffe. Vedremo altri esempi più complessi di utilizzo dello statement `for` per realizzare cicli limitati quando introdurremo le strutture dati di tipo array.

Approfondimento (Controllo ciclo limitato). Il C++ non prevede alcun controllo per garantire che un ciclo realizzato tramite `for` sia davvero un ciclo limitato e quindi termini in un numero finito di passi. In particolare in C++ è possibile modificare liberamente la variabile di controllo e i suoi limiti iniziale e finale all'interno del corpo del ciclo. Ad esempio possiamo scrivere

```
for (int i = 0; i < 9; i++) i--;
```

che porta chiaramente ad un ciclo infinito.

Per garantire che un ciclo limitato termini sempre in un tempo finito bisognerebbe porre delle forti restrizioni sui valori iniziale e finale della variabile di controllo, sulla condizione di uscita dal ciclo e sul modo in cui la variabile di controllo viene modificata all'interno del ciclo stesso. In certi linguaggi di programmazione, come ad esempio il Pascal, queste restrizioni sono imposte nella forma sintattica del `for`, permettendo così

[2] A rigore questo significa che la sintassi del `for` non è esattamente quella mostrata all'inizio di questo paragrafo, ma andrebbe estesa in modo da prevedere che l'espressione E_1 possa essere sostituita anche da una dichiarazione di variabile.

di effettuare i controlli che garantiscono la finitezza del ciclo `for` già a tempo di compilazione. La scelta del C++, invece, è come al solito quella di favorire la flessibilità d'uso dei suoi costrutti, lasciando completamente al programmatore la responsabilità di garantire il funzionamento corretto dei programmi realizzati. Come detto sopra, lo statement `for` del C++ è un costrutto molto più generale del `for` di altri linguaggi, utilizzabile per realizzare svariate strutture di controllo, non soltanto il cicloo limitato.

Esercizio 3.15 *Scrivere un programma C++ che calcoli e stampi il prodotto dei primi 10 numeri naturali positivi (ovvero* 10!*) utilizzando uno statement* `for` *(SUGG.: si utilizzi una variabile* `i` *che assume, uno alla volta, i valori da 1 a 10, e per ogni nuovo valore di* `i` *si calcoli il prodotto tra* `i` *e il prodotto parziale calcolato in precedenza ed "accumulato" in una variabile* `p`*, ...).*

Esercizio 3.16 *Come per l'Esercizio 3.15, ma con il numero n dei naturali positivi da moltiplicare tra loro letto da standard input (ovvero, calcola $n!$). (SUGG.: il programma inizia leggendo in una variabile* `N` *il numero n, possibilmente verificando anche che sia ammissibile, e cioè > 0); quindi procede come nell'Esercizio 3.15, ma con il valore finale del ciclo* `for` *costituito da* `N` *(invece che da* `10`*) ...).*

3.5.2 Altri utilizzi dello statement `for`

Come più volte sottolineato, il C++ è molto permissivo riguardo all'utilizzo delle espressioni all'interno del costrutto `for`, dando così la possibilità di utilizzare questo costrutto anche per realizzare strutture di controllo diverse da quella del ciclo limitato di base visto finora.

In particolare, possiamo facilmente realizzare altre forme di ciclo limitato, ma con passo diverso da +1. Per ottenere questo è sufficiente specificare in modo opportuno la terza espressione (E_3) dello statement `for`. I seguenti esempi mostrano due cicli limitati, rispettivamente con passo −1 e con passo +2.

Esempio 3.17 *Stampa tutti i numeri interi compresi tra* 0 *e* 9 *in ordine decrescente:*

```
for (i = 9; i >= 0; i--)
  cout << i << ' ';
```

Esempio 3.18 *Stampa tutti i numeri pari compresi tra* −10 *e* 10*, in ordine crescente:*

```
for (int i = -10; i <= 10; i = i + 2)
  cout << i << ' ';
```

Le espressioni dello statement `for` possono essere anche espressioni più complesse di quelle viste finora. In particolare l'espressione E_2, che funge da condizione d'uscita dal ciclo, può essere un'espressione booleana composta costruita con i soliti connettivi logici (ad esempio, `i < 10 && i != j`). Si può anche utilizzare, al posto di una qualsiasi delle tre espressioni del `for`, la concatenazione di due espressioni ottenuta tramite l'operatore ',', come mostrato nel seguente esempio:

Esempio 3.19 *Stampa contemporaneamente tutti i numeri interi compresi tra* 0 *e* 9 *in ordine crescente e tra* 1 *e* 10 *in ordine decrescente:*

```
int x;
int y;
for (x = 0, y = 10; x < 10; ++x, --y)
  cout << x << ' ' << y << '\n';
```

Si osservi che è la variabile `x` *a fungere da variabile di controllo all'interno del ciclo.*

Nota. L'operatore (infisso) ',' permette di concatenare due o piu espressioni qualsiasi. Il significato di E_1 , E_2, con E_1, E_2 espressioni qualsiasi, è: valuta E_1, valuta E_2 e restituisci come risultato dell'espressione composta il valore di E_2. Ad esempio, `x = 1, y = 2, x + y` valuta le tre sottoespressioni, nell'ordine da sinistra verso destra (l'operatore ',' è associativo a sinistra) e restituisce come risultato finale dell'intera espressione il valore `3`.

Infine, tutte e tre le espressioni del `for` possono essere l'*espressione vuota* e quindi, di fatto, essere omesse (si noti che i "`;`" che separano le espressioni del `for` vanno comunque inseriti, anche se le espressioni sono mancanti). Ad esempio, il programma mostrato nell'Esempio 3.14 potrebbe essere scritto alternativamente come:

```
int i = 0;
for ( ; i < 10; ) {
  cout << i << ' ';
  i++;
}
```

In questo caso si sta di fatto utilizzando lo statement `for` come un `while`.

Come altro esempio, il programma mostrato nell'Esempio 3.17 può essere scritto equivalentemente nel seguente modo:

```
for (i = 10; i-- > 0; )
  cout << i << ' ';
```

In questo caso si è omessa la terza espressione del `for` ed inserito il decremento della variabile di controllo all'interno della seconda.

Per ultimo, il seguente frammento di programma realizza il calcolo del fattoriale di un numero intero `N`, utilizzando (una forma particolare) di `for`:

```
int Fatt, N;
cin >> N;
for (Fatt = (N ? N : 1) ; N > 1; )
  Fatt *= (--N);
```

Si tratta evidentemente di un codice particolarmente sintetico, in cui vengono sfruttate varie pecularietà del C++ (che lasciamo da scoprire ed analizzare al lettore attento). È altrettanto evidente, comunque, che la sinteticità è ottenuta a discapito della chiarezza del codice.

La possibilità di omettere le espressioni del `for` rende ancor più evidente il fatto che con il `for` del C++ è possibile scrivere cicli infiniti. In particolare, se manca l'espressione E_2 del `for` e non sono presenti interruzioni forzate del ciclo (per esempio attraverso l'utilizzo di un `break`), allora il `for` provoca sicuramente un ciclo infinito.

L'utilizzo di queste forme particolari di `for` può rendere il programma più difficile da comprendere ed è quindi, in generale, sconsigliato. Come regola generale, è sempre preferibile utilizzare il costrutto sintattico che modella in modo più naturale la struttura di controllo (iterativa) che si deve realizzare, sebbene la flessibilità del linguaggio di programmazione permetta di utilizzare indifferentemente uno qualsiasi dei costrutti offerti.

3.6 Statement `switch`

Come abbiamo già avuto modo di evidenziare, una struttura di controllo molto comune è quella del test multiplo. In particolare, è frequente la situazione in cui si deve scegliere di eseguire una fra k possibili azioni in base ad m valori distinti ($m \geq k$) che una data espressione E può assumere. Questa situazione può essere modellata tramite `if-else` annidati. Ma molti linguaggi, tra cui il C++, offrono un apposito costrutto sintattico (`case`, `switch`, ...) che permette di modellare in modo più naturale ed immediato questa forma di controllo del flusso.

Nel caso del C++ lo statement in questione è lo `switch`. Vediamo come utilizzare questo statement per realizzare la struttura di controllo di test multiplo sopra descritta, nel caso $m = k$.

Sintassi

```
switch (E) {
case C1:
   S_Seq1; break;
case C2:
   S_Seq2; break;
...
case Ck:
   S_Seqk; break;
default:
   S_Seq;
}
```

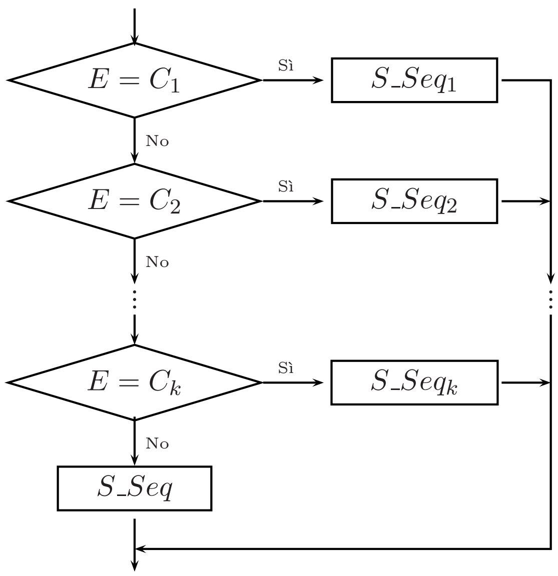
dove

E:	espressione di tipo compatibile con `int`;
C_1, C_2, ..., C_k ($k \geq 0$):	espressioni costanti di tipo compatibile con `int`;
S_Seq_1,..., S_Seq_k, S_Seq:	sequenze di statement qualsiasi.

Semantica (informale)

Valuta l'espressione E; se esiste i, con $i = 1, \ldots, k$, tale che il valore di E è uguale al valore di C_i, allora esegue la sequenza di statement S_Seq_i e quindi termina; altrimenti, esegue lo statement S_Seq, se presente, e quindi termina.

La situazione è descrivibile con un diagramma di flusso nel seguente modo:



Si tratta di un test multiplo in cui si sceglie una tra k possibili sequenze di statement S_Seq_1, ..., S_Seq_k, in base al valore di un'espressione E. La parte introdotta dalla parola chiave `default` è opzionale e viene eseguita soltanto se il valore di E non combacia con nessuna delle costanti specificate nei rami `case`.[3]

Il seguente frammento di programma mostra un semplice esempio di questo particolare utilizzo dello statement `switch`. Scopo del programma è di stampare il nome italiano dell'operatore aritmetico rappresentato dal carattere contenuto nella variabile `c`.

```
...
char c;
switch (c) {
case '+':
   cout << c << " -> addizione" << endl; break;
case '-':
   cout << c << " -> sottrazione" << endl; break;
case '*':
   cout << c << " -> moltiplicazione" << endl; break;
case '/':
   cout << c << " -> divisione" << endl; break;
default:
   cout << "Non e' un operatore aritmetico!" << endl;
}
```

In questo esempio (come spesso accade), l'espressione di controllo è semplicemente una variabile singola.

