Capitolo 2

Elementi di base di un programma

2.1 Identificatori

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

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

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

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

Identificatori C . Un identificatore in C è una sequenza di caratteri alfabetici e numerici, iniziante obbligatoriamente con un carattere alfabetico, e non contenente alcun carattere speciale, come ad esempio "+", ">", "!", spazio, ..., ad eccezione del carattere "_" (underscore). □

Esempio 2.1 (Identificatori)

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

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

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

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

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

Ciascun identificatore può essere associato ad un solo oggetto all'interno di uno stesso "ambiente". Che cosa si intenda per "ambiente" e come si determinano le associazioni tra identificatori ed oggetti in ciascun punto di un programma verrà precisato nel Capitolo 3.10, dedicato alle cosiddette *"regole di scope"*.

2.2 Variabili

Variabile. Una *variabile* in un linguaggio di programmazione convenzionale è un'astrazione di un "contenitore" (specificatamente, di una cella di memoria) ed è caratterizzata da:

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

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

[1] Come vedremo meglio in seguito, non tutte le variabili hanno necessariamente un nome e, nel caso in cui ce l'abbiano, questo può non essere unico. Specificatamente, una variabile allocata dinamicamente (vedi Capitolo 7) non ha un nome, ma ad essa ci si può riferire direttamente tramite il suo indirizzo, memorizzato in un *puntatore*. Viceversa, una variabile passata come parametro per riferimento ad una funzione (vedi par. 5.5) avrà più di un nome associato ad essa.

L'operazione di scrittura della variabile (ad esempio un'istruzione di assegnamento—vedi par. 2.4) permette di modificare il valore della variabile. L'operazione è distruttiva, nel senso che il nuovo valore sostituisce quello attuale, che quindi è definitivamente perso. Viceversa, un'operazione di lettura di una variabile restituisce il suo valore attuale senza modificarlo.

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

Prima di poter essere utilizzata, una variabile deve essere creata.

Creazione e distruzione di una variabile. La *creazione di una variabile* di tipo t consiste nell'allocazione dello spazio di memoria necessario a contenere un dato di tipo t e nella sua eventualmente inizializzazione. L'operazione simmetrica alla creazione è quella di *distruzione* di una variabile e comporta la *deallocazione* della memoria principale ad essa riservata. Il tempo che intercorre tra la creazione di una variabile e la sua distruzione è detto *tempo di vita* della variabile. □

Si osservi che la quantità di memoria allocata per una variabile dipende dal tipo associato alla variabile stessa. Ad esempio, tipicamente, 4 byte per variabili di tipo intero, 1 byte per variabili di tipo carattere, ecc.

Una variabile può essere creata tramite l'elaborazione di una *dichiarazione di variabile* o tramite un'apposita istruzione di allocazione della memoria, come l'istruzione `new` in C++. Per ora ci limiteremo al caso di variabili create tramite dichiarazione, rimandando la creazione tramite **new** (variabili *dinamiche*) ad un successivo capitolo, in cui ritorneremo più approfonditamente anche sulle modalità di creazione e distruzione e sulla nozione di tempo di vita di una variabile.

Dichiarazione di variabile. Una *dichiarazione di variabile* è un costrutto linguistico che permette di specificare le caratteristiche essenziali di una variabile: nome, tipo ed un eventuale valore iniziale. L'*elaborazione* di una dichiarazione di variabile comporta:

- la creazione della variabile (ovvero, quello che nello studio della semantica dei linguaggi di programmazione viene indicato come l'*oggetto denotabile*), con la conseguente allocazione dello spazio di memoria necessario
- la creazione dell'associazione (*"binding"*) tra la variabile ed il suo nome. □

Dalla dichiarazione in poi, la variabile può essere utilizzata, in lettura o in scrittura, tramite il suo nome. Nella maggior parte dei linguaggi di programmazione convenzionali (compreso il C++) utilizzare una variabile senza averla prima esplicitamente dichiarata viene considerato un errore, normalmente individuato a tempo di compilazione.

