Introduzione alla programmazione con linguaggio di riferimento il C++

Gianfranco Rossi, Tatiana Zolo

Università di Parma
Dip. di Matematica
43100 Parma (Italy)
gianfranco.rossi@unipr.it

4 febbraio 2006

Indice

1 Introduzione alla programmazione **3**

1.1 Algoritmi 3

1.1.1 Problemi ed algoritmi 3

1.1.2 Descrizione di algoritmi 6

1.1.3 I diagrammi di flusso (concetti di base) 8

1.2 I linguaggi di programmazione 15

1.2.1 Linguaggi “a basso livello” e “ad alto livello” 15

1.2.2 Modalità d’esecuzione: compilazione e interpretazione . 16

1.2.3 Linguaggi di programmazione esistenti 19

1.3 Il linguaggio C++ 20

1.3.1 Dal C al C++ 20

1.3.2 Un esempio di programma C++ 21

1.3.3 Convenzioni di programmazione 24

1.4 L’ambiente di programmazione 26

2 Elementi di base di un programma **30**

2.1 Identificatori 30

2.2 Variabili 31

2.2.1 Dichiarazione di variabile 32

2.3 Tipi di dato primitivi 34

2.3.1 Il tipo `int` 34

2.3.2 Il tipo `float` 35

2.3.3 Il tipo `char` 36

2.3.4 Il tipo `bool` 37

2.4 Statement di assegnamento 39

2.5 Espressioni ed operatori 41

2.5.1 Valutazione di una espressione 43

2.5.2 Tipo di un’espressione 46

2.5.3 Espressioni booleane 47

2.6 Ancora sullo statement di assegnamento 48

Bibliografia **50**

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."([4]).

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.

Prefiggiamoci come scopo la risoluzione del 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.

Algoritmo: sequenza 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.

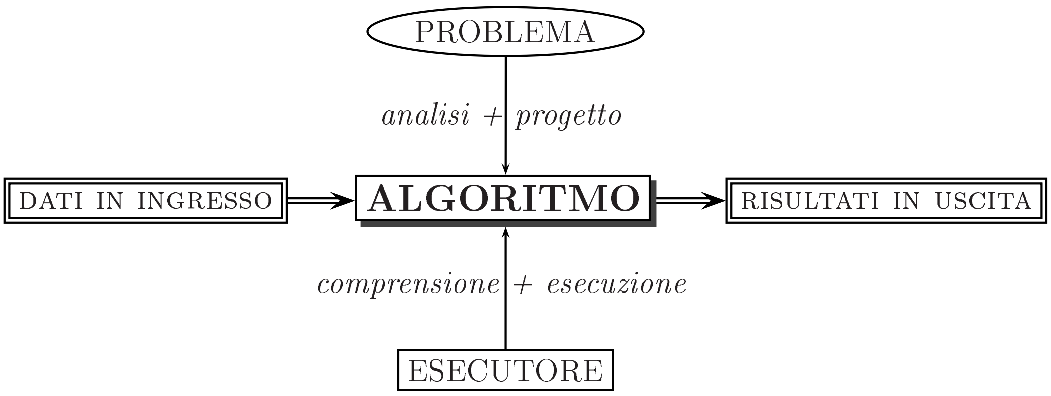


Figura 1.1: Dal problema all'algoritmo.

Alla formulazione dell'algoritmo si arriva attraverso una fase di analisi del problema e quindi di progettazione del procedimento risolutivo adeguato. 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.

Approfondimento 1.1 (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 teorica 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* [6] *e* [9].

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 teorica 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* [3] *e* [2].

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 1.2 (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 1, a meno di modifica esplicita dell'ordine di esecuzione (ad esempio, "`vai all'istruzione 2`".*

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 1.2 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 sottocapitolo 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—ovali, rettangolari, romboidali—connessi da frecce. 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 sottocapitolo 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.

Blocchi rettangolari

I blocchi rettangolari vengono utilizzati principalmente per rappresentare istruzioni di assegnamento:[1]

[1]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.

Istruzione di assegnamento

$$\downarrow$$
$$\boxed{v \leftarrow E}$$
$$\downarrow$$

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 $x + 1$).