Si osservi che le costanti $C_1, \ldots, C_k$ non possono essere espressioni qualsiasi, ma devono essere necessariamente espressioni *costanti*, e cioè espressioni il cui valore è determinabile a tempo di compilazione (ad esempio `2` o `2+3`, ma non `2+x`, con `x` variabile). Inoltre i valori di $C_1, \ldots, C_k$ devono essere tutti diversi tra di loro (in caso contrario il compilatore segnala un errore). Infine, il tipo di $C_1, \ldots, C_k$ e quello di E devono essere compatibili tra loro e tutti di tipo scalare (e cioè, compatibile con `int`).

Per quanto riguarda il "corpo" di ogni alternativa `case` si osservi che si tratta di una sequenza di statement e non di un singolo statement: questo implica tra l'altro che, nel caso il corpo dell'alternativa debba contenere più statement, non è necessario racchiuderli tra parentesi graffe in modo da ottenere lo statement composto.

Esercizio 3.20 *Scrivere un programma C++ che legge da standard input un numero intero e stampa il corrispondente mese dell'anno o un opportuno messaggio di errore se il numero non è compreso tra 1 e 12.*

[3] Se presente, l'alternativa `default` deve essere unica, e può comparire ovunque nello `switch` (anche se è prassi inserirla sempre come ultima alternativa).

Approfondimento (Implementazione dello `switch`). Lo statement `switch` viene normalmente implementato in modo particolarmente efficiente su una macchina convenzionale. Senza entrare nei dettagli di un'implementazione reale, si può pensare che le costanti delle diverse alternative `case` vengano viste a livello di linguaggio Assembly come altrettante *etichette* associate al codice della sequenza di statement che segue il relativo `case`. L'esecuzione di uno `switch` allora procede valutando dapprima l'espressione di controllo dello `switch` stesso e quindi eseguendo un salto al codice che ha come etichetta il valore dell'espressione appena calcolato. Questa tecnica di implementazione spiega varie caratteristiche dello `switch` tra cui il fatto che le costanti associate ai `case` debbano essere valutabili a tempo di compilazione e debbano essere tutte diverse tra loro. Inoltre si osservi che il codice relativo alle sequenze di statement dei diversi rami `case` sarà disposto in memoria in modo consecutivo, in base all'ordine che gli statement hanno nello `switch`, e che quindi, una volta effettuato un salto alla porzione di codice individuata dall'etichetta corrispondente al valore assunto dall'espressione di controllo, l'esecuzione prosegue sul codice che segue, a meno che non sia prevista un'istruzione di salto esplicito che permette di trasferire il controllo oltre il codice dello `switch`.

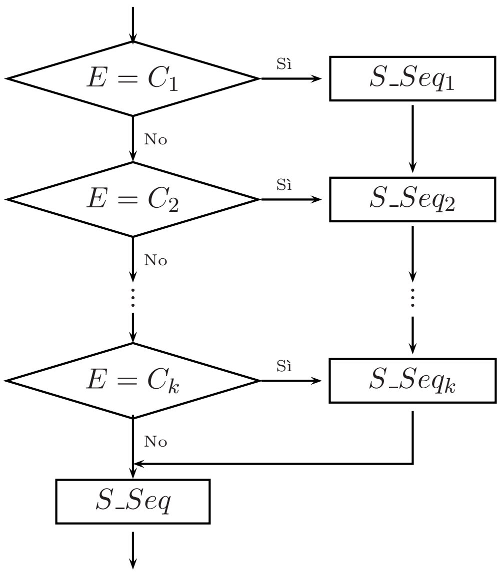
Nella forma di `switch` fin qui considerata, ogni alternativa, tranne l'ultima, termina con uno statement `break`. L'esecuzione di questo statement—su cui ritorneremo in modo più approfondito nel paragrafo successivo—provoca la terminazione immediata dello statement `switch` (e quindi la prosecuzione dell'esecuzione dallo statement successivo lo `switch`).[4]

In realtà lo statement `switch` del C++ non richiede, in generale, che un'alternativa `case` termini necessariamente con uno statement `break`: la sequenza di statement successivi alla parola chiave `case` può essere una sequenza qualsiasi (anche vuota).

Dunque, nel caso più generale (cioè senza i `break`), la semantica dello `switch` diventa: valuta l'espressione E; se esiste i, con $i = 1, \ldots, k$, tale che il valore di E è uguale al valore di C_i, allora esegue nell'ordine le sequenze di statement S_Seq_i, $S\_Seq_{i+1} \ldots, S\_Seq_k$ e quindi termina; altrimenti, esegue lo statement S_Seq, se presente, e quindi termina.

Con un diagramma di flusso la situazione è rappresentata nel modo seguente:

[4] Si noti che dopo uno statement `break` in un'alternativa `case` non ha senso che compaiano altri statement all'interno della stessa alternativa in quanto non potrebbero mai essere eseguiti.



In altri termini, l'esecuzione del corpo dello `switch` comincia, in generale, dall'alternativa `case` a cui è associato un valore uguale al valore della condizione dello `switch` e continua attraverso le successive alternative fino a che non termina lo `switch` o incontra un `break` (o un'altra istruzione che provoca forzatamente il salto fuori dallo `switch`). Si tratta dunque di una struttura di controllo decisamente diversa da quella del test multiplo descritto all'inizio del paragrafo.

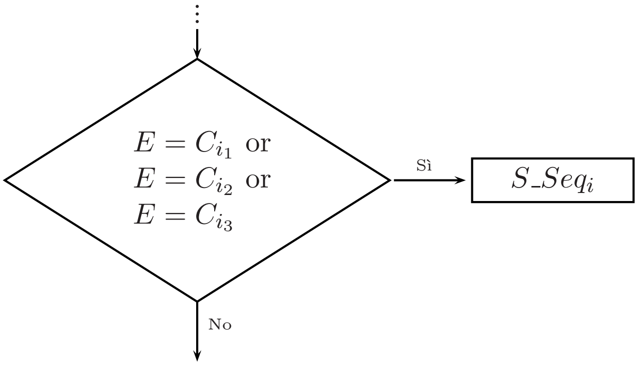
Riferendoci al frammento di programma mostrato sopra, l'assenza dell'istruzione `break` all'interno per esempio del `case '*'` provocherebbe come output

```
* -> moltiplicazione
* -> divisione
Non e' un operatore aritmetico!
```

in risposta al carattere `'*'` fornito come input che, evidentemente, non è quanto si voleva ottenere. Si noti come la mancanza dei `break` renda significativo l'ordine in cui appaiono le diverse alternative dello `switch`.

Nota. Il costrutto `switch` viene utilizzato nella maggior parte dei casi per realizzare la struttura di controllo di test multiplo considerata all'inizio del paragrafo, che richiede necessariamente l'utilizzo di un `break` alla fine di ogni ramo `case` dello `switch` (tranne l'ultimo che non necessita del `break` non essendo seguito da altre alternative `case`). Dunque l'assenza di un `break` è molto probabilmente segnale di un possibile errore e quindi si consiglia di evidenziare con un commento un'eventuale omissione voluta di un `break`.

La forma più generale dello `switch` permette facilmente di modellare anche la situazione di test multiplo in cui ci siano più valori possibili in corrispondenza a ciascuna alternativa dello `switch` (caso $m > k$). Ad esempio supponiamo che la sequenza di statement S_Seq_i, $1 \leq i \leq k$, debba essere eseguita quando l'espressione E assume uno tra tre possibili valori, C_{i_1}, C_{i_2}, C_{i_3}, e cioè, con un diagramma di flusso:



Per modellare questa situazione basta inserire nello `switch` tre alternative `case` consecutive nel modo seguente:

`switch (`E`) {`
...
`case` C_{i_1}`:`
`case` C_{i_2}`:`
`case` C_{i_3}`:`
   S_Seq_i`; break;`
...
`}`

Si noti che i primi due `case` sono alternative dello `switch` con la parte relativa alla sequenza di statement vuota (in particolare, senza `break`) e quindi la loro esecuzione comporta semplicemente la prosecuzione sullo statement dell'alternativa successiva.

Esempio 3.21 (Conta numero vocali)
Problema. *Scrivere un programma che legge da standard input una sequenza di caratteri terminata da un punto, determina il numero di vocali (maiuscole o minuscole) presenti nella sequenza e quindi stampa il numero di vocali presenti per ciascuna delle 5 vocali (si veda per confronto il programma dell'Esempio 3.8).*

Programma:

```
#include <iostream>
using namespace std;
int main() {
  char c;
  int n_a = 0, n_e = 0, n_i = 0, n_o = 0, n_u = 0;
```

```
  cout << "Inserisci sequenza di caratteri terminata da ."
       << endl;
  cin >> c;   //legge un carattere e lo assegna a c
  while (c != '.'){
    switch(c)  {
    case 'a': case 'A':
      n_a++; break;
    case 'e': case 'E':
      n_e++; break;
    case 'i': case 'I':
      n_i++; break;
    case 'o': case 'O':
      n_o++; break;
    case 'u': case 'U':
      n_u++;
    }
    cin >> c;
  }
  cout << "La sequenza data contiene " << endl;
  cout << n_a << " vocali a" << endl;
  cout << n_e << " vocali e" << endl;
  cout << n_i << " vocali i" << endl;
  cout << n_o << " vocali o" << endl;
  cout << n_u << " vocali u" << endl;
  return 0;
}
```

Approfondimento (`switch` vs `if`). Uno statement `switch` può essere facilmente sostituito da una serie di statement `if-else` annidati. Ad esempio, la situazione descritta dallo statement `switch`

$$
\begin{array}{l}
\texttt{switch (}E\texttt{) \{} \\
\texttt{case } C_1\texttt{: case } C_2\texttt{: case } C_3\texttt{:} \\
\quad S\_Seq_1\texttt{; break;} \\
\texttt{case } C_4\texttt{:} \\
\quad S\_Seq_2\texttt{; break;} \\
\texttt{default:} \\
\quad S\_Seq\texttt{;} \\
\texttt{\}}
\end{array}
$$

può essere realizzata in modo equivalente (almeno per quanto riguarda il comportamento esterno) tramite `if-else` annidati nel modo seguente:

$$
\begin{array}{l}
\texttt{if (}E\texttt{ == }C_1\texttt{ || }E\texttt{ == }C_2\texttt{ || }E\texttt{ == }C_3\texttt{) }\{S\_Seq_1\} \\
\texttt{else if (}E\texttt{ == }C_4\texttt{) }\{S\_Seq_2\} \\
\texttt{else }\{S\_Seq\}
\end{array}
$$

Il codice scritto utilizzando il costrutto `switch` presenta alcuni vantaggi rispetto al codice che utilizza `if-else` annidati:

- risulta, in generale, più leggibile;
- può essere eseguito, in generale, in modo più efficiente grazie alla tecnica di implementazione dello `switch` che richiede sempre un solo test per individuare l'alternativa da selezionare (per contro, con gli if-else annidati, se l'alternativa è l'n-esima verranno effettuati n test prima di poterla selezionare).