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

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

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

dove:

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

Il significato di questo costrutto è la creazione di n variabili, di nome $v_1,\ldots,v_n$, tutte di tipo t, e la loro eventuale inizializzazione con i valori risultanti dalla valutazione, rispettivamente, di $i_1,\ldots,i_n$. □

Esempio 2.2 (Dichiarazioni di variabili)

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

Notare che

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

è del tutto equivalente a

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

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

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

In molti linguaggi (ad esempio il Pascal) le dichiarazioni di variabile possono essere poste soltanto in punti precisi del programma, tipicamente all'inizio del programma principale o dei sottoprogrammi. In C++ invece una dichiarazione di variabile può essere posta in qualsiasi punto del programma dove possa stare uno statement. Ad esempio, nel programma introduttivo presentato nel sottocapitolo 1.4.2, 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.

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

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

Una nozione importante associata alla dichiarazione di variabile è quella di *"campo d'azione"* (o *"scope"*) di una dichiarazione, definito come la porzione di programma in cui rimane attiva l'associazione nome–variabile stabilita dalla dichiarazione stessa. In presenza di regole di "scope" statiche, come nel caso del C++, lo "scope" di una dichiarazione di variabile è determinato dalla struttura (statica) del programma e solitamente, ma con varie, precise eccezioni, coincide con il tempo di vita della variabile stessa. Ritorneremo più approfonditamente sul concetto di "scope" di una variabile, con particolare riferimento al linguaggio C++, nel Capitolo 3.10.

2.3 Tipi di dato primitivi

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

In generale in un linguaggio di programmazione si distinguono tra *tipi semplici* e *tipi strutturati*, e tra *tipi primitivi* e *tipi definiti dall'utente*. In

questo capitolo tratteremo dei tipi semplici primitivi, con particolare riferimento a quelli presenti in C++. In successivi capitoli tratteremo di tipi strutturati e di tipi definiti da utente.

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

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

2.3.1 Il tipo `int`

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

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

I valori del tipo `int` sono denotati da *costanti intere* che possono avere la forma standard della rappresentazione decimale dei numeri (sequenza di cifre decimali con eventuale segno—ad esempio, `12`, `+34`, `-5`) oppure rappresentare un valore intero in base 8 o in base 16—ad esempio, `012` per il numero 12 in base 8 (ovvero 10 in base 10) e `0x1A` per il numero 1A in base 16 (ovvero 26 in base 10).

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

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

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

2.3.2 Il tipo `float`

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

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

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

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

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

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

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

Ritorneremo nella seconda parte del testo sul concetto di sottotipo, in connessione con la nozione di ereditarietà. Vedremo invece altri casi di sottotipo e supertipo per altri tipi semplici primitivi nel paragrafo 2.3.5.

Nota. L'utilizzo del nome `float` (invece del più significativo `real` utilizzato da altri linguaggi, quali il Pascal) è un'altra "concessione" fatta dai progettisti del C alla "visibilità" dell'implementazione sottostante. `float` si riferisce infatti alla tecnica "floating point"

tipicamente adottata per rappresentare i numeri con virgola all'interno del calcolatore. Anche il termine `double` fa riferimento ad una tecnica di implementazione, quella della rappresentazione dei numeri reali in "doppia precisione".

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

2.3.3 Il tipo `char`

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

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

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

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

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

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

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

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

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

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

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

2.3.4 Il tipo `bool`

inserisci #include <stdbool.h>

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

```
bool b1 = false;
```

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

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

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

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

Tabella 2.1: Operatori logici.

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

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

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

Tramite le tavole di verità è possibile ricavare il significato di espressioni booleane qualsiasi a partire da quello delle espressioni componenti.