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:

$$\downarrow$$
$$\boxed{x \leftarrow 1}$$
$$\downarrow$$

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

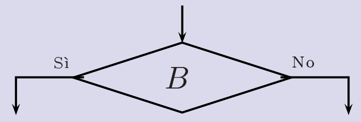
$$\downarrow$$
$$\boxed{x \leftarrow y + 1}$$
$$\downarrow$$

  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:

Istruzione di salto con condizione

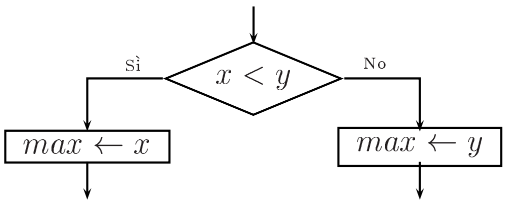


dove B è un'espressione booleana, cioè un'espressione costruita tramite operatori di relazione (quali "=", ">", "<", . . .) e connettivi logici (*and*, *or*, *not*), il cui risultato può essere soltanto di tipo booleano (*vero* o *falso*).

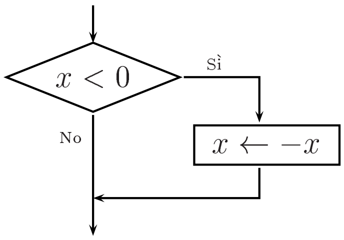
Il significato di tale blocco è: *valuta l'espressione B; se B ha valore "vero" continua dalla parte etichettata con* Sì*; altrimenti, cioè quando B ha valore "falso", continua dalla parte etichettata con* No.

Esempi:

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

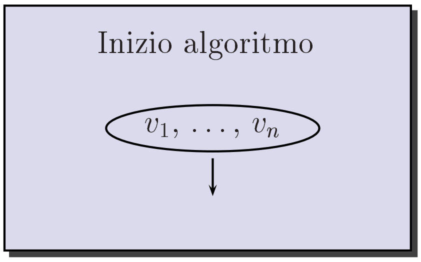


- (*esecuzione condizionale*) se il valore della variabile x è minore di 0 assegna ad x il suo opposto e passa all'istruzione successiva; altrimenti, passa direttamente all'istruzione successiva:



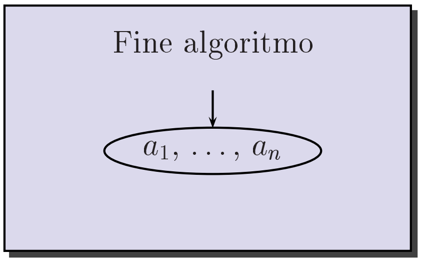
Blocchi ovali

I blocchi ovali vengono utilizzati per rappresentare l'inizio e la fine dell'algoritmo. La loro forma generale è di seguito descritta.



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.



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 1.3 (Moltiplicazione per somme–continuazione)
Riprendiamo l'algoritmo di moltiplicazione per somme precedentemente descritto, 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 a blocchi che descrive questo algoritmo.*

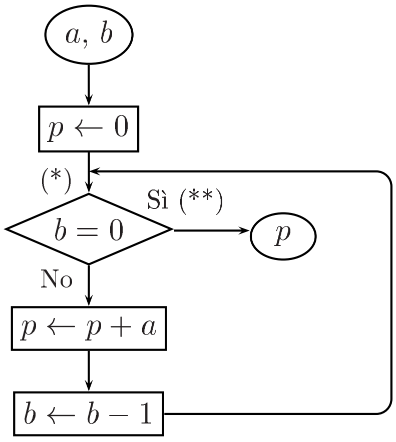


Figura 1.3: Diagramma a blocchi che descrive l'algoritmo di moltiplicazione per somme.

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 fino a che "$b = 0$" è falsa. 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..

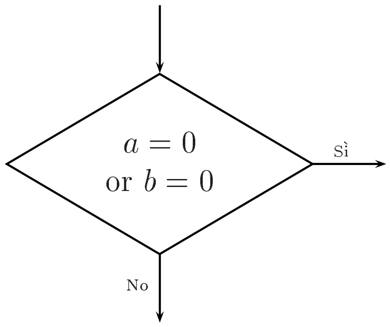
Casi particolari, ottimizzazioni

Riferendoci sempre all'Esempio 1.2 analizziamo un caso particolare, cioè cosa succede quando uno dei due dati in ingresso è nullo. La tabella in Figura 1.4 mostra che l'algoritmo continua a funzionare, ma nel secondo caso proposto è particolarmente inefficiente.

	(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*.

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



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.