Si osservi che il test usato per selezionare un'alternativa in uno statement `switch` è necessariamente un test di uguaglianza (il valore di E è uguale al valore di C_i?). Questo rende molto più problematico realizzare tramite `switch` un test multiplo in cui le condizioni di selezione delle diverse alternative siano in generale delle disuguaglianze. Ad esempio, una situazione del tipo *"se $C_1 \leq E < C_2$ allora esegui S_1; se $C_2 \leq E < C_3$ allora esegui S_2; altrimenti esegui S_3"* viene realizzata in modo molto più naturale tramite `if-else` annidati (si veda ad esempio il programma di conversione di voti in giudizi mostrato nell'Esempio 3.4).

Nota. Si osservi che è possibile ottenere il comportamento di un generico `if-else`

```
if (E) S1
else S2
```

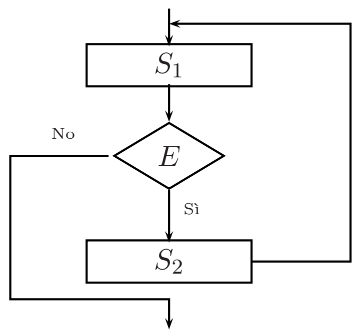
utilizzando uno statement `switch` nel modo seguente:

```
switch (E) {
   case true: S1; break;
   case false: S2: break;
}
```

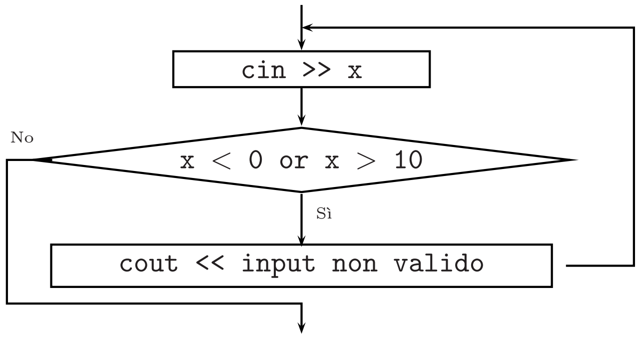
È chiaro che si tratta di un "trucco" di programmazione che sicuramente non contribuisce alla chiarezza del programma e che quindi andrebbe di norma evitato.

3.7 Statement `break`

Finora abbiamo visto strutture di controllo per l'iterazione non limitata in cui la condizione d'uscita dal ciclo è posta o all'inizio o alla fine del ciclo, e che possono essere realizzate in modo naturale rispettivamente tramite gli statement `while` e `do-while`. Nella pratica, però, sono frequenti anche casi in cui la condizione d'uscita dal ciclo è posta in mezzo al ciclo, cioè preceduta e seguita da altre azioni, come illustrato nel seguente diagramma di flusso:



Una simile struttura di controllo può ovviamente essere realizzata con gli statement `while` e `do-while`, ma, come vedremo, in modo non del tutto naturale. Si consideri ad esempio la sequenza di istruzioni descritta dal seguente diagramma di flusso:



che rappresenta una tipica situazione di lettura di un dato di input con controllo di validità del dato stesso e ripetizione della lettura in caso di dato non valido (preceduta da un opportuno messaggio d'errore). Questo schema può essere realizzato ad esempio utilizzando un costrutto `do-while` nel modo seguente:

```
int x;
do {
  cin >> x;
  if (x < 0 || x > 10)
    cout << "input non valido";
}
while (x < 0 || x > 10);
```

È evidente che questo codice realizza una struttura di controllo diversa (e senz'altro più complicata) rispetto a quella indicata sopra anche se il comportamento esterno delle due è lo stesso (lasciamo al lettore come esercizio la descrizione di questa struttura di controllo tramite un diagramma di flusso).

In casi come questo può essere comodo avere a disposizione uno statement che permetta di "saltar fuori" da un ciclo in un punto qualsiasi del ciclo stesso. Il C++ offre una simile possibilità attraverso lo statement `break`.

La semantica dell'istruzione `break` è quella del salto: l'esecuzione di un `break` provoca un salto al primo statement successivo al costrutto che lo contiene. Come costrutti contenenti il `break` si considerano soltanto quelli iterativi `while`, `do-while`, `for` e lo `switch` (ma non lo statement `if` o lo statement composto). L'utilizzo di un `break` all'interno di questi costrutti causa l'immediata uscita dal costrutto stesso, mentre il suo utilizzo all'esterno di questi costrutti (ad esempio all'interno di uno statement `if` non contenuto a sua volta all'interno di un ciclo o di uno `switch`) viene segnalato come errore dal compilatore.

Utilizzando il `break` è possibile riscrivere il codice dell'esempio mostrato sopra in modo che rispecchi in modo più preciso la struttura di controllo desiderata:

```
int x;
do {
  cin >> x;
  if (x < 0 || x > 10) cout << "input non valido";
  else break;
}
while (true);
```

Si osservi che il `break` comporta l'uscita dal costrutto `do-while`, ignorando il fatto che il `break` appare all'interno di un `if-else`. Si osservi anche che nel caso in cui il `break` sia all'interno di più costrutti iterativi o `switch` annidati la sua esecuzione provoca l'uscita soltanto dal costrutto che lo contiene direttamente e non da quelli più esterni.

L'istruzione `break` (così come tutte le altre istruzioni di salto, quali `continue`, `goto`, `return`) vanno contro i principi della *programmazione strutturata* secondo cui ogni costrutto, ed in particolare quelli iterativi, devono avere un unico punto di ingresso ed un unico punto d'uscita. Infatti il `break` permette di uscire da un costrutto, ad esempio da un `while`, in un punto qualsiasi, aggiungendo quindi un punto d'uscita a quello che già il costrutto che lo contiene prevede. Questa situazione è illustrata dal seguente esempio.

Esempio 3.22 (Conta il numero di lettere 'a')
Problema. *Leggere da standard input una sequenza di caratteri (di lunghezza massima 32) terminata da spazio o da "a capo" e determinare il numero di lettere 'a' presenti. Scrivere quindi il risultato su standard output. Nel caso in cui venga raggiunta la lunghezza massima della sequenza stampare anche un opportuno messaggio d'avviso.*

Programma*:*

```
#include <iostream>
using namespace std;
int main() {
  int i = 0, num_a = 0;
  cout << "Inserisci una sequenza di caratteri "
       << " terminata da 'spazio' o da 'a capo'" << endl;
  char c = cin.get();
  ++i;
  while (c != ' ' && c != '\n') {
    if (i > 32) {
```

```
      cout << "Attenzione: la sequenza e' stata troncata!"
       << endl;
      break;
    }
    if (c == 'a')
      ++num_a;
    c = cin.get();
    ++i;
  }
  cout << "Il numero di caratteri 'a' e' " << num_a << endl;
  return 0;
}
```

Un utilizzo indiscriminato del `break` può portare a programmi poco strutturati e pertanto più difficili da capire e da dimostrare corretti. L'uso di tale statement dovrebbe essere, in generale, limitato ai soli casi in cui è evidente un effettivo miglioramento nella leggibilità del programma, come mostrato, ad esempio, nelle situazioni presentate all'inizio del paragrafo.

Esercizio 3.23 *Riscrivere il programma dell'Esempio 3.22 senza utilizzare l'istruzione* `break`. *(SUGG.: aggiungere la condizione* `i <= 32` *all'interno dell'espressione booleana di controllo del ciclo* `while` *e portare il test per determinare se la sequenza è stata troncata dopo lo statement* `while`*).*

Capitolo 4

Strutture dati

4.1 Introduzione

Nei capitoli precedenti abbiamo considerato soltanto tipi primitivi semplici, precisamente i tipi `int`, `float`, `char` e `bool` del C++. Ricordiamo che il tipo di una variabile indica l'insieme dei possibili valori che essa può assumere e l'insieme delle possibili operazioni applicabili a tali valori. Nel caso dei tipi semplici, il valore corrente di una variabile è un valore "atomico", non ulteriormente scomponibile.

In C++ (come nella maggior parte dei linguaggi di programmazione) sono presenti anche tipi non semplici, detti *tipi strutturati*, i cui valori sono costituiti ciascuno da un *insieme di valori*, a loro volta di tipo semplice o strutturato, omogenei o disomogenei, organizzati tra loro secondo opportuni criteri.

In generale, un tipo strutturato è caratterizzata da:

- il tipo, ed eventualmente il numero, dei suoi componenti;
- un criterio di aggregazione dei suoi componenti;
- l'insieme delle possibili operazioni applicabili all'intera struttura, tra cui una o più operazioni di selezione dei singoli elementi.

I linguaggi di programmazione permettono, solitamente, di definire tipi strutturati nuovi a partire dai tipi (già esistenti) degli elementi che li costituiscono tramite opportuni *costruttori di tipo*. In particolare, il C++ offre i seguenti costruttori per la creazione di tipi strutturati:

- array (che in C++ non ha una parola chiave specifica, ma è implicito nella forma sintattica della dichiarazione)
- `struct`
- `union`

- `class`.

Costruttori diversi realizzano forme diverse di aggregazione dei dati e quindi tipi strutturati diversi, con proprietà diverse ed un diverso insieme di operazioni ad essi applicabili. Ad esempio, due tipi strutturati costituiti entrambi da tre elementi di tipo intero, ma definiti uno tramite il costruttore array e l'altro tramite il costruttore `struct` sono tipi distinti. Inoltre, tipi strutturati definiti tramite lo stesso costruttore possono lo stesso differire tra loro a causa del tipo e/o del numero dei componenti. Ad esempio, due tipi strutturati realizzati entrambi tramite il costruttore array, ma uno composto da tre elementi interi e l'altro da tre elementi reali (`float` in C++) sono tipi distinti.

Analizzeremo gli array nel paragrafo 4.3. Il costruttore `struct` sarà invece trattato nel paragrafo 4.6, mentre il costruttore `class` sarà introdotto ed analizzato in dettaglio nella II Parte del testo. Non tratterremo invece il costruttore `union` per il quale rimandiamo ad un manuale di linguaggio C, come ad esempio il classico [13].