Esempio 2.3 (Tavola di verità) *Data l'espressione booleana* a `||` `!` b *la sua tavola di verità è la seguente:*

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

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

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

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

2.3.5 Modificatori di tipo

È possibile far precedere ai tipi di dato semplici primitivi, escluso il `bool`, dei *modificatori* in modo da alterare alcune caratteristiche del tipo e poterlo adattare in modo più preciso a diverse situazioni. Ci sono quattro modificatori: `long`, `short`, `signed` e `unsigned`.

`long` e `short` modificano il massimo ed il minimo valore che il dato può contenere. La gerarchia per quanto riguarda la dimensione di tipi interi è `short int`, `int` e `long int`; per i numeri in virgola mobile è `float`, `double` e `long double`. "long float" non è un tipo legale e non esistono numeri in virgola mobile `short`.

I modificatori `signed` o `unsigned` indicano al compilatore come usare il bit di segno (il bit più a sinistra) per tipi interi e caratteri (i numeri in virgola mobile sono sempre `signed`). Un numero `unsigned` non tiene traccia del segno e quindi ha un bit in più disponibile. Questo significa che può immagazzinare numeri il doppio più grandi di quanto possa fare un numero `signed`. Per esempio un `signed char` a 8 bit può rappresentare i valori da $-2^7 = -128$ a $2^7 - 1 = 127$; un `unsigned char` da 0 a $2^8 - 1 = 255$. Per interi e caratteri, di default è assunto che il valore sia dotato di segno.

Il seguente esempio mostra la dimensione in byte di diversi tipi di dati ottenuta utilizzando l'operatore `sizeof()`.

```
#include <iostream>
using namespace std;
int main(){
  cout << "char           " << sizeof(char) << endl;
  cout << "unsigned char  " << sizeof(unsigned char) << endl;
  cout << "short          " << sizeof(short) << endl;
  cout << "unsigned short " << sizeof(unsigned short) <<endl;
  cout << "int            " << sizeof(int) << endl;
  cout << "unsigned int   " << sizeof(unsigned int) << endl;
  cout << "long           " << sizeof(long) << endl;
  cout << "unsigned  long " << sizeof(unsigned long) << endl;
  cout << "float          " << sizeof(float) << endl;
  cout << "double         " << sizeof(double) << endl;
  cout << "long double    " << sizeof(long double) << endl;
  cout << "bool           " << sizeof(bool) << endl;
  return 0;
}
```

L'output prodotto dal programma è:

```
char           1
unsigned char  1
short          2
unsigned short 2
int            4
unsigned int   4
long           4
unsigned  long 4
float          4
double         8
long double    12
bool           1
```

2.4 Statement di assegnamento

Introduciamo in questo sottocapitolo uno statement fondamentale per tutti i linguaggi di programmazione convenzionali, lo statement di assegnamento. Come al solito faremo riferimento al linguaggio C++, ma le considerazioni che verranno fatte sono del tutto generali. Gli altri statement per realizzare

strutture di controllo diverse (precisamente, gli statement di selezione, di iterazione e di salto) saranno esaminati nel capitolo successivo.

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

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

dove:

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

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

Esempio 2.4 (Statement di assegnamento in C++)

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

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

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

Parte di queste note, nel capitolo dedicato alla "ridefinizione" dell'operatore di assegnamento.

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

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

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

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

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

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

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

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

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

2.5 Espressioni ed operatori

Una nozione fondamentale presente in qualsiasi linguaggio di programmazione è quella di espressione. 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++) che riportiamo qui di seguito.

