2. Tipi di dato primitivi

Sommario:

- Tipi di dato
- Tipi di dato interi, reali e caratteri
- Precisione nelle rappresentazioni
- Espressioni
- Insieme dei valori rappresentabili
- Conversioni di tipo

2.1. Tipi di dato e allocazione di memoria

I tipi di dato vengono usati nelle dichiarazioni di variabili e costanti per determinare quali valori esse possono assumere e quali operazioni possono essere effettuate su di esse. Ad esempio, l'istruzione `int i;` dichiara la variabile `i`, indicando che essa può assumere (solo) valori interi. Come si è visto, quando si dichiara una variabile, viene allocata memoria per memorizzarne il valore assegnato. La quantità di memoria che deve essere allocata dipende dal tipo associato alla variabile. Ad esempio, nel caso di variabili di tipo `int`, vengono tipicamente allocati 32 bit (4 byte). Com'è noto, la dimensione della memoria influisce sull'intervallo di valori rappresentabili.

Per trattare le informazioni di tipo numerico, il C definisce tre tipi primitivi:

- `int`, usati per rappresentare numeri interi;

- `float`, usati per la rappresentazione di reali in virgola mobile;

- `double`, usati per la rappresentazione di reali in virgola mobile con doppia precisione.

Per trattare caratteri alfanumerici e simboli speciali, il C offre il tipo di dato:

- `char`.

Il C fornisce inoltre i qualificatori `short`, `long`, `signed`, `unsigned` che consentono di modificare la dimensione (in bit) del formato di alcuni tipi e la rispettiva interpretazione.

Il C non mette a disposizione un tipo primitivo booleano, ma sfrutta gli interi, interpretando il valore 0 come `false` e qualunque valore non nullo come `true`. Inoltre, alcune versioni del C mettono a disposizione un tipo booleano (ad es., `bool` o `_Bool` in C99). Tale tipo, tuttavia non è altro che una ridefinizione del tipo intero.

2.1.1. Il tipo di dato primitivo `int`

Rappresentiamo le caratteristiche più rilevanti dei tipi di dato primitivi riportando in una tabella le seguenti informazioni:

- dimensione della rappresentazione[1] , ovvero numero di bit allocati per memorizzare il contenuto della variabile;

- dominio del tipo, ovvero l'insieme dei valori rappresentati dal tipo;

- operazioni comuni disponibili sugli elementi del dominio;

- letterali, ovvero gli elementi sintattici del C che denotano valori del dominio (ad esempio `23`).

[1] In alcuni tipi, questa grandezza dipende dalla macchina e/o dal compilatore. Noi faremo riferimento alle combinazioni più comuni nei calcolatori moderni.

Tipo	`int`	
Dimensione	32 bit (4 byte)	
Dominio	numeri interi in $[-2^{31}, +2^{31}-1]$ (oltre 4 miliardi di valori)	
Operazioni	`+`	somma
	`-`	sottrazione
	`*`	prodotto
	`/`	divisione intera
	`%`	resto della divisiona intera
Letterali	sequenze di cifre che denotano valori del dominio (es. 275930)	

Esempio 2.1.

```
int a,b,c;  // Dichiarazione di variabile di tipo int
a = 1;      // Uso di letterali
b = 2;
c = a + b;  // Espressione aritmetica che coinvolge
            // operatori del linguaggio
```

I valori limite del tipo `int` sono definiti nelle costanti `INT_MIN` e `INT_MAX` nel file di sistema `limits.h`

2.1.1.1. Overflow numerico

Poiché l'insieme dei valori interi rappresentabili mediante un tipo primitivo è limitato ad un dato intervallo (ad esempio, [-2^{31},2^{31}-1] per il tipo `int`), applicando operatori aritmetici a valori di tipo intero si può verificare il cosiddetto **overflow numerico** (trabocco). Cioò avviene quando il risultato dell'operazione non può essere rappresentato con il numero di bit messi a disposizione dal tipo.

Esempio 2.2.

```
int x = INT_MAX;
printf("%d\n",x); // Stampa 2147483647
                  // (massimo valore rappresentabile con int)
x++; // Incrementa x di 1
printf("%d\n",x); // Stampa -2147483648
                  // (minimo valore rappresentabile con int)
```

L'overflow nell'esempio è dovuto al fatto che il valore 2147483647 equivale, in rappresentazione binaria, a $0\underbrace{1\ldots1}_{31}$. Incrementando tale valore di 1, si ottiene $1\underbrace{0\ldots0}_{31}$ (si noti il bit di segno pari ad 1) che, in complemento a due, rappresenta appunto -2147483648. Considerazioni analoghe valgono nel caso in cui il valore iniziale sia decrementato.