4.2 Strutture dati astratte e concrete

Chiamiamo *struttura dati astratta* una struttura dati definita (soltanto) in base alle sue proprietà ed alle operazioni applicabili ad essa. Viceversa, l'implementazione, ossia quale *struttura dati concreta* è utilizzata per la sua realizzazione, non è visibile.

Esempi di tipiche strutture dati astratte sono:

- matrici (a k dimensioni)
- stringhe
- tabelle
- pile
- code
- insiemi
- alberi
- grafi.

Ciascuna di queste strutture dati ha precise caratteristiche che la distinguono dalle altre. Ad esempio, una *pila di interi* è costituita da una sequenza (anche vuota) di elementi di tipo intero su cui è possibile effettuare (soltanto) le operazioni di inserimento o di estrazione dell'elemento che si trova su uno dei due estremi della sequenza (la "testa" della pila). Una *coda di interi*, invece, è analoga alla pila di interi, ma le operazioni di inserimento o di estrazione di un elemento si effettuano una su un estremo della sequenza e l'altra sull'altro estremo.

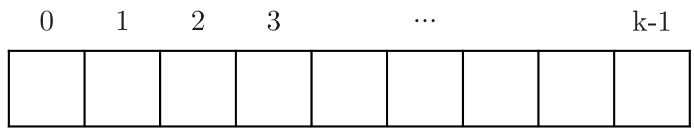
Una stessa struttura dati astratta può essere implementata utilizzando diverse strutture dati concrete. Ad esempio, una pila può essere realizzata tramite un array ed una variabile intera che indica la posizione della "testa" della pila, oppure da una lista di elementi concatenati tramite puntatori a partire dall'elemento di testa. Viceversa, una stessa struttura dati concreta può essere utilizzata per implementare diverse strutture dati astratte. Per esempio gli array possono essere usati per realizzare matrici, stringhe, tabelle, pile, ecc..

In questo testo tratteremo soltanto alcune delle strutture dati (astratte) sopra elencate. Precisamente, in questo capitolo, mostreremo la definizione ed una possibile implementazione delle strutture dati matrice, stringa e tabella. Alcune altre strutture dati, tra cui le pile, saranno invece trattate nella II Parte del testo, dopo aver introdotto il costruttore `class`. Inoltre ritorneremo sulla nozione di struttura dati astratta nel capitolo **??**. Per una trattazione più completa ed approfondita delle diverse strutture dati e dei relativi algoritmi per la loro gestione rimandiamo, ad esempio, al testo [2].

4.3 Array

[**Array**] Un oggetto di tipo *array* (o, più semplicemente, un *array*) è una sequenza di k elementi, $k \geq 0$, tutti di un certo tipo t, individuati singolarmente da un numero intero compreso tra 0 e $k-1$ detto *indice*. k costituisce la *dimensione*, o *capacità*, dell'array.

Una tipica rappresentazione grafica per un array di dimensione k è la seguente:



Nei paragrafi successivi esamineremo in dettaglio le possibilità offerte dal C++ per la creazione e manipolazione di array *statici*, ovvero array allocati a tempo di compilazione e, quindi, di dimensioni completamente note a priori. Affronteremo invece la creazione di array dinamici in un capitolo successivo, dopo aver introdotto le nozioni di puntatore ed allocazione dinamica della memoria.

4.3.1 Dichiarazione di array

La creazione di un array (statico) in un programma C++ avviene tramite un'opportuna dichiarazione.

[**Dichiarazione di array**] La dichiarazione

$$t \texttt{ A[}k\texttt{];}$$

dove:

- `A` è un identificatore,
- t è un tipo qualsiasi (semplice o a sua volta strutturato),
- k è un'espressione intera *costante* ≥ 0,

è una *dichiarazione di array* in C++ (forma base). Il significato di questa dichiarazione è la creazione di una variabile di nome `A`, di tipo strutturato *array di* t (ovvero t `[ ]`), costituita da k elementi di tipo t. Più sinteticamente, diremo che la dichiarazione t `A[`k`]` crea un array `A` di k elementi di tipo t.

Come per le variabili semplici, la creazione di un array comporta l'allocazione dello spazio di memoria necessario a contenere gli elementi dell'array, ovvero, nel caso della dichiarazione t `A[`k`]`, l'allocazione di k celle di memoria consecutive, ciascuna di dimensioni adeguate a contenere un valore di tipo t.

Si osservi che nella dichiarazione t `A[`k`]` il tipo della variabile `A` (array di t) è definito direttamente nella dichiarazione stessa. Si osservi anche la sintassi piuttosto "infelice" della dichiarazione di array in C++, in cui il nome della variabile `A` appare in mezzo a quello del suo tipo (t `[ ]`) e in quest'ultimo non è presente alcuna parola chiave che indichi la presenza del costruttore di array (come invece accade, ad esempio, in Pascal). In un capitolo successivo vedremo come in C++ sia anche possibile dichiarare il tipo array a parte, associandogli un nome, da usare poi nella dichiarazione di array.

Esempio 4.1 (Dichiarazioni di array)

`int V1[10];`	*variabile di nome* `V1`*,* *di tipo array di interi, di dimensione 10*
`int V2[20];`	*variabile di nome* `V2`*,* *di tipo array di interi, di dimensione 20* *(n.b. stesso tipo dell'array* `V1`*)*
`char S[32];`	*variabile di nome* `S`*,* *di tipo array di caratteri, di dimensione 32*

Nel seguito spesso scriveremo, per comodità, "l'array di interi `A`" invece che "la variabile di nome `A` di tipo array di interi".

In una dichiarazione di array, come nella dichiarazione di variabili semplici, è possibile specificare anche un eventuale valore iniziale che, in questo caso, sarà costituito da un insieme di valori da assegnare agli elementi dell'array.

[**Dichiarazione di array con inizializzazione**] La dichiarazione

$$t\ \texttt{A[}k\texttt{]} = \{i_1, i_2, \ldots, i_m\};$$

dove i_1,i_2,...,i_m sono espressioni di tipo t, è una *dichiarazione di array con inizializzazione* (forma base). $\{i_1, i_2, \ldots, i_m\}$ è detta *lista di inizializzatori.* Il significato di questa dichiarazione è la creazione di un array di dimensione k, con l'inizializzazione dei suoi primi m elementi, rispettivamente, con i valori di i_1,i_2,...,i_m.

Esempio 4.2 (Dichiarazione di array con inizializzazione)

```
char vocali[5] = {'a','e','i','o','u'};
```

array di 5 caratteri, di nome `vocali`*, inizializzati nel modo seguente:*

0	1	2	3	4
'a'	'e'	'i'	'o'	'u'

Se il numero degli elementi specificati nella lista di inizializzatori è maggiore della capacità dell'array, cioè $m > k$, si verifica un errore rilevato in fase di compilazione; se invece è $m < k$ allora vengono inizializzati i primi m elementi dell'array, mentre i restanti $k - m$ elementi vengono riempiti con 0, indipendentemente dal loro tipo.

Esempio 4.3 (Dichiarazione di array con inizializzazione parziale)

```
int V[10] = {2,-1,1};
```

array di 10 interi di cui i primi 3 inizializzati con i valori `2`*,* `-1` *e* `1`*, rispettivamente, ed i rimanenti elementi tutti inizializzati con* `0`*.*

Si osservi che, come per le variabili di tipo semplice, nel caso in cui non si specifichi alcun valore iniziale, gli elementi di un array avranno un valore *indeterminato.*

La presenza di una lista di inizializzatori permette di dichiarare l'array senza dover necessariamente indicare in modo esplicito la sua dimensione. In questo caso, infatti, la dimensione dell'array è "dedotta" automaticamente dal numero di inizializzatori presenti.

Esempio 4.4 (Dichiarazione di array con dimensione implicita)

```
float F[] = {1.0,-1.5,+1.5};
```

array di 3 elementi di tipo `float`*.*

Si noti che l'assegnazione di valori agli elementi di un'array tramite una lista di inizializzatori è possibile soltanto all'interno della dichiarazione dell'array stesso. Non è quindi possibile scrivere, ad esempio:

```
int A[5];
A = {1, 2, 3, 4, 5};
```

che comporterebbe una segnalazione di errore da parte del compilatore.

Infine si osservi che, come per le variabili di tipo semplice, l'elaborazione di una dichiarazione di array comporta, in generale, l'allocazione dell'area di memoria per l'intera struttura dati e l'associazione dell'indirizzo iniziale di tale area al nome dell'array.

4.3.2 Operazione di selezione

Operazione fondamentale per poter utilizzare un array, e più in generale una qualsiasi struttura dati, è quella di poter riferirsi, e quindi accedere, sia in lettura che in scrittura, ad un suo singolo elemento. Le caratteristiche di questa operazione, detta *operazione di selezione*, differiscono da struttura a struttura. Nel caso dell'array l'operazione di selezione di un elemento avviene semplicemente tramite l'indice dell'elemento stesso, ovvero il numero intero associato univocamente ad ogni elemento dell'array. Si tratta di un *accesso diretto*, il cui tempo d'accesso non dipende dalla posizione relativa dell'elemento all'interno dell'array, ma è costante (in contrasto con l'*accesso sequenziale*). L'implementazione dell'array fornita dal linguaggio dovrà garantire questa importante proprieà.

In C++ la selezione di un elemento di un array avviene tramite un'opportuna espressione.

[**Espressione con indice**] Dato l'array

$$t\ \texttt{A[}k\texttt{];}$$

con t tipo qualsiasi, l'espressione

$$\texttt{A[}e\texttt{]}$$

dove e è un'espressione di tipo compatibile con `int`, è detta *espressione con indice*. Il suo significato è quello di selezionare l'elemento dell'array `A` di indice il valore dell'espressione e (si noti che prima si valuta e e quindi si usa il valore risultate per selezionare l'elemento dell'array). Il tipo dell'espressione `A[`e`]` è t, e cioè il tipo degli elementi di `A`.

Esempio 4.5 (Selezione di elementi di un array) *Dato l'array di interi*

```
int V[10] = {5,11,20,17,8,4,9,13,5,12};
```

0	1	2	3	4	5	6	7	8	9
5	11	20	17	8	4	9	13	5	12

e cioè, graficamente,
le seguenti sono operazioni corrette di accesso in lettura a singoli elementi dell'array:

```
cout << V[0];    // stampa il valore 5
int i = 2;
cout << V[i];    // stampa il valore 20
cout << V[i+1];  // stampa il valore 17
cout << V[i]+1;  // stampa il valore 21
cout << V[0];    // stampa il valore 5
cout << V[9];    // stampa il valore 12
```

Negli esempi di sopra l'espressione `V[...]` denota il *valore* dell'elemento di `V` selezionato. L'espressione `V[...]` può comparire però anche nella parte sinistra di un'istruzione di assegnamento ed in questo caso denota la *locazione* dell'elemento selezionato, non il suo valore. L'istruzione `V[i] =` *expr* permette perciò di assegnare il valore dell'espressione *expr* alla locazione di memoria individuata dall'espressione `V[i]`, e cioè all'elemento dell'array `V` di indice i.