Sintassi delle espressioni. Un'espressione è un'entità sintattica che può assumere una delle seguenti forme:

- una costante—ad esempio, `1`, `3.5`, `'a'`, `"ciao"`;
- una variabile—ad esempio, `x`, `contatore`;
- 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*—ad esempio, `x + 1`, `x > y - 1`, `a && b || c`;
- un'espressione composta della forma

  $$op \ expr$$

  dove *op* è un operatore (unario) prefisso—ad esempio, `!a`, `++x`;
- un'espressione *expr* tra parentesi

  $$\texttt{(}expr\texttt{)}$$

  ad esempio, `x - (y - 3)`;
- una chiamata di funzione

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

  ad esempio, `fatt(n)`, `pow(x,y)`. □

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

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

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

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

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

2.5.1 Valutazione di una espressione

Valutare un'espressione significa determinarne, se possibile, il valore.[3]

Valore di un'espressione. Il valore di un'espressione costante c è la costante c stessa. Il valore di un'espressione variabile x è il valore contenuto nella variabile x. Il valore di un'espressione composta, e_1 `op` e_2, `op` e_1, `f`$(e_1, e_2, \ldots, e_n)$, è il valore ottenuto applicando l'operatore `op` (o la funzione `f`) ai valori delle espressioni e_1, e_2, ..., e_n. □

Si noti che la valutazione di un'espressione composta avviene valutando prima le sotto-espressioni e_1, ..., e_n e poi l'espressione stessa. Ad esempio, data l'espressione `x + 5`, con `x` che supponiamo valga `3`, si valuta prima `x`, ottenendo il valore 3, poi si valuta `5` ottenendo il valore 5, poi si valuta `+`, applicato a 3 e 5, ottenendo il valore 8.

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 `&&` sui booleani come l'AND logico, l'operatore `==` come l'uguaglianza, ecc..

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

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

Precedenza più alta	! ++ −−
	* / %
	+ −
	< > <= >=
	&&
	||
Precedenza più bassa	=

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

Esempio 2.5 *Consideriamo l'espressione*

[3] Un'espressione è una funzione parziale e come tale produce sempre un risultato esplicito per tutti i valori del dominio per cui essa è definita. Un'espressione può essere indefinita per alcuni valori del suo dominio—ad esempio `1 / x` è chiaramente indefinita per `x` uguale a 0. In questo caso la sua valutazione non terminerà con un risultato esplicito, ma, tipicamente, darà origine ad un errore a *"run-time"*.

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

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

Esempio 2.6 *Consideriamo l'espressione*

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

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

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

Esempio 2.7 *La valutazione dell'espressione*

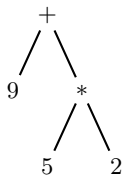
```
(9 + 5) * 2
```

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

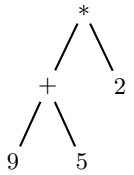
In generale, dunque, la presenza di opportune regole di precedenza ci permette di ridurre l'uso esplicito delle parentesi, semplificando la scrittura dei programmi. Ad esempio, nell'espressione dell'Esempio 2.6 abbiamo potuto evitare le parentesi attorno alle due sottoespressioni di confronto, grazie 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 (Alberi sintattici). Le espressioni possono essere convenientemente rappresentate sotto forma di alberi, detti *alberi sintattici*. Senza addentrarci in una definizione precisa di questa nozione, ne diamo qui un'idea intuitiva tramite alcuni esempi (per un trattamento preciso e completo dell'argomento si rimanda, ad esempio, a [1]).

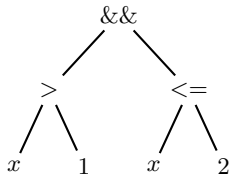
L'espressione dell'Esempio 2.5 può essere rappresentata con l'albero sintattico:



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



Così, ad esempio, l'espressione dell'Esempio 2.7 viene rappresentata dall'albero sintattico:



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

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

Esempio 2.8 (Associatività)

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

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

Si noti che per gli operatori binari infissi della forma $expr_1$ *op* $expr_2$, la definizione del linguaggio C++ non specifica in modo esplicito quale delle due sottoespressioni, $expr_1$ o $expr_2$, venga valutata per prima.[4] Questa decisione è lasciata alla realizzazione dello specifico compilatore e potrebbe quindi portare a comportamenti diversi in diversi compilatori. Nella maggior parte

[4] 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.

dei casi, comunque, i compilatori scelgono di procedere sempre da sinistra verso destra, e cioè di valutare per prima la sottoespressione $expr_1$ e poi la sottoespressione $expr_2$.