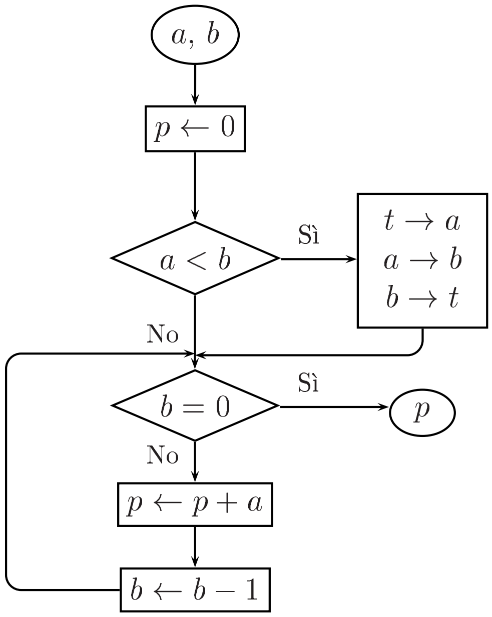


Figura 1.5: Algoritmo ottimizzato di moltiplicazione per somme.

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

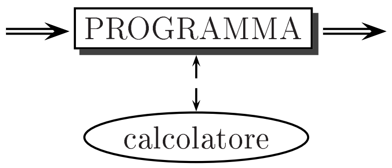
Un linguaggio di programmazione è un formalismo di descrizione di algoritmi eseguibile dal calcolatore. La descrizione di un algoritmo per un problema $\mathcal{P}$ in un linguaggio di programmazione $\mathcal{L}$, e cioè un *programma in* $\mathcal{L}$, è costituito da una sequenza finita di istruzioni 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.[2]

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:



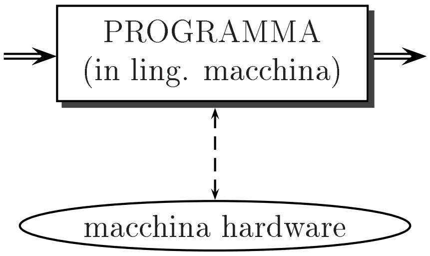
Vedremo come questo schema generale possa essere ulteriormente precisato distinguendo tra vari tipi di linguaggi di programmazione.

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.

I linguaggi a basso livello sono quelli che possono essere interpretati direttamente dalla macchina hardware e di conseguenza sono strettamente legati alla progettazione dell'hardware stesso. 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.

[2] Per una trattazione approfondita delle problematiche inerenti la semantica dei linguaggi di programmazione si vedano, ad esempio, i testi [7] e [11].



Approfondimento 1.4 (Linguaggio assembler) *Generalmente per ogni linguaggio macchina esiste anche una versione più simbolica, detta* linguaggio assembler*. 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 assembler richiede l'utilizzo di programmi traduttori, detti* assemblatori*, in grado di trasformare il programma scritto in linguaggio assembler nel corrispondente programma in linguaggio macchina (e quindi comprensibile per il calcolatore).*

Per un approfondimento di linguaggio macchina e assembler, nonché, in generale, di tutto ciò che riguarda la macchina hardware, si veda ad esempio il testo [10].

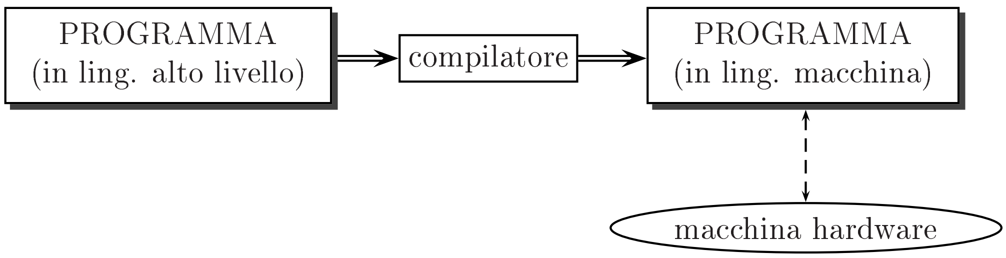
I linguaggi ad alto livello nascondono, in modo più o meno completo, le caratteristiche proprie delle diverse macchine hardware ed offrono al programmatore una sintassi molto più vicina al linguaggio naturale rispetto a quanto possono fare i linguaggi macchina o assembler. Essi risultano pertanto più 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 esecuzione di programmi scritti in un linguaggio ad alto livello: la *compilazione* e l'*interpretazione* (si veda 1.2.2).