Si noti che, in generale, il comportamento conseguente ad un overflow dipende dal compilatore. Inoltre, linguaggi diversi possono trattare gli overflow in maniera diversa, o addirittura prevedere meccanismi che lo impediscono. Ad esempio, il Python, quando necessario, sfrutta una rappresentazione degli interi più flessibile che permette di codificare valori arbitrari.

2.1.2. I qualificatori `short`, `long` e `unsigned`

I qualificatori `short`, `long` e `unsigned`, permettono di modificare la dimensione del formato di alcuni tipi e la relativa interpretazione, specificando di fatto nuovi tipi. Ciascuno di essi può essere usato in combinazione con `int` (ma non esclusivamente).

Il qualificatore `short`, associato al tipo `int`, viene usato per indicare il tipo intero in un intervallo di valori ridotto.

Tipo	`short int`
Dimensione	16 bit (2 byte)
Dominio	numeri interi in $[-2^{15}, +2^{15}-1] =$ [-32768,+32767]

Esempio 2.3.

```
short int a,b; // Dichiarazione di variabile di tipo short
a = 22700; // Uso di letterali
```

Il qualificatore `long`, associato al tipo `int`, viene usato per indicare il tipo intero in un intervallo di valori esteso.

Tipo	`long int`
Dimensione	64 bit (8 byte)
Dominio	numeri interi in $[-2^{63}, +2^{63}-1]$

Esempio 2.4.

```
long a,b;         // Dichiarazione di variabili di tipo long
a = 9000000000L; // Uso di letterali
b = a + 2L;       // Espressione aritm. che coinvolge
                  // operatori del linguaggio
```

Il tipo `long int` può essere a 32 o a 64 bit a seconda del compilatore usato e dell'architettura del calcolatore. Nel caso di 32 bit è equivalente al tipo `int`.

Il qualificatore `unsigned` indica che il primo bit del vettore non deve essere considerato come bit di segno.

Tipo	`unsigned int`
Dimensione	32 bit (4 byte)
Dominio	numeri interi in $[0, +2^{32} - 1]$

Esempio 2.5. L'istruzione: `unsigned int a;` dichiara la variabile `a` di tipo *intero senza segno*.

Si noti che, non dovendo usare il primo bit per rappresentare il segno, il valore massimo rappresentabile con un `unsigned int` è maggiore (doppio) rispetto a quello rappresentabile con un `int`.

Il qualificatore `unsigned` può essere combinato con `short` e `long` (in qualunque ordine), quando questi siano applicati al tipo `int`. Il numero di valori rappresentabili cambia di conseguenza, per effetto dell'eliminazione del segno.

Esempio 2.6. L'istruzione `unsigned short int x;` definisce la variabile `x` di tipo `short int` senza segno. I valori che la variabile può assumere sono nell'intervallo $[0, 2^{16} - 1]([0, 65534])$.

Quando i qualificatori appena visti sono applicati al tipo `int`, quest'ultimo può essere omesso (e tipicamente lo è). Ad esempio, la dichiarazione `unsigned int x` è equivalente ad `unsigned x` e `unsigned long int x` è equivalente ad `unsigned long x`.

Precisiamo infine che `unsigned` può essere associato anche al tipo `char` e `long` al tipo `double` (v. seguito).

2.1.3. Tipi di dato primitivi reali

Come anticipato, il C mette a disposizione due tipi di dato primitivi, `float` e `double`, per la rappresentazione dei numeri reali in virgola mobile.

2.1.3.1. Il tipo di dato `float`

Il tipo `float` identifica i numeri reali, rappresentati in virgola mobile, a precisione singola.

Tipo	`float`		
Dimensione	32 bit (4 byte)		
Dominio	2^{32} reali pos. e neg.	min	$1.4012985 \cdot 10^{-38}$
		max	$3.4028235 \cdot 10^{+38}$
		Precisione	$\sim$ 7 cifre decimali
Operazioni	`+`	somma	
	`-`	sottrazione	
	`*`	prodotto	
	`/`	divisione	
Letterali	sequenze di cifre con punto decimale terminate con una f che denotano valori del dominio (es. `3.14f`)		
	rappresentazione in notazione scientifica (es. `314E-2f`)		

Esempio 2.7.

```
float a;   // Dichiarazione di variabile di tipo float
a = 3.14f; // Uso di letterali
a *= 2f;   // Espressione aritmetica che coinvolge operatori
           // del linguaggio (equivalente a a=a*2f)
```

2.1.3.2. Il tipo di dato `double`

Il tipo `double` identifica i numeri reali, rappresentati in virgola mobile, a precisione doppia.