2.5.2 Tipo di un'espressione

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

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

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

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

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

`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`.[5]

Per le funzioni definite dall'utente il tipo del risultato della funzione è quello specificato nella dichiarazione della funzione stessa (si veda il Capitolo 5).

[5] Si noti che nel primo esempio di programma C++ mostrato nel sottocapitolo 1.4.2 abbiamo usato la costante reale `3.0` (invece che la costante intera `3`) nell'espressione `(x + y + z) / 3.0` proprio per "forzare" l'espressione ad essere di tipo `float`; in questo modo, infatti, l'espressione `(x + y + z)` che è di tipo `int` viene "promossa" a `float` e si applica l'operatore `/` per `float` (applicando `/` per `int` si sarebbe avuto un troncamento del risultato della divisione).

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

2.5.3 Espressioni booleane

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

- espressioni booleane *semplici*, come per esempio

  ```
  x > y + 1
  ```

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

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

  oppure

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

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

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

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

può essere scritta equivalentemente come

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

Abbiamo cioè applicato la *proprietà distributiva*:

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

dove p, q ed r rappresentano generiche espressioni booleane.

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

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

Ad esempio, nella valutazione dell'espressione

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

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

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

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

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

2.5.4 Espressioni condizionali

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

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

dove

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

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

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

```
#include <iostream>
using namespace std;
int main() {
  cout << "Inserisci un numero" << endl;
  int x;
  cin >> x;
  int abs_x = (x >= 0) ? x : -x;
  cout << "Il valore assoluto di " << x << " e' "
       << abs_x << "." << endl;
  return 0;
}
```

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

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

2.6 Ancora sullo statement di assegnamento

Lo statement di assegnamento agisce, di base, tramite un *effetto collaterale* (o *side effect*): la sua esecuzione modifica in modo permanente il contenuto di una locazione di memoria, precisamente il valore della variabile che appare alla sua destra[6]. In C++ (ma non in altri linguaggi, come ad esempio il Pascal) l'assegnamento non si limita a produrre un effetto collaterale, ma può essere usato anche come un'espressione che restituisce un risultato esplicito. Precisamente, lo statement di assegnamento *var* `=` *expr* restituisce come suo risultato il valore dell'espressione *expr* assegnato alla variabile *var*.

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

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

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

Un'ultima osservazione sull'operatore di assegnamento riguarda i tipi dei suoi argomenti. Dato un assegnamento *var* `=` *expr*, il tipo di *var* e il tipo di *expr* devono essere *compatibili*. Questo è senz'altro vero se il tipo di *var* e quello di *expr* sono coincidenti. Ad esempio, entrambi `int` o entrambi `float`. Nel caso dei tipi semplici primitivi, però, è possibile anche che i tipi delle due parti dell'assegnamento non siano coincidenti: in questo caso viene applicata una regola di *conversione automatica* che prevede che il tipo dell'espressione a destra dell'"`=`" sia convertito in quello della variabile a sinistra dell'"`=`". Ad esempio, il codice seguente è corretto:

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

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

assegnato e quindi in generale una perdita di informazione solitamente segnalata dai compilatori con un *warning* (in questo esempio, il valore assegnato ad `i` sarà 1).

2.6.1 Altri operatori di assegnamento

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

- Operatori della forma

  op`=`

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

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

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

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

  Ad esempio:

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

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

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

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

- Operatori di autoincremento e autodecremento:

  `++` e `--`.

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

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

  è:

  l-expr `=` *l-expr* `+ 1;`

  e, analogamente, il significato degli statement:

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

  è:

  l-expr `=` *l-expr* `- 1;`

  Ad esempio:

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

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

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

ma anche

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

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

Come altro esempio, il seguente frammento di programma

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

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

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

stampa `10` e poi `11`.