Il C ed il C++ sono tra i più potenti e più diffusamente utilizzati linguaggi ad alto livello.

1.2.2 Modalità d'esecuzione: compilazione e interpretazione

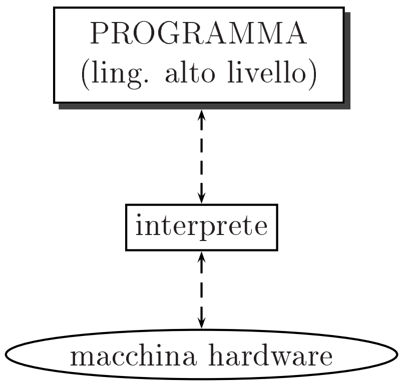
La *compilazione* di un programma consiste nell'analisi delle istruzioni del programma che vengono lette e tradotte da un programma traduttore, il *compilatore*, e nella generazione delle istruzioni macchina in grado, quando verranno successivamente eseguite dalla macchina hardware, di realizzare il 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[3].

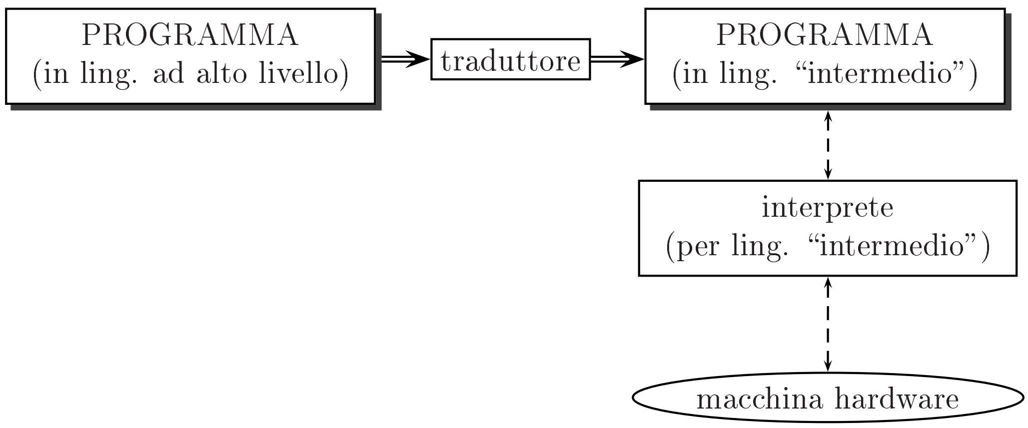
Sono solitamente realizzati in questo modo linguaggi di programmazione "non convenzionali", quali il Prolog o il LISP.



[3] 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 comperterà una maggiore occupazione di memoria. D'altra parte, l'esecuzione tramite interpretazione rende il linguaggio di programmazione più adatto ad un uso 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 1.5 (Compilazione e interpretazione) *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 in Byte-code*, che verrà poi interpretato tramite una macchina astratta (e cioè un programma interprete), denominato "Java Virtual Machine" (JVM). 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 ed in particolare della modalità di esecuzione basata sulla "compilazione". Per una trattazione più completa ed approfondita dei linguaggi macchina ed assembler si rimanda, ad esempio, al testo [10], mentre, per quanto riguarda le problematiche relative a compilazione ed interpretazione, si rimanda, ad esempio, al testo [5].

1.2.3 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 [8]) 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, compilato, 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.

1.3 Il linguaggio C++

1.3.1 Dal C al C++

Il C venne progettato ed implementato nel 1972 da Ritchie. Inizialmente era strettamente legato a UNIX (buona parte di UNIX e delle sue utilities di sistema sono scritte in C). Venne pensato come linguaggio per la programmazione di sistema, e quindi ad ampio spettro di applicabilità. Punta principalmente su efficienza e flessibilità di uso (è un linguaggio piuttosto permissivo).

Il C++ nasce come estensione del C e venne sviluppato da Bjarne Stoustrup ai laboratori Bell AT&T agli inizi degli anni '80. Include tutte le possibilità del linguaggio C (è per esempio possibile usare un compilatore C++ anche per sviluppare ed eseguire programmi in C) più molte altre. Introduce i concetti di *classe*, *oggetti* ed *ereditarietà* tra classi, divenendo un ottimo supporto per la programmazione orientata agli oggetti. Inoltre, altre possibilità non strettamente correlate con la programmazione ad oggetti sono l'*overloading di operatori e funzioni*, nuove possibilità per il passaggio di parametri, 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 overloading di funzioni ed operatori e 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.