Esempio 4.6 (Accesso in lettura e scrittura ad elementi di un array) *Sia* `V` *l'array dell'Esempio 4.5.*

```
int i = 2;
V[1] = 7;
V[2] = V[1] * V[0];
V[i] = V[i] + 1;    // equivalente a I[i]++
V[i] = V[i+1];
int W[5] = {7, 3, 1, 4, 2}
cout << V[W[i]];    // stampa il valore 7
```

Il singolo elemento di un array `A` con elementi di tipo t, individuato tramite l'operazione di selezione `A[`e`]`, è a tutti gli effetti una variabile di tipo t. Dunque è possibile applicare ad esso tutte le operazioni previste dal tipo t. Ad esempio, se `V` è l'array di interi dell'Esempio 4.5 possiamo scrivere ad esempio

```
//resto della divisione intera tra V[1] e 2
int x = V[1] % 2;
```

oppure

```
//moltiplicazione tra interi
V[2] = V[1] * V[0];
```

L'array è una collezione limitata di valori di un dato tipo t, il cui numero massimo è stabilito dalla costante k specificata nella dichiarazione dell'array stesso. Di conseguenza, gli indici di un array possono assumere valori in un intervallo ben definito, che in C++ è costituito dagli interi tra 0 e $k - 1$. Qualsiasi riferimento ad un elemento dell'array con un indice di valore minore di 0 o maggiore o uguale a k è da considerarsi un errore, che produce un risultato indefinito.

In C++ però è stata fatta la scelta, essenzialmente per motivi di efficienza, di non controllare, nè a compile-time, nè a run-time, la correttezza (dei valori) degli indici di un array. Dunque in C++ è possibile scrivere ad esempio

```
int V[10];
V[11] = 3;
```

senza che il compilatore dia nessuna segnalazione di errore. L'esecuzione dell'operazione di selezione sull'array comporterà un accesso ad una locazione di memoria che non fa parte dello spazio di memoria allocato per l'array e quindi il risultato del programma diventa del tutto imprevedibile (ad esempio, potrebbe andare a scrivere il valore da assegnare all'elemento dell'array nella locazione di memoria di un'altra variabile del programma, oppure, se l'indice usato è molto più grande del massimo previsto, provocare una violazione di indirizzamento della memoria principale con conseguente terminazione forzata dell'esecuzione del programma). Si tratta quindi di errori, in generale, molto insidiosi e difficili da individuare. È dunque importante prestare molta attenzione alla correttezza dei valori attribuiti agli indici di un array[1].

Memorizzazione di un array

La creazione di un array `A` di k elementi di tipo t (e cioè, t `A[`k`]`) comporta l'allocazione di k celle di memoria di tipo t, consecutive, in cui memorizzare nell'ordine i k elementi dell'array. Per *cella di memoria* di tipo t si intende la sequenza di m byte consecutivi utilizzati per memorizzare un dato di tipo t. Ad esempio, per `int`, sarà, tipicamente, $m = 4$, mentre per `char` sarà $m = 1$. L'*indirizzo base* b dell'area di memoria dedicata all'array viene stabilito dal sistema al momento dell'allocazione dell'array stesso e l'utente non ne ha nessun controllo.

Esempio 4.7 (Allocazione di un array) *L'array*

```
int V[6];
```

[1] In C++ è possibile usare in alternativa al costruttore primitivo di array la classe predefinita `vector` della libreria standard o definire una propria classe che preveda anche il controllo a run-time di correttezza degli indici.

comporta l'allocazione di 6 *celle di tipo intero e quindi occupa* $6 \times 4 = 24$ *byte. Se assumiamo* $b = 15216$ *e* $m = 4$ *la situazione della memoria dedicata all'array* `V` *è la seguente:*

$V[0]$	$b = 15216$
$V[1]$	15220
$V[2]$	15224
$V[3]$	15228
$V[4]$	15232
$V[5]$	15236

Si noti che, poichè gli indirizzi di memoria sono relativi al singolo byte, la cella successiva alla prima avrà indirizzo (in byte) $b + 4$ (nell'esempio, $15216 + 4$).

Con questa implementazione è facile garantire l'*accesso diretto* ai singoli elementi dell'array, cioè che il tempo di accesso al singolo elemento sia indipendente dalla sua posizione. Dato un array t `A[`k`]` con indirizzo base b e dimensione (in byte) di un singolo elemento d ($d =$ `sizeof`(t)), l'*indirizzo di memoria* del generico elemento `A[`i`]`, per $i = 0, \ldots, n-1$, è dato da

$$b + i * d.$$

Ad esempio, se `V` è l'array considerato nell'Esempio refes:allocazione array, l'indirizzo di `V[3]` è: $15216 + 3 * 4 = 15228$.

In C++ la memorizzazione concreta di un array non è esattamente quella sopra considerata. In C++, infatti, per un array t `A[`k`]` viene allocato, oltre allo spazio di memoria costituito dalle k celle di tipo t consecutive destinate a contenere gli elementi dell'array, anche una cella di memoria che conterrà l'indirizzo di base dell'array, cioè è un *puntatore a* t (si veda il Capitolo **??**). Tutti gli accessi agli elementi dell'array avverranno in modo indiretto (ma sempre in tempo costante) attraverso la cella puntatore. La situazione è rappresentata in modo grafico nella Figura 4.1.

Nel seguito comunque potremo, in linea di massima, utilizzare gli array del C++ senza doverne necessariamente conoscere l'implementazione concreta. L'unica eccezione sarà costituita dal meccanismo di passaggio parametri (vedi par. 5.3) in cui sarà necessario prendere in considerazione le specificità dell'implementazione degli array in C++.

4.3.3 Operazioni su array

Oltre alle operazioni sui singoli elementi di un array, è, in linea di principio, possibile considerare anche operazioni che si applicano ad un'intero array (e, più in generale, ad un'intera struttura dati). Ad esempio, la stampa di un array, l'assegnamento o il confronto tra due array, ecc.

In C++, in generale, non sono previste operazioni primitive su interi array. È pertanto necessario realizzarle a programma mediante cicli che

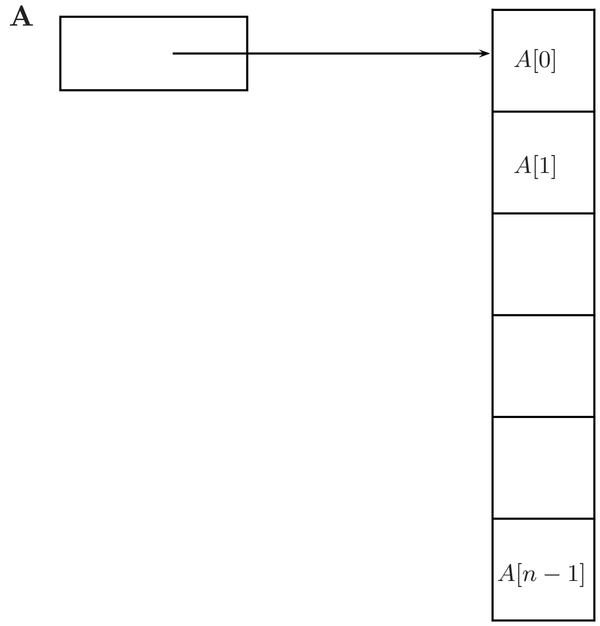


Figura 4.1: Rappresentazione grafica dell'allocazione in memoria di un array monodimensionale.

permettano di iterare su tutti gli elementi (singoli) degli array considerati. Vediamone alcuni esempi.

Esempio 4.8 (Assegnamento tra array) *Dati i due array*

```
int A[12] = {34,2,6,41,3,23,7,9,47,25,1,15};
int B[12];
```

l'assegnamento di (tutti gli elementi di) `A` *a* `B` *si può realizzare nel modo seguente:*

```
for (int i = 0; i < 12; i++)
  B[i] = A[i];
```

Esempio 4.9 (Confronto tra due array) *Dati i due array* `A` *e* `B` *dell'Esempio 4.8, per verificare se i due array sono uguali (cioè hanno tutti gli elementi ordinatamente uguali) e quindi stampare un opportuno messaggio, si può procedere nel modo seguente:*

```
bool array_uguali =  true;
for (int i = 0; i < 12; i++)
  if (B[i] != A[i]) {
    array_uguali = false;
```

```
      break;
    }
  if (array_uguali)
    cout << ‘‘Gli array sono uguali.’’
  else
    cout << ‘‘Gli array sono diversi.’’
```

Si noti che se si scrive, ad esempio, `A = B` oppure `A == B`, con `A` e `B` array, il compilatore C++ non segnala errore, e quindi esegue le operazioni indicate ma con un significato diverso da quello che esse hanno solitamente. Di fatto, le operazioni si applicano sull'indirizzo iniziale dell'area di memoria contenente gli elementi dell'array, non sugli elementi stessi. Ad esempio, `A = B` significa che l'indirizzo iniziale di `B` diventa anche l'indirizzo iniziale di `A` e quindi che da lì in poi `A` e `B` coincidono (nel senso che condividono gli stessi elementi).

Esempio 4.10 (Input/output di array) *Il seguente codice C++ permette di leggere (da standard input) e di stampare (su standard output) tutti gli elementi dell'array* `A` *dichiarato nell'Esempio 4.8.*

```
// Operazione di input
for (int i = 0; i < 12; i++)
  cin >> A[i];

// Operazione di output (stampa)
for (int i = 0; i < 12; i++)
  cout << A[i];
```

Analogamente a quanto detto sopra per le operazioni di assegnamento e confronto, anche le operazioni di input e output predefinite (operatori `>>` e `<<`) applicate ad un intero array non funzionano correttamente. Ad esempio, l'istruzione `cout << A`, con `A` array, viene eseguita, ma il suo significato è quello di stampare l'indirizzo iniziale dell'area di memoria contenente gli elementi dell'array e non gli elementi stessi (a questo, come vedremo in un prossimo paragrafo, fanno eccezione gli array di caratteri usati per realizzare stringhe, che possono essere letti e stampati come un tutt'uno con gli operatori `>>` e `<<`).

Vediamo ora un esempio più completo di operazioni su array.

Esempio 4.11 (Minimo e massimo di una sequenza di interi)

Problema. Scrivere un programma che legga da standard input una sequenza di 10 interi e ne calcoli e visualizzi su standard output il valore minimo ed il valore massimo.