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

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

2.7 Costanti con nome

Nei linguaggi di programmazione è solitamente possibile associare un nome simbolico ad un valore costante e quindi usare il nome al posto del valore all'interno del programma. Un modo per ottenere questo è tramite *dichiarazioni di costante*. In C++ la dichiarazione di costante con nome ha la seguente forma sintattica di base:

```
const t c = e;
```

dove: `t`: tipo qualsiasi
`c`: identificatore (= nome della costante)
`e`: espressione costante di tipo t (= valore della costante)

Il suo significato è: dichiara una costante di tipo `t`, di nome `c`, con valore (il risultato della valutazione dell'espressione) `e`.

Esempio 2.9

```
const int num_elem = 10;          // costante di nome
                                  // num_elem e valore 10
const float pi_greco = 3.14;      // costante di nome
                                  // pi_greco e valore 3.14
```

Dunque la forma sintattica della dichiarazione di costante è simile a quella della dichiarazione di variabile con inizializzazione. La sua semantica però, è profondamente diversa: una dichiarazione di costante infatti introduce soltanto un'associazione tra un nome ed un valore, senza richiedere la presenza di una locazione di memoria cui legare il nome e in cui memorizzare il valore corrente, come invece avviene nel caso delle variabili.

Una dichiarazione di costante può essere inserita ovunque in un programma (anche nella parte delle dichiarazioni globali, come spesso avviene). La visibilità dei nomi di costanti è determinata in base alle normali regole di "scope" previste per le variabili e per tutti gli altri nomi. In particolare una costante potrà essere usata dalla sua dichiarazione in avanti (non prima) e non sarà visibile in un blocco più esterno a quello in cui è dichiarata.

Una volta dichiarato, il nome di una costante può essere usato ovunque possa apparire un valore costante.

Esempio 2.10

```
const int num_elem = 10;
int main() {
     ...
     int A[num_elem];
     if (i > num_elem) ...;
```

ma

```
num_elem = 12;
```

viene segnalato come errore in quanto una costante non può apparire nella parte sinistra di uno statement di assegnamento.

Vantaggi ad usare costanti con nome al posto di valori costanti:

- maggior leggibilità del programma
- maggior modificabilità del programma: un'eventuale modifica del valore della costante circoscritta alla sola dichiarazione della costante (l'uso resta invariato).

Vantaggi ad usare costanti con nome al posto di variabili):

- maggior affidabilità del programma: evita modifiche non volute

- possibile maggior efficienza del codice generato: il compilatore può evitare l'allocazione di memoria per contenere le costanti (usando ad es. istruzioni con operando immediato), oppure le può allocare nella parte della memoria statica (invece che sullo stack).

Nota Il C++ ammette che nella dichiarazione

```
const t c = e;
```

l'espressione `e` possa essere un'espressione qualsiasi, anche contenente variabili (il cui valore può essere determinato in generale solo a run-time).

A rigore, in questo caso, la dichiarazione `const` non introduce una costante con nome, quanto piuttosto *una variabile di sola lettura* (o *variabile costante*). Si tratta a tutti gli effetti di una variabile, ma il compilatore controlla che non possa essere modificata (e quindi, ad esempio, non può apparire nella parte sinistra di un assegnamento). Può essere usata come una costante con nome, ma la sua dichiarazione, nel caso in cui l'espressione `e` contenga variabili, non può apparire nella parte delle dichiarazioni globali di un programma.

In C++ è possibile associare un nome simbolico ad un valore anche tramite la direttiva di preprocessore `#define`. Ad esempio, le due direttive:

```
#define NUM_ELEM 10
#define PI_GRECO 3.14
```

associano i nomi `NUM_ELEM` e `PI_GRECO` rispettivamente ai valori `10` e `3.14`.