Tipo	`double`		
Dimensione	64 bit (8 byte)		
Dominio	2^{64} reali pos. e neg.	min	$1.79769313486231570 \cdot 10^{-308}$
		max	$2.250738585072014 \cdot 10^{+308}$
		Precisione	$\sim$ 15 cifre decimali
Operazioni	`+`	somma	
	`-`	sottrazione	
	`*`	prodotto	
	`/`	divisione	
Letterali	sequenze di cifre con punto decimale, opzionalmente terminate con `d`, che denotano valori del dominio (es. `3.14`)		
	rappresentazione in notazione scientifica (es. `314E-2`)		

Esempio 2.8.

```
double a;     // Dichiarazione di variabile di tipo double
a = 628E-2;   // Uso di letterali
a++;          // Espressione aritmetica che coinvolge operatori
              // del linguaggio (equivalente a: a=a+1.0)
```

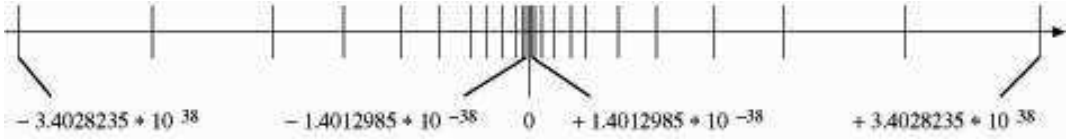
Il qualificatore `long` può anche essere applicato al tipo `double`, per identificare reali con doppia precisione estesa.

2.1.3.3. Precisione nella rappresentazione: errori di arrotondamento

In C, i numeri reali vengono rappresentati facendo uso della rappresentazione in virgola mobile (*floating point*, da cui il nome `float`), ovvero riservando un bit per il segno s, un certo numero di bit per l'esponente e ed i restanti bit per la mantissa m (il numero effettivo dei bit riservati per ogni componente dipende dal compilatore usato). Il valore del numero rappresentato è pari a $(-1)^s \cdot m \cdot 2^e$.

Come visto, il tipo `float` permette di rappresentare valori nell'intervallo $[-3.4028235 \cdot 10^{38}, +3.4028235 \cdot 10^{38}]$. Tuttavia, diversamente dagli interi, i `float` non permettono di rappresentare tutti i numeri compresi tra gli estremi dell'intervallo (considerazioni analoghe valgono anche per i `double`). Ad esempio il numero più vicino a 1222333440.0 rappresentabile in `float` è 1222333568.0, mentre non è possibile rappresentare il numero 122333441.0. Ciò è dovuto al fatto che, essendo il numero di bit per la rappresentazione della mantissa finito, la precisione dei numeri rappresentabili è necessariamente finita.

Questo aspetto è illustrato nella figura seguente: i numeri rappresentati in virgola mobile (`float` e `double`) sono tanto più vicini tra loro quanto più sono prossimi allo zero e, viceversa, tanto più lontani fra loro quanto più vicini al massimo valore rappresentabile (in valore assoluto).



Il fatto che si debba lavorare con precisione finita dà luogo ad approssimazioni nei calcoli dovute ad errori di arrotondamento.

Esempio 2.9. Si consideri il seguente frammento di codice

```
float x = 1222333444.0;
printf("x = %f\n", x);
x += 1.0;
```

```
printf("x + 1.0 = %f\n", x);
```

Quando eseguito, produce il seguente output:

```
x = 1222333440.000000
x + 1.0 = 1222333440.000000
```

Si noti che l'operazione di incremento non produce alcun effetto sul valore di `x`.

Nel seguente frammento, invece, il valore della variabile `x` cambia a seguito dell'incremento.

```
float x = 1222333440.0;
printf("x = %f\n", x);
x+=65;
printf("x + 65 = %f\n", x);
```

L'output prodotto è infatti il seguente:

```
x = 1222333440.000000
x + 65 = 1222333568.000000
```

Si noti, tuttavia, che il risultato dell'incremento è diverso dal valore atteso, ovvero 1222333505.000000. Infatti, non essendo il valore atteso rappresentabile come `float`, esso viene approssimato al valore rappresentabile ad esso più prossimo. Osserviamo infine che se avessimo incrementato la variabile `x` di `64`, l'operazione non avrebbe prodotto alcun cambiamento sul valore della variabile.

2.1.3.4. Precisione nelle misure

La precisione del risultato di un'operazione dipende dalla precisione con cui si conoscono i dati.

Ad esempio, se conosciamo le dimensioni di un rettangolo con una precisione di una sola cifra decimale, l'area non potrà avere una precisione superiore, quindi non ha senso considerare come significativa la sua seconda cifra decimale:

9.2 * 5.3 = 48.76 (la seconda cifra decimale non è significativa)
9.25 * 5.35 = 49.48 (qui lo è)

Questo non è causato dalla rappresentazione dei numeri nel linguaggi di programmazione, ma dai limiti sulla conoscenza dei valori di input di un problema.