Input: una sequenza di 10 interi.
Output: i due valori interi calcolati (minimo e massimo della sequenza).
Strutture dati: un array di interi, di nome `lista`, in cui memorizzare la sequenza di interi letta da standard input.

Programma:

```
#include <iostream>
#include <climits>

using namespace std;

int main() {
  const int dim = 10;
  int lista[dim];

  for (int i = 0; i < dim; i++) {
    cout << "Inserisci l'elemento numero " << i+1 << ": ";
    cin >> lista[i];
  }

  // Stampa gli elementi dell'array 'lista'.
  cout << "L'array e' costituito dai seguenti elementi: "
       << endl;
  for (int i = 0; i < dim; i++)
    cout << lista[i] << "  ";
  cout << endl;

  // Trova il minimo.
  // La variabile 'valore_min' contiene il valore piu'
  // piccolo trovato fino a quel punto; inizializzata
  // con il massimo intero
  int valore_min = INT_MAX;
  for (int i = 0; i < dim; i++)
    if (lista[i] < valore_min)
      valore_min = lista[i];
  cout << endl << "Valore minimo: " << valore_min << endl;

  // Trova il massimo.
  // La variabile 'valore_max' contiene il valore piu'
  // piccolo trovato fino a quel punto; inizializzata
  // con il minimo intero
  int valore_max = INT_MIN;
  for (int i = 0; i < 10; i++)
    if (lista[i] > valore_max)
```

```
        valore_max = lista[i];
    cout << endl << "Valore massimo: " << valore_max << endl;

    return 0;
}
```

4.3.4 Dimensione array

La dimensione di un array deve essere fissata a priori e non può essere modificata successivamente.

Nel caso di una sequenza di lunghezza non nota di dati che si ricevono dall'esterno e che devono essere memorizzati in un array è necessario fissare una lunghezza massima e dichiarare la dimensione dell'array uguale alla lunghezza massima. È quindi molto probabile che la lunghezza effettiva della sequenza di valori che si ricevono dall'esterno sia minore della dimensione dell'array. Il numero n degli elementi significativi presenti nell'array sarà in ogni istante minore o uguale della dimensione d dell'array stesso, lasciando indefiniti gli ultimi $d - n$ elementi dell'array.

In questi casi, dunque, la dimensione dell'array rappresenta una stima per eccesso della lunghezza massima della sequenza di input e dipenderà, in generale, dalla specifica applicazione. Ad esempio, se si vuole realizzare un programma in grado di memorizzare in un array i voti in trentesimi ottenuti negli esami di un corso di laurea, per poi applicarvi diverse elaborazioni (ad esempio l'individuazione del voto minimo), allora la dimensione dell'array potrebbe essere ragionevolmente fissata a 50 che rappresenta una stima per eccesso realistica del numero massimo di voti distinti che si dovranno trattare.

Vediamo ora un esempio completo in cui la dimensione della sequenza in input non è nota a priori e la terminazione dell'input è determinato dalla presenza di un valore particolare.

Esempio 4.12 (Sequenza palindroma):

Problema. Scrivere un programma in grado di leggere una sequenza di caratteri (di lunghezza massima 100), terminata dal un carattere '.' e controllare se la sequenza è palindroma. Ad esempio, le sequenze `radar.` e `abba.` sono palindrome, mentre la sequenza `abcabc.` non lo è (n.b., il carattere `'.'` non è considerato parte della sequenza di caratteri da controllare, ma serve soltanto come delimitatore della sequenza, dato che la sua lunghezza non è specificata a priori).

Input: una sequenza di caratteri.
Output: un messaggio di risposta ("sequenza palindroma" oppure "sequenza non palindroma").
Strutture dati: un array di caratteri in cui memorizzare la sequenza di caratteri letta dallo standard input.

Programma:

```
#include <iostream>
using namespace std;
int main() {
  const int dim = 100;
  char sequenza[dim], c;
  int i = 0;
  cout << "Dai sequenza di caratteri (terminata da '.') "
       << endl;
  cin >> c;
  while (c != '.' && i < dim)
     {sequenza[i] = c;
      i++;
      cin >> c;
     }
  // numero di caratteri componenti la sequenza letta
  int n = i;
  i = 0;
  while (i < n/2 && sequenza[i] == sequenza[n - i - 1])
      i++;
  if (i == n/2)
    cout << "sequenza palindroma" << endl;
  else
    cout << "sequenza non palindroma" << endl;
  return 0;
}
```

Data in input, ad esempio, la sequenza `radar.` il programma termina producendo il messaggio `sequenza palindroma`.

4.3.5 Array bidimensionali

Un *array bidimensionale* è un array di m elementi ciascuno dei quali è, a sua volta, un array di n elementi di tipo t (m, $n \geq 0$ costanti).

In C++ un array bidimensionale (statico) di $m \times n$ elementi di tipo t può essere dichiarato nel modo seguente:

$$t\ \mathtt{B[}m\mathtt{][}n\mathtt{];}$$

dove:

- `B` è un identificatore (= il nome dell'array bidimensionale)
- t è un tipo qualsiasi, semplice o a sua volta strutturato
- m ed n sono espressioni intere positive costanti.

L'array bidimensionale `B` può essere rappresentato graficamente come un array di m elementi in cui ogni cella contiene un array di n elementi:

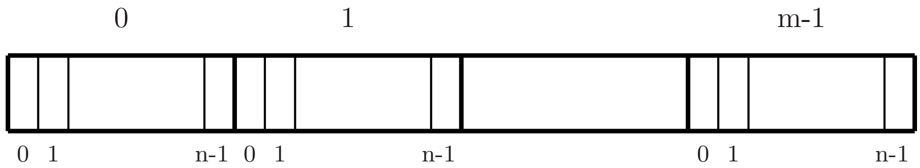


Figura 4.2: Rappresentazione grafica di array bidimensionale.

Esempio 4.13 (Dichiarazione di array bidimensionale)

```
int M1[10][20];
```

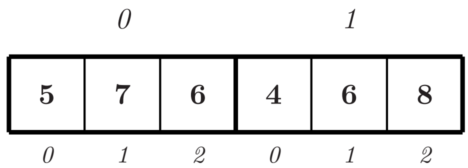
array bidimensionale di nome `M1` *costituito da* 10×20 *elementi di tipo intero (ovvero, un array di 10 elementi ciascuno di tipo array di 20 interi).*

Un array bidimensionale di $m \times n$ elementi di tipo t può anche essere inizializzato in fase di dichiarazione, fornendo una lista di m inizializzatori, ciascuno costituito da n valori di tipo t.

Esempio 4.14 (Dichiarazione di array bidimensionale inizializzato)

```
int M2[2][3] = {{5,7,6},{4,6,8}};
```

array bidimensionale di nome `M2` *costituito da* 2×3 *elementi di tipo intero inizializzati nel modo seguente:*



Operazione di selezione

Ciascun elemento di un array bidimensionale

t `B[`m`][`n`];`

è individuato univocamente da una coppia di numeri interi i, j, detti *indici* dell'array, con $0 \leq i < n$ e $0 \leq j < m$. L'espressione

t `B[`e_1`][`e_2`]`

con e_1 ed e_2 espressioni intere qualsiasi, seleziona l'elemento dell'array `B[`e_1`]` di indice il valore di e_2.

Esempio 4.15 (Selezione di elementi di un array bidimensionale) *Dato l'array bidimensionale di interi* `M2` *mostrato nell'Esempio 4.14, le seguenti istruzioni permettono di incrementare di 1 il valore dell'ultimo elemento di* `M2` *e quindi di stamparlo sullo standard output:*

```
M2[1][2] = M2[1][2] + 1;  //ovvero, M2[1][2]++
cout << M2[1][2];         //viene stampato il valore 9
```

Altre operazioni su array bidimensionali saranno illustrate nel successivo paragrafo 4.4 in cui si mostrerà l'impiego degli array bidimensionali per la realizzazione della struttura dati (astratta) matrice.

Memorizzazione di un array bidimensionale

La creazione di un array bidimensionale di $m \times n$ elementi di tipo t (ovvero, t `B[`m`][`n`]`) comporta l'allocazione in memoria di m array (monodimensionali) consecutivi, ciascuno costituito da m celle di memoria consecutive di tipo t. Pertanto, supponendo che l'area di memoria in cui viene allocato `B` abbia indirizzo base b e che la dimensione (in byte) di un singolo elemento di `B` sia d ($d =$`sizeof(`t`)`), l'indirizzo di memoria del generico elemento `B[`i`][`j`]`, $i = 0, \ldots, n-1$, $j = 0, \ldots, m-1$[2], sarà dato da

$$b + (i * m + j) * d.$$

Ad esempio, per l'array `M2` precedentemente considerato, assumendo $b = 15216$, si ha che l'indirizzo di `M[1][2]` è dato da $15216 + (1 * 3 + 2) * 4$.

Array k-dimensionali

La nozione di array bidimensionale si estende facilmente a quella di array a k dimensioni con $k \geq 1$, la cui dichiarazione in C++ ha la seguente forma:

t `B[`n_1`][`n_2`]`$\cdots$`[`n_k`];`

dove n_1, n_2,...,n_k sono espressioni intere positive costanti.

La selezione del generico elemento di un array a k-dimensioni avviene, in modo analogo a quanto mostrato per gli array bidimensionali, con una variabile con k indici, che in C++ assume la seguente forma:

t `B[`e_1`][`e_2`]`$\cdots$`[`e_k`]`

dove e_1, e_2,...,e_k sono espressioni intere qualsiasi.

4.4 Matrici

Gli array bidimensionali si prestano in modo naturale all'implementazione di una struttura dati (astratta) molto familiare in matematica ed in molti altri campi, quella delle matrici.

Introduciamo le matrici da un punto di vista matematico, in modo rigoroso.

[2]N.B. Per $m = 1$ e $j = 0$ si ricade nel caso degli array ad una dimensione.

[**Matrice** $m \times n$] Sia K un campo e siano m ed n due interi positivi; $m \cdot n$ scalari di K disposti nel modo seguente

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \tag{4.1}$$

costituiscono una *matrice* $m \times n$ *in* K. Si può abbreviare la notazione di questa matrice scrivendola (a_{ij}), $i = 1, \dots, m$ e $j = 1, \dots, n$.

Chiamiamo a_{ij} il *termine* o *componente* o *elemento ij* della matrice.

Se indichiamo con A la matrice scritta nella (4.1), allora la i-esima *riga* è indicata con A_i ed è definita ponendo

$$A_i = (a_{i1}, a_{i2}, \dots, a_{in}).$$

La j-esima *colonna* è indicata da A^j ed è definita ponendo

$$A^j = \begin{pmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{pmatrix}.$$

[**Vettore**] Una matrice $1 \times n$ è un *vettore riga*, mentre una matrice $m \times 1$ è un *vettore colonna* (o *vettore*).