La direttiva `#define` è concettualmente molto diversa dalla dichiarazione `const`, anche se l'uso può essere simile. Infatti, la direttiva

```
#define s1 s2
```

viene trattata in fase di preprocessing e comporta una semplice sostituzione di stringhe nel programma sorgente: tutte le occorrenze della stringa `s1` vengono rimpiazzate dalla stringa `s2`. Dunque è una trasformazione da programma sorgente a programma sorgente. `s1` e `s2` non sono identificatori, ma stringhe qualsiasi (senza alcun tipo associato). La visibilità dei nomi introdotti non è determinata dalle regole di "scope", ma si applica all'intero programma, dalla direttiva in poi.

2.8 Input/output di base

Abbiamo già visto nel Capitolo 1.1 che, tipicamente, un algoritmo riceve dei dati su cui operare (dati in ingresso o *input*) e produce dei risultati (dati in uscita o *output*). Tutti i linguaggi di programmazione forniscono perciò opportuni strumenti per esprimere operazioni di input/output (o I/O). Questi strumenti possono essere istruzioni primitive del linguaggio (come la `read` e la `write` del Pascal) oppure sottoprogrammi predefiniti facenti parte dell'ambiente di programmazione standard del linguaggio (si veda il Capitolo 1.5).

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

La forma standard di input/output utilizzata in C++ è quella basata su *stream*, un tipo di dato astratto predefinito che rappresenta un *canale di comunicazione* tra un programma ed una sorgente/destinazione di dati. Quest'ultima puó essere sia un dispositivo esterno di I/O (ad esempio la tastiera o il monitor), sia un generico file. In questo capitolo esamineremo soltanto le possibilità di base offerte dal C++ per l'input/output su stream, in particolare per la comunicazione con i dispositivi standard di I/O (tastiera e monitor). Ritorneremo, in modo piú approfondito e generale, sulla nozione di stream nel Capitolo 6 dove esamineremo l'input/output su file.

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

#include <stdio.h> #include <stdlib.h>

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

- scanf() **s**tream di input, associato di default alla tastiera;
- printf() stream di output, associato di default al monitor.

.

2.8.1 Lettura da stream tramite >>

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

```
s >> v
```

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

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

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

Esempio 2.11 *Se si esegue*

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

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

2.8.3 Lettura e scrittura di caratteri

in C++ sono presenti anche funzioni per la lettura e scrittura di singoli caratteri: funzioni `get` e `put`.

Lettura di caratteri (funzione `get`)

`s.get()`

legge (= estrae) dallo stream di input `s` il carattere corrente, se presente, e lo restituisce come suo risultato. Se il dato non è presente, allora attende.

Esempio 2.16 *Date le seguenti istruzioni:*

```
char c;
c = cin.get();
```

con input (std input = `cin`*)*

```
abc
```

la loro esecuzione assegna a `c` *il carattere* `'a'` *e lascia sullo stream* `cin` *i caratteri* `'b'` *e* `'c'`*. Successive* `get` *estrarranno i caratteri successivi:*

```
c = cin.get(); //assegna 'b' a c
c = cin.get(); //assegna 'c' a c
```

Nell'ipotesi che la sequenza `abc` *sia seguita da un "a capo", una successiva istruzione*

```
c = cin.get();
```

assegnerà `'\n'` *("a capo") a* `c`*.*

Si osservi che un singolo carattere può essere letto anche tramite l'operatore di estrazione `>>` applicato ad una variabile di tipo `char`. A differenza della `get`, però, l'operatore `>>` ignora i caratteri "spazio" e "a capo" eventualmente presenti sullo stream di input (li estrae, ma li scarta).