1.3.2 Un esempio di programma C++

In questo sottocapitolo mostriamo un primo semplice esempio di programma scritto in C++. Tutte le diverse istruzioni C++ qui introdotte in modo intuitivo verranno riprese ed analizzate in modo più approfondito in seguito.

Un programma C++—così come i programmi scritti nella maggior parte dei linguaggi di programmazione convenzionali—è costituito da due entità sintattiche di base: le *dichiarazioni* e gli *statement* (o comandi). Le prime servono fondamentalmente a descrivere i dati utilizzati nel programma. Gli statement invece costituiscono le istruzioni vere e proprie del programma, che verranno eseguite dal calcolatore.

Si noti che nei diagrammi di flusso ci si basava su una descrizione dei dati informale, mentre nei linguaggi di programmazione questo aspetto è precisamente formalizzato e quindi anche il linguaggio con cui si descrivono i dati rispetta 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.[4]

[4] Tutto il codice presentato in queste note è stato provato con compilatore gcc (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;
}
```

Commenti:

- `#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`.

- `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 costanti di 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.

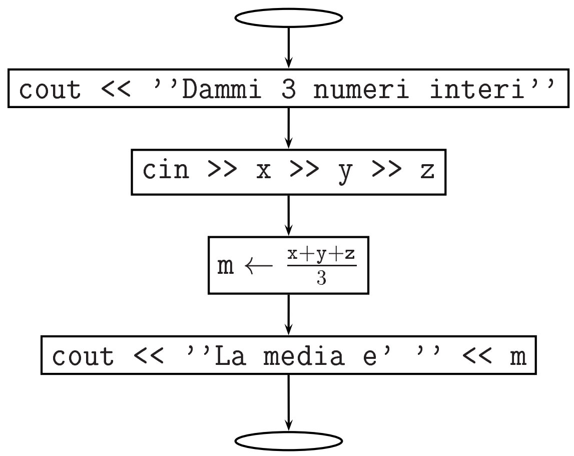


Figura 1.6: Diagramma di flusso per il calcolo della media di 3 numeri interi.

- 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 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 tutti 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 */`
  - `// 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 poter scrivere ed eseguire programmi 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 a tutte le fasi dello sviluppo di un programma. Questo insieme di strumenti—più o meno sofisticati, integrati fra loro, 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++.

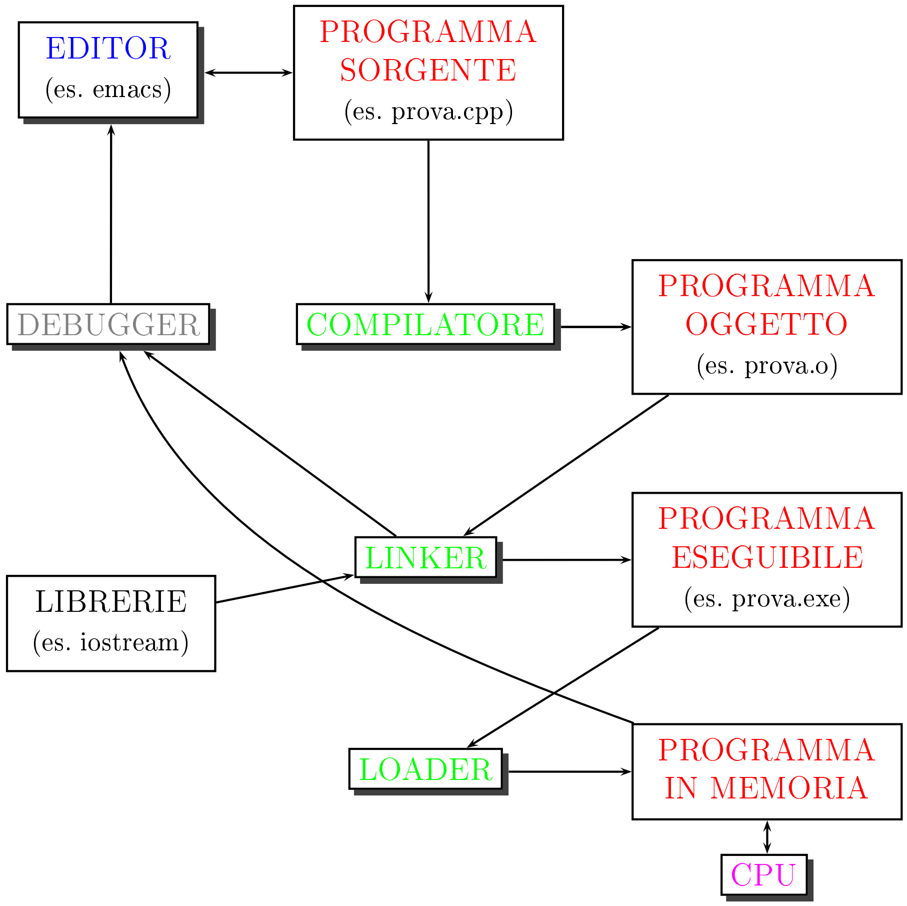


Figura 1.7: Schema di un ambiente di programmazione.

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 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*). La fase di compilazione è 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 e rimandiamo il lettore interessato ad approfondirli in [1].

La terza fase è quella del *linking* (che in italiano si può tradurre con *collegamento*). 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 quinta 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 simile), ottenendo come risultato il corrispondente programma oggetto, o la generazione di opportune segnalazioni nel caso siano presenti errori sintattici.

Approfondimento 1.6 (Fase di "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* preprocessor, *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 del programma, 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`, `>`, `main`, 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, usati 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 qualsiasi di lettere alfabetiche e cifre, iniziante con una lettera, con esclusione dei simboli speciali (ad esempio, "+", ">", "!", spazio, ...), ad eccezione del simbolo "_" (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.

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 nel paragrafo **??**, dedicato alle cosiddette *"regole di scope"*.

2.2 Variabili

Una variabile è un'astrazione della cella di memoria, o più in generale, di un "contenitore" in grado di contenere un dato alla volta. Una variabile è caratterizzata da:

- un nome;
- un valore attuale;
- un tipo che specifica l'insieme dei possibili valori e delle possibili operazioni per la variabile
- l'indirizzo della cella (o *locazione*) di memoria che contiene il valore della variabile.

Il valore di una variabile è modificabile, ad esempio tramite un'istruzione di assegnamento. Il nuovo valore sostituisce quello attuale, che è definitivamente perso.

Approfondimento 2.2 (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 variabile 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* [5].

2.2.1 Dichiarazione di variabile

Nella maggior parte dei linguaggi di programmazione (compreso il C++) qualsiasi variabile prima di essere usata deve essere dichiarata. La dichiarazione di variabile serve a specificare le caratteristiche di una variabile: nome, tipo ed eventualmente un valore iniziale. Il tipo serve tra l'altro a specificare quanto spazio di memoria deve essere allocato quando la variabile verrà creata. Dalla dichiarazione in poi, la variabile può essere utilizzata. Nel caso si utilizzi una variabile senza averla prima dichiarata viene normalmente segnalato un errore dal compilatore.

La dichiarazione di variabile avviene principalmente tramite un'apposito costrutto, detto di "dichiarazione di variabile". Vediamone le caratteristiche base nel linguaggio C++.

Dichiarazione di variabile. 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).[1]

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 2.3 (Dichiarazioni di variabili)

[1]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.