[**Matrice quadrata**] Una matrice $n \times n$ è detta *matrice quadrata*: in una matrice quadrata, la *diagonale principale* è formata dagli elementi a_{ij} tali che $i = j$, mentre gli elementi tali che $i+j = n+1$ costituiscono la *diagonale secondaria.* I termini della diagonale principale sono anche detti *diagonali.*

Esempio 4.16 (Una matrice 3×2)

$$A = \begin{pmatrix} 2 & 5 \\ 3 & 0 \\ 1 & 1 \end{pmatrix}$$

rappresenta una matrice 3×2 *di interi, i cui elementi sono:* $a_{11} = 2$*,* $a_{12} = 5$*,* $a_{21} = 3$*,* $\dots$*.*

4.4.1 Realizzazione tramite array bidimensionali

Una matrice M di $m \times n$ elementi di tipo t come quella mostrata in (4.1) può essere realizzata con un array bidimensionale

t `M[`*m*`][`*n*`];`

memorizzando le m righe della matrice, ciascuna costituita da n elementi di tipo t, negli m elementi dell'array `M`, a partire dall'elemento `M[0][0]` (memorizzazione *per righe*). [3]

Esempio 4.17 *Consideriamo la matrice* B *di* 2×3 *interi*

$$B = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

In C++ questa matrice può essere realizzata con il seguente array bidimensionale

```
int B[2][3] = {{0, 1, 0}, {1, 1, 0}};
```

Si noti che questa realizzazione "diretta" della struttura dati astratta matrice tramite array bidimensionale in C++ risente delle scelte implementative della struttura dati concreta utilizzata. In particolare, risulta necessario considerare gli indici della matrice a partire da 0, piuttosto che da 1 come tipicamente avviene nella definizione astratta di matrice. Quindi, ad esempio, il primo elemento della matrice B mostrata nell'esempio 4.18 (e cioè b_{11}) sarà individuato, nella realizzazione concreta, da `B[0][0]`, mentre l'ultimo (e cioè b_{23}) sarà `B[1][2]`. Questa dipendenza dalle scelte implementative della struttura dati concreta verrà eliminata introducendo la nozione di tipo di dato astratto e la sua realizzazione tramite il costrutto di classe (si veda il capitolo **??**).

Data la corrispondenza immediata tra matrici ed array bidimensionali, così come tra vettori ed array monodimensionali, spesso si usano i termini vettore e matrice anche per riferirsi agli array. Quindi è comune dire che, ad esempio, la dichiarazione

```
int M[2][4];
```

definisce una matrice di `2` righe e `4` colonne di elementi di tipo intero e quindi rappresentare graficamente `M` come una tabella bidimensionale, organizzata in righe e colonne, analogamente a quanto visto per le matrici:

	0	1	2	3
0				
1				

Infine si osservi che, come già sottolineato per gli array (si veda il paragrafo 4.3.4), nel caso in cui la dimensione esatta della matrice non sia nota a

[3] In alternativa, ma meno comunemente, la matrice M potrebbe essere memorizzata *per colonne* in un array bidimensionale t `M[`n`][`m`]`.

priori, si può comunque realizzare la matrice tramite un array bidimensionale di dimensioni ritenute sufficienti (stima per eccesso), utilizzando eventualmente solo una parte dell'array per contenere gli elementi della matrice e memorizzando il numero effettivo dei suoi elementi (che dovrà essere sicuramente minore o uguale alle dimensioni dell'array utilizzato) in due variabili intere che completano l'implementazione della matrice.

4.4.2 Operazioni su matrici

Mostriamo ora l'implementazione in C++ di alcune tipiche operazioni su matrici, assumendo che quest'ultime siano realizzate tramite array bidimensionali nel modo precedentemente illustrato.

Esempio 4.18 (Lettura e scrittura di una matrice) *Il seguente programma permette di leggere da standard input una sequenza di numeri interi, di memorizzarli in una matrice di nome* `mat`*, e quindi di stampare su standard output la matrice* `mat`*, una riga della matrice per ogni linea di stampa. Il numero di righe e di colonne della matrice non è noto a priori e viene richiesto all'utente all'inizio del programma. È però stabilito un limite massimo alle dimensioni della matrice (fissato ad esempio in* 100×100*).*
Problema*. Scrivere un programma che legga, memorizzi e stampi una matrice di interi.*

Input*: numero di righe e di colonne della matrice; elementi della matrice.*
Output*: matrice in forma tabellare.*
Strutture dati*: array bidimensionale.*

Programma*:*

```
#include <iostream>
#include <iomanip>

using namespace std;

#define MAX_RIGHE 100
#define MAX_COLONNE 100

int main() {

  // Richiesta delle dimensioni
  int num_righe;
  do {
    cout << "Numero di righe (max " << MAX_RIGHE << "): ";
    cin >> num_righe;
  } while (num_righe < 1 || num_righe > MAX_RIGHE);
```

```
  int num_colonne;
  do {
    cout<<"Numero di colonne (max "<<MAX_COLONNE << "): ";
    cin >> num_colonne;
  } while (num_colonne < 1 || num_colonne > MAX_COLONNE);

  // Dichiarazione e caricamento della matrice
  int mat[MAX_RIGHE][MAX_COLONNE];
  for (int i = 0; i < num_righe; i++)
    for (int j = 0; j < num_colonne; j++) {
      cout << "Inserisci l'elemento (" << i << ",
                                " << j << "): ";
      cin >> mat[i][j];
    }

  // Stampa
  for (int i = 0; i < num_righe; i++) {
    cout << endl;
    for (int j = 0; j < num_colonne; j++)
      cout << setw(7) << mat[i][j];
  }
  cout << endl;

  // Eventuali elaborazioni sulla matrice

  return 0;
}
```

I due esempi seguenti mostrano altrettante possibili operazioni su matrici. I frammenti di codice mostrati possono facilmente essere inseriti nel programma dell'Esempio 4.19 nella parte in cui sono previste "eventuali elaborazioni sulla matrice".

Esempio 4.19 (Trasposta di una matrice)
Problema*. Scrivere un programma che calcoli la trasposta di una matrice fornita dall'utente.*

Programma*:*

```
  ...
  // Dichiarazione e calcolo della matrice trasposta
  int mat_trasposta[MAX_COLONNE][MAX_RIGHE];
  for (int i = 0; i < num_righe; i++)
    for (int j = 0; j < num_colonne; j++)
      mat_trasposta[j][i] = mat[i][j];
```

```
// Stampa la matrice trasposta.
cout << endl << "\t *** MATRICE TRASPOSTA ***" << endl;
for (int i = 0; i < num_colonne; i++) {
  cout << endl;
  for (int j = 0; j < num_righe; j++)
    cout << setw(6) << mat_trasposta[i][j];
}
cout << endl << endl;
...
```

Esempio 4.20 (Sommatoria diagonale di una matrice quadrata)
Problema. *Scrivere un programma che calcoli e stampi la somma degli elementi sulla diagonale principale di una matrice quadrata fornita dall'utente.*

Programma:

```
...
// Calcolo della somma degli elementi sulla diagonale.
int somma = 0;
for (int i = 0; i < num_righe_colonne; i++)
  somma += mat[i][i];

cout<<"Somma degli elementi sulla diagonale principale: "
    << somma << "." << endl;
...
```

Altre tipiche operazioni del calcolo matriciale, come ad esempio somma, differenza, prodotto tra matrici, inversione, dipendenza lineare, ecc., possono essere realizzate come programmi/funzioni C++ in modo del tutto analogo a quanto visto sopra e sono pertanto lasciate come (utile) esercizio al lettore.

4.5 Stringhe

Un'altra struttura dati di uso molto comune, facilmente realizzabile tramite array, è quella delle stringhe.

Una *stringa* è una sequenza di n caratteri adiacenti, con $n \geq 0$.

Ad esempio, `"ciao"` è una stringa di 4 caratteri, mentre `""` è una stringa di 0 caratteri (detta *stringa vuota*).[4]

[4] I doppi apici sono usati, nella tipica notazione per le stringhe, come delimitatori della stringa e pertanto non fanno parte di essa.

4.5.1 Realizzazione tramite array

Una stringa può essere facilmente realizzata tramite un array di caratteri. Questa è ad esempio l'implementazione delle stringhe standard utilizzata in C ed è quella che esamineremo in questo capitolo. Altre implementazioni sono ovviamente possibili. In C++, in particolare, la stringa è vista come un tipo di dato astratto e quindi realizzata tramite un'opportuna classe (denominata `string`) che di fatto nasconde all'utente l'implementazione concreta della struttura dati (astratta) stringa. Ritorneremo sulla realizzazione delle stringhe del C++ in un capitolo successivo. Per ora esaminiamo in dettaglio l'implementazione fornita dal C. Si noti che essendo il C++ un sovrainsieme del C è comunque sempre possibile utilizzare questa implementazione anche all'interno dell'ambiente C++.

Una stringa "$c_1 c_2 \ldots c_n$" di n caratteri è realizzata in C tramite un array di k elementi di tipo `char`, con $k > n$, costituiti nell'ordine dagli n caratteri della stringa, seguiti dal carattere speciale `'\0'` per indicare la fine della stringa.

0	1	$\cdots$	$n-1$	n	$n+1$	$\cdots$	k
'c_1'	'c_2'	$\cdots$	'c_n'	'\0'	-	$\cdots$	-

Una stringa di nome `s` può pertanto essere dichiarata in C (C++) come un normale array di caratteri, con un'opportuna dimensione (maggiore della lunghezza massima prevista per la stringa), opportunamente inizializzata con i caratteri componenti la stringa stessa più il delimitatore `'\0'`.

Esempio 4.21 *La dichiarazione*

```
char s[10] = {'c','i','a','o','\0'};
```

definisce una stringa di nome `s` *con valore (iniziale)* `"ciao"` *e una lunghezza massima di 9 caratteri (si noti che, come già osservato per gli array in generale, anche in questo caso è possibile fornire una lista di inizializzatori con un numero di elementi inferiore alla dimensione dell'array):*

s	0	1	2	3	4	5	6	7	8	9
	'c'	'i'	'a'	'o'	'\0'	-	-	-	-	-

Il C++, così come la maggior parte dei linguaggi di programmazione, prevede anche la possibilità di utilizzare apposite *costanti stringa*, in cui la stringa viene scritta con la più naturale notazione tra doppi apici. Le costanti stringa possono essere usate sia come inizializzatori in una dichiarazione di una variabile, sia all'interno di un'espressione che preveda l'uso di questo tipo di dato. Ad esempio, la dichiarazione dell'Esempio 4.22, può essere scritta, equivalentemente, come

```
char s[10] = "ciao";
```

In questo caso il delimitatore `'\0'` verrà inserito automaticamente al termine della stringa (la rappresentazione interna della stringa e', comunque, la stessa mostrata nell'Esempio 4.22).