Esempio 2.17 *L'esecuzione di*

```
char x,y,z;
scanf(" %c",&x);
scanf(" %c",&y);
scanf(" %c",&z);
```

con input a,b,c

assegna `'a'` *a* `x`*,* `'b'` *a* `y` *e* `'c'` *a* `z`*,*

2.9 *Domande per il Capitolo 2*

1. Dire cosa è e da cosa è caratterizzata in generale una variabile in un linguaggio di programmazione convenzionale.
2. Quali informazioni fornisce una dichiarazione di variabile? Cosa provoca la sua elaborazione?
3. Qual è la forma generale di una dichiarazione di variabile in C++?
4. Che forma sintattica può avere un identificatore del C++?
5. Dove possono essere poste le dichiarazioni di variabili in un programma C++?
6. Qual è il valore di una variabile introdotta con una dichiarazione senza inizializzazione?
7. Qual è il valore stampato sullo standard output dall'esecuzione del seguente frammento di programma C++:

   ```
   float x;

   cout << x + 1;
   ```

   Motivare la risposta.
8. Qual è la semantica informale dellistruzione di assegnamento $v = E$? (in particolare, cosa sono v ed E?).
9. Quali sono le differenze tra la parte sinistra e la parte destra di un'istruzione di assegnamento?

10. Qual è il risultato restituito da $v = E$ quando usata come espressione in C++? Dare almeno un esempio d'uso dell'assegnamento come espressione.
11. Cosa sono e come si dichiarano in C++ le costanti con nome?
12. Indicare almeno un vantaggio ad usare costanti con nome al posto di valori costanti e almeno un vantaggio ad usare costanti con nome al posto di variabili (modificabili).
13. Dare la definizione (generale) di tipo di dato.
14. Cosa significa che un tipo è semplice e primitivo? Quali sono i tipi semplici primitivi del C++?
15. Indicare quali sono i possibili valori e le possibili operazioni primitive per il tipo `int` del C++.
16. Indicare quali sono i possibili valori e le possibili operazioni primitive per il tipo `float` del C++. Qual è la forma sintattica di un numero di tipo `float` in C++?
17. In cosa differiscono il tipo `float` e il tipo `double`?
18. Indicare quali sono i possibili valori e le possibili operazioni primitive per il tipo `char` del C++. In che senso i caratteri in C++ possono essere visti come "interi piccoli" e viceversa?
19. Indicare quali sono i possibili valori e le possibili operazioni primitive per il tipo `bool` del C++.
20. Dare la tavola di verità dell'espressione booleana `a || !b` (dove `a` e `b` sono espressioni booleane qualsiasi).
21. Per quali valori di x le seguenti espressioni booleane risultano vere?
    (a) `(x>3 && x<10) && x<0`
    (b) `(x>3 || x<10) || x<0`
    (c) `(x<3 && x>10) || x<0`
22. Trasformare lespressione booleana `(x == 1) && (y < 1.5 || z > 1)` in un'espressione equivalente, applicando la proprietà distributiva.
23. Descrivere (in modo informale, ma preciso) la sintassi di un'espressione in C++ (casi base).
24. Cosa si intende con i termini infisso, prefisso e postfisso riferiti ad un operatore?
25. A cosa serve le precedenza tra gli operatori? Illustrare con esempi.
26. A cosa serve l'associatività degli operatori? Illustrare con esempi.
27. Cosa si intende per tipo di un'espressione?

28. Data la seguente dichiarazione C++:

    ```
    int x=2, y=5;
    ```

    qual è il tipo e qual è il valore delle seguenti due espressioni:

    (a) `(x + y)/2;`

    (b) `(x + y)/2.0;`

    Motivare precisamente la risposta.

29. Cosa prevedono le regole di conversione automatica in C++ quando un'espressione aritmetica non ha tutti gli operandi dello stesso tipo semplice primitivo?

30. Date le seguenti dichiarazioni C++:

    ```
    float x = 3.7;
    int y = 5;
    ```

    quali sono i valori stampati nei due casi seguenti:

    (a) `x = y; cout << x;`

    (b) `y = x; cout << y;`

    Motivare precisamente la risposta.

31. Descrivere le principali differenze tra le funzioni `>>` e `get()` applicate allo stream di input `cin`. Illustrare anche con esempi.