```
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 un errore logico utilizzare il valore di una variabile prima di avergliene assegnato uno esplicitamente.

In molti linguaggi (ad esempio in 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:

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

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

L'elaborazione di una dichiarazione di variabile di tipo t comporta, tra l'altro, l'allocazione della memoria principale necessaria per contenere un valore di tipo t, con la conseguente creazione dell'associazione tra l'identificatore della variabile e la porzione di memoria allocata. Indicheremo questa operazione come *creazione* della variabile. L'operazione simmetrica di *distruzione* di una variabile avviene invece solitamente in modo implicito, al termine del programma o del sottoprogramma contenente la dichiarazione. Il tempo che intercorre tra la creazione di una variabile e la sua distruzione è detto *tempo di vita* della variabile. Ritorneremo sui concetti di creazione e distruzione di una variabile, con particolare riferimento al linguaggio C++, nel paragrafo **??**.

La porzione di programma in cui rimane attiva una certa associazione identificatore - variabile, ovvero la porzione di programma in cui una variabile è visibile, e quindi utilizzabile, prende il nome di *campo d'azione* (o *"scope"*) della variabile. Il campo d'azione di una variabile dipende in generale dalla struttura (statica) del programma e non necessariamente coincide con il tempo di vita della variabile. Analizzeremo le regole di "scope"del C++ nel paragrafo **??**.