Nota. Una costante di stringa può contenere qualsiasi carattere, compresi lo spazio ed i caratteri speciali come `'\n'` ("a capo"), `'\t'` ("tabulazione"), ma non `'\0'`.

È possibile dichiarare una stringa anche senza specificare esplicitamente la dimensione dell'array che la contiene; in questo caso, come per un qualsiasi array con inizializzazione esplicita, la dimensione sarà quella dedotta dal numero di elementi presenti nell'inizializzatore (nel caso delle stringhe bisogna ricordarsi sempre che c'è anche da contare il delimitatore di stringa `'\0'`). Ad esempio, con la dichiarazione

```
char saluti[] = "ciao";
```

vengono allocati esattamente 5 byte nel modo seguente:

	0	1	2	3	4
`saluti`	`'c'`	`'i'`	`'a'`	`'o'`	`'\0'`

Nota. La costante di stringa `""` indica la *stringa vuota* e può essere usata come qualsiasi altra costante di stringa. Ad esempio la dichiarazione

```
char r[] = "";
```

definisce una stringa di nome `r`, inizializzata a stringa vuota. La sua rappresentazione interna è pertanto la seguente:

	0
`r`	`'\0'`

Nota. Attenzione a non confondere le *costanti carattere* con le *costanti stringa*. Ad esempio, la costante `'a'` e la costante `"a"` denotano oggetti profondamente diversi: la prima, `'a'`, è un carattere singolo, rappresentato dal suo codice ASCII (un intero tra 0 e 255), memorizzato in un singolo byte; il secondo, `"a"`, è un stringa di lunghezza 1 e come tale memorizzata in un array di due elementi di cui il primo è il carattere `'a'` ed il secondo è il delimitatore di stringa `'\0'`.

4.5.2 Operazioni su stringhe

Tipiche operazioni su stringhe, oltre a quelle generali di assegnamento, di confronto e di input/output, sono le operazioni di calcolo della lunghezza di un stringa, di concatenazione tra due stringhe, di ricerca di una sottostringa in una stringa data. E' inoltre comune dover accedere ad un singolo carattere di una stringa, specialmente al suo carattere di testa, o dover aggiungere o rimuovere un singolo carattere da una stringa data.

Le stringhe stile C che abbiamo presentato finora sono a tutti gli effetti array di caratteri e come tali non prevedono alcuna operazione primitiva direttamente supportata dal linguaggio, a parte l'accesso al singolo elemento tramite l'operazione di selezione prevista dagli array. In C++ è però possibile operare sulle stringhe stile C con una serie di funzioni predefinite, e precisamente:

- per l'input/output, tramite le funzioni della libreria `iostream`;
- per altre operazioni tipiche della manipolazione di stringhe, tramite le funzioni della libreria `cstring`.

Input e output di stringhe

Gli operatori di input (`>>`) e di output (`<<`) che già conosciamo si applicano in modo naturale anche alle stringhe.
Esempio: scrittura di una stringa.

L'esecuzione delle istruzioni:

```
char msg[100]="errore";
cout << msg << endl;
```

causa l'invio sul monitor (lo standard output di default) del messaggio:

```
errore
```

seguito da un comando di “a capo”.

Lo stesso effetto si può ottenere anche scrivendo direttamente la costante di stringa `"errore"` come espressione dell'operatore `<<`:

```
cout << "errore" << endl;
```

oppure, inserendo il carattere speciale che provoca l'“a capo” direttamente nella stringa da stampare:

```
cout << "errore\n";
```

Esempio: lettura di una stringa.

L'esecuzione delle istruzioni:

```
char nome[32];
cin >> nome;
```

provoca la lettura, uno dopo l'altro, dei caratteri presenti sullo standard input fino ad incontrare il carattere "spazio" o "a capo" e li assegna nell'ordine all'array `nome`.[5] Al termine, viene inserito automaticamente il carattere di fine stringa `'\0'`.

Supponiamo che l'utente digiti la sequenza di caratteri

`mario bianchi`

Al termine della lettura la stringa `nome` avrà il seguente contenuto:

	0	1	2	3	4	5		···		31
nome	'm'	'a'	'r'	'i'	'o'	'\0'				

I caratteri rimasti sullo stream di input potranno essere letti con successive operazioni di input. Se invece venisse digitata, ad esempio, la sequenza `mario_bianchi` (seguita, al solito, dal comando di "a capo"), allora tutta la sequenza verrebbe memorizzata come stringa nell'array `nome`, con il carattere `'\0'` inserito, automaticamente, al termine della sequenza.

Nota. Si osservi che è necessario dichiarare l'array utilizzato per memorizzare la stringa di dimensioni adeguate a contenere la stringa stessa. Dichiarare, ad esempio, l'array senza dimensioni e quindi memorizzarvi una stringa come nella sequenza di istruzioni

```
char nome[];
cin >> nome;
```

è sicuramente una situazione d'errore e può portare ad un comportamento anomalo del programma dato che l'operazione di lettura preleva comunque i caratteri presenti sullo stream di input e li memorizza comunque nell'area di memoria individuata da `nome`: siccome l'indirizzo memorizzato in `nome`, e quindi l'area in cui verranno scritti i caratteri della stringa, è indefinito il risultato del programma è del tutto imprevedibile. Problemi analoghi si presentano anche quando il numero di caratteri che vengono letti dallo stream di input eccede la dimensione massima prevista per l'array.

4.6 Struct

4.7 Tabelle

[5] La lettura termina anche se si incontra il carattere di "end_of_file", come vedremo meglio in un successivo capitolo, dedicato all'input/output da file.

Capitolo 5

Sottoprogrammi

5.1 Definizione di funzioni

5.2 Uso di funzioni

5.3 Passaggio parametri

Passaggio parametri: per valore, per riferimento. Passaggio parametri di tipo array. Funzioni `void`.

5.4 Esempi

5.5 Funzioni ricorsive

Ricorsione. Definizioni e funzioni ricorsive. Esempi. Implementazione ricorsione (cenni).

Capitolo 6

Approfondimenti: nomi e tipi

6.1 Visibilità degli identificatori: regole di "scope"

Nomi, ambienti e regole di "scope": struttura a blocchi, ambiente locale e non-locale, variabili globali, "scope" statico vs. dinamico (cenni)

6.2 Ancora sui tipi

6.2.1 Modificatori di tipo

BOZZA

È possibile far precedere ai tipi di dato fondamentali, escluso il `bool`, dei *modificatori* in modo da alterare alcune caratteristiche del tipo fondamentale e poterlo adattare in modo più preciso a diverse situazioni. Ci sono quattro modificatori: `long`, `short`, `signed` e `unsigned`.

`long` e `short` modificano il massimo ed il minimo valore che il dato può contenere. La gerarchia per quanto riguarda la dimensione di tipi interi è `short int`, `int` e `long int`; per i numeri in virgola mobile è `float`, `double` e `long double`. "long float" non è un tipo legale e non esistono numeri in virgola mobile `short`.

I modificatori `signed` o `unsigned` dicono al compilatore come usare il bit di segno (il bit più a sinistra) per tipi interi e caratteri (i numeri in virgola mobile sono sempre `signed`). Un numero `unsigned` non tiene traccia del segno e quindi ha un bit in più disponibile. Questo significa che può immagazzinare numeri il doppio più grandi di quanto possa fare un numero `signed`. Per esempio un `signed char` a 8 bit può rappresentare i valori da $-2^7 = -128$ a $2^7 - 1 = 127$; un `unsigned char` da 0 a $2^8 - 1 = 255$. Per interi e caratteri, di default è assunto che il valore sia dotato di segno. Il seguente esempio mostra la dimensione di diversi tipi di dati in byte utilizzando l'operatore `sizeof()` che restituisce la dimensione, in byte, di un oggetto o di un tipo.

```
#include <iostream>
using namespace std;
int main(){
  cout << "char           " << sizeof(char) << endl;
```

```
  cout << "unsigned char  " << sizeof(unsigned char) << endl;
  cout << "short          " << sizeof(short) << endl;
  cout << "unsigned short " << sizeof(unsigned short) <<endl;
  cout << "int            " << sizeof(int) << endl;
  cout << "unsigned int   " << sizeof(unsigned int) << endl;
  cout << "long           " << sizeof(long) << endl;
  cout << "unsigned  long " << sizeof(unsigned long) << endl;
  cout << "float          " << sizeof(float) << endl;
  cout << "double         " << sizeof(double) << endl;
  cout << "long double    " << sizeof(long double) << endl;
  cout << "bool           " << sizeof(bool) << endl;
  return 0;
}
```

L'output prodotto dal programma è:

```
char           1
unsigned char  1
short          2
unsigned short 2
int            4
unsigned int   4
long           4
unsigned  long 4
float          4
double         8
long double    12
bool           1
```

6.2.2 Tipi definiti da utente

Tipi per enumerazione

Il costrutto `typedef`

6.2.3 Equivalenza tra tipi

6.2.4 Compatibilità e conversione di tipo

Capitolo 7

Input/output su file

Bibliografia

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] A. Bertossi. *Algoritmi e strutture dati*. UTET libreria, 2000.

[3] M. Cadoli, M. Lenzerini, P. Naggar, and A. Schaerf. *Fondamenti della progettazione dei programmi (Principi, tecniche e loro applicazioni in C++)*. CittàStudiEdizioni, 1997.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduzione agli algoritmi*. Jackson, 1994.

[5] E. W. Dijkstra. *Notes on structured programming*. Hoare, 1972.

[6] M. Gabbrielli and S. Martini. *Linguaggi di programmazione: Principi e Paradigmi*. McGraw-Hill, 2005.

[7] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Automi, linguaggi e calcolabilità*. Addison-Wesley Italia, 2003.

[8] C. Laneve. *La Descrizione Operazionale dei Linguaggi di Programmazione*. Franco Angeli, 1998.

[9] T. W. Pratt and M. V. Zelkowitz. *Programming languages: Design and Implementation*. Prentice-Hall, 2001.

[10] H. Rogers. *Teoria delle funzioni ricorsive e della calcolabilità effettiva*. Tecniche Nuove, 1992.

[11] A. S. Tannenbaum. *Structured Computer Organization*. Prentice-Hall, 1999.

[12] G. Winskel. *La semantica formale dei linguaggi di programmazione*. UTET libreria, 1999.

[13] B. W.Kernighan and D.M.Ritchie. *The C Programming Language*. Aprentice Hall, 1988.

Elenco delle figure

Elenco delle tabelle