2.3 Tipi di dato primitivi

Tipo: il tipo di una variabile è una coppia, costituita 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 $[-2^{31}, 2^{31}-1]$.[2]

[2] 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` (per poterle utilizzare è quindi necessario inserire la direttiva `#include <climits>`).

I valori del tipo `int` sono denotati da *costanti di intero* 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 div 2` restituisce come risultato 2, mentre `5 % 2` restituisce 1;
- operazioni di confronto: `==` (uguaglianza), `!=` (diversità), `<`, `>`, `<=`, `>=`.

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 tutti quelli sugli altri tipi di dato semplici primitivi che introdurremo fra breve, sono usati in modo *infisso* (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")[3].

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 cosidetta *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`.

Acceniamo già ora al fatto che il C++ offre come tipi primitivi diversi sottotipi e supertipi dei tipi semplici primitivi qui presentati. In particolare,

[3]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.

per quanto riguarda i numeri reali, viene spesso utilizzato, in alternativa al tipo `float`, il tipo `double`. `double` è un supertipo di `float` (ovvero `float` è un sottotipo di `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 sovrainsieme dei valori di `float`. A livello concreto questo significa solitamente—ma non necessariamente—una quantità di memoria doppia per un numero `double` rispetto a 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. Il fatto che `float` sia un sottotipo di `double` significa 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 al termine di questo capitolo altri casi di sottotipo e supertipo per altri tipi semplici primitivi.

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 di 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' $<$ 'C' $<$ $\ldots$

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'
```

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` (ricordando che si assume `false` < `true`).

Le operazioni logiche prendono operandi di tipo `bool` e restituiscono un risultato di tipo `bool`. Gli operatori di 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".

b_1	b_2	$!\ b_1$	b_1 && b_2	$b_1 \mid\mid b_2$
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.

Tramite le tabelle di verità è possibile ricavare il significato di espressioni booleane qualsiasi a partire da quello delle espressioni componenti. Ad esempio la tabella di verità per l'espressione $a\ \mid\mid\ !\ b$ è la seguente:

Come nel caso del tipo `char`, anche per il tipo `bool` il C++ (e prim'ancora il C) fa la scelta di permettere che l'implementazione del tipo sia visibile a livello

a	b	a || $!\,b$
T	T	T
T	F	T
F	T	F
F	F	T

utente: i valori booleani `true` e `false`, infatti, sono realizzati semplicemente con i due interi 1 ed 0, rispettivamente. Dunque, in C++, i booleani possono a tutti gli effetti essere trattati come gli interi 0 e 1 e, viceversa, gli interi 0 e 1 possono vedersi come booleani. Più precisamente, tutti i valori interi diversi da 0 sono visti come il valore booleano `true`.

Questo spiega anche perché 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

Iniziamo in questo paragrafo l'esame degli statement di un linguaggio di programmazione, con particolare riferimento, come al solito, agli statement del linguaggio C++. Esaminiamo dapprima lo statement di assegnamento, istruzione fondamentale per tutti i linguaggi imperativi. Mostreremo anche come specificare l'esecuzione in sequenza di due o più statement e come raggruppare una sequenza di statement in un unico statement composto. Gli altri statement per realizzare strutture di controllo diverse (precisamente, gli statement di selezione, iterativi e di salto) saranno invece presentati nel capitolo successivo.

Assegnamento. Lo statement di assegnamento in C++ ha la seguente forma base:

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).

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 2.4 (Statement di assegnamento)

```
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 soltnato 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, come "effetto collaterale" della sua esecuzione, 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 terza variabile temporanea (o ausiliaria), che chiameremo `temp`, nel modo seguente:

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

Ritorneremo ancora sullo statement di assegnamento dopo aver visto alcune precisazioni sulla nozione di espressione nella sezione successiva.

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;
- un'espressione *expr* tra parentesi
$$(expr);$$
- un'espressione *expr* preceduta da segno
$$+expr \text{ oppure } -expr.$$
- una chiamata di funzione
$$nome\_funzione(expr_1, \ldots, expr_n);$$

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

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 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 **??**) 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 (si veda l'*overloading degli operatori* presentato nel capitolo **??**).

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 nell'approfondimento **??**).

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.

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.[4]

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 nota 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 2.5 *Consideriamo l'espressione*

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

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

Esempio 2.6 *Consideriamo l'espressione*

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

[4] Si noti che questa è una forma semplice di controllo del flusso d'esecuzione, che si applica invece che a statement, come vedremo per le diverse strutture di controllo che introdurremo successivamente, alla valutazione della singola espressione.

Poiché 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 2.7 *La valutazione dell'espressione*

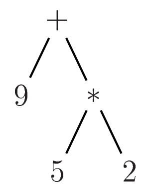
```
(9 + 5) * 2
```

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

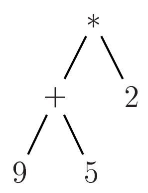
In generale, dunque, la presenza di opportune regole di precedenza ci permette di ridurre l'uso esplicito delle parentesi, semplificando la scrittura dei programmi. Ad esempio, nell'espressione dell'Esempio 2.6 abbiamo potuto evitare le parentesi attorno alle due sottoespressioni di confronto, grazie 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 2.8 (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 [*?*]).*

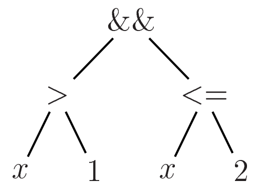
*Assumiamo che dato un albero sintattico di un'espressione si valutino per primi i sottoalberi più in profondità. L'espressione dell'*Esempio 2.5 *corrisponde all'albero sintattico:*



mentre l'espressione dell'Esempio 2.7 corrisponderà all'albero sintattico:



*Analogamente l'espressione dell'*Esempio 2.6 *corrisponde all'*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 invece associativi a destra.

Esempio 2.9 (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))`

Nel Paragrafo 2.6 *vedremo la possibilità di concatenare più operatori di assegnamento.*

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.[5] Questa decisione è lasciata alla realizzazione dello specifico compilatore e potrebbe quindi

[5] 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.

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 `boolean`, 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`. 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:

`int` < `float` < `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`.[6]

[6]Si noti che nel primo esempio di programma C++ mostrato nel paragrafo 1.3.2 abbiamo usato la costante `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, infatti,

Per le funzioni definite dall'utente il tipo del risultato della funzione è quello specificato nella dichiarazione della funzione stessa (si veda il Capitolo ??).

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.
  ```

Si noti che la scrittura matematica $0 \leq x < 10$ viene realizzata in C++ con l'espressione booleana

```
x >= 0  &&  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*:

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

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*[7].

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*. Analogamenta, 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++.

2.6 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[8]. 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* visto come espressione 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

[7] 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.

[8] 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 [?].

```
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 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;
double 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).

Bibliografia

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1998.

[2] A. Bertossi. *Algoritmi e strutture dati*. UTET libreria, 2000.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduzione agli algoritmi*. Jackson, 1994.

[4] E. W. Dijkstra. *Notes on structured programming*. Hoare, 1972.

[5] M. Gabbrielli and S. Martini. *Linguaggi di programmazione: Principi e Paradigmi*. McGraw-Hill, 2005.

[6] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Automi, linguaggi e calcolabilità*. Addison-Wesley Italia, 2003.

[7] C. Laneve. *La Descrizione Operazionale dei Linguaggi di Programmazione*. Franco Angeli, 1998.

[8] T. W. Pratt and M. V. Zelkowitz. *Programming languages: Design and Implementation*. Prentice-Hall, 2001.

[9] H. Rogers. *Teoria delle funzioni ricorsive e della calcolabilità effettiva*. Tecniche Nuove, 1992.

[10] A. S. Tannenbaum. *Structured Computer Organization*. Prentice-Hall, 1999.

[11] G. Winskel. *La semantica formale dei linguaggi di programmazione*. UTET libreria, 1999.

Elenco delle figure

1.1 Dal problema all'algoritmo. 4
1.2 Esecuzione di 2×3 con l'algoritmo di *moltiplicazione per somme*. 7
1.3 Diagramma a blocchi che descrive l'algoritmo di moltiplicazione per somme. 12
1.4 Esecuzione di 3×0 (caso (i)) e 0×3 (caso (ii)) con l'algoritmo di *moltiplicazione per somme*. 13
1.5 Algoritmo ottimizzato di moltiplicazione per somme. 14
1.6 Diagramma di flusso per il calcolo della media di 3 numeri interi. 25
1.7 Schema di un ambiente di programmazione. 27

Elenco delle tabelle

2.1 Operatori logici. 38
2.2 Tavole di verità degli operatori logici NOT, AND, OR. 38
2.3 Precedenza degli operatori in C++ (parziale). 43