2.1.4. Il tipo di dato primitivo char

Una variabile di tipo `char` può contenere un carattere alfanumerico o un simbolo. Il dominio del tipo `char` è normalmente costituito dai caratteri dello standard ASCII [2], contenente 128 caratteri. Il formato del tipo `char` è a 8 bit. Lo standard ASCII stabilisce una corrispondenza tra numeri e simboli alfabetici, numerici o simboli speciali (come il ritorno a capo o tab).

Ad esempio, il carattere 'A' corrisponde al codice numerico 65, il carattere 'B' al codice numerico 66, ecc. Per maggiori dettagli sullo standard ASCII, si veda ad esempio il sito web: `www.asciitable.com`.

I letterali del tipo `char` sono denotati in diversi modi; il più semplice è tramite apici singoli (`'`).

Esempio 2.10.

```
char c = 'A';
char d = '1';
```

La variabile `c` viene dichiarata di tipo `char` ed inizializzata al valore (carattere) `A`. La variabile `d` viene dichiarata di tipo `char` ed inizializzata al valore (carattere) 1.

Nel codice dello standard ASCII, le lettere alfabetiche e le cifre hanno codici consecutivi e ordinati (ad esempio, il codice del carattere 'b' è uguale a: codice del carattere 'a' + 1, il codice del carattere 'B' è uguale a: codice del carattere 'A' + 1, il codice del carattere '1' è uguale a: codice del carattere '0' + 1).

2.1.5. Il tipo `char` come tipo intero

Il tipo `char` rappresenta in realtà un tipo intero a 8 bit (i cui valori vengono interpretati come caratteri del codice ASCII). Esso permette di rappresentare i valori interi nell'intervallo $[-128, 127]$.

Esempio 2.11.

```
char c = 'A';
int i = 65;
```

[2] American Standard Code for Information Interchange.

```
printf("%c\n",i); // Stampa: A
printf("%d\n",c); // Stampa: 65
```

Le operazioni disponibili per i `char` sono le stesse disponibili per i tipi interi.

Analogamente agli interi, è possible applicare il qualificatore `unsigned` anche al tipo `char`. In questo modo si possono rappresentare gli interi nell'intervallo $[0, 255]$.

2.2. Costanti e numeri magici

Un *numero magico* è un letterale che compare nel codice senza spiegazioni. La presenza di numeri magici diminuisce la leggibilità e la modificabilità dei programmi.

Esempio 2.12.

```
int compenso = 20000 * orelavoro;
// cosa rappresenta 20000?
```

Per evitare inconvenienti dovuti all'impossibilità di interpretare i numeri magici, di regola si usano *nomi simbolici* che siano autoesplicativi. Potremmo usare una variabile, ad esempio `COMPENSO_ORARIO` a cui assegnare il valore del numero magico. In tal modo, tuttavia, potremmo accidentalmente modificarne il valore. Per impedire che ciò accada definiamo `COMPENSO_ORARIO` come una *costante*, ovvero come una variabile il cui valore non cambia:

```
const int COMPENSO_ORARIO = 20000;
int compenso = COMPENSO_ORARIO * orelavoro;
```

`COMPENSO_ORARIO` è una *costante*, cioè una variabile il cui contenuto non può variare durante l'esecuzione del programma. Questo effetto è ottenuto usando il qualificatore `const` nella dichiarazione della variabile, che indica che il valore non può essere modificato (cioè rimane costante).

I principali vantaggi nell'usare le costanti sono:

- *Leggibilità del programma:* un identificatore di costante con un nome significativo è molto più leggibile di un numero (`COMPENSO_ORARIO` è autoesplicativo, `20000` non lo è);
- *Modificabilità del programma:* per modificare un valore costante usato nel programma basta semplicemente cambiare la definizione

della costante (es. `const int COMPENSO_ORARIO = 35000`), usando i numeri magici dovremmo modificare tutte le occorrenze del valore nel programma (es. sostituendo dappertutto `20000` con `35000`).

Oltre alla definizione di costanti tramite la parola chiave `const`, è possibile definire costanti anche tramite la direttiva `#define`. Questa è una direttiva che viene processata dal pre-compilatore, sostituendo ad ogni occorrenza del nome definito (nell'esempio `COMPENSO_ORARIO`) il valore corrispondente (nell'esempio `20000`).

Esempio 2.13.

```
#define COMPENSO_ORARIO 20000
...
int compenso = COMPENSO_ORARIO * orelavoro;
```

Si osservi che nel caso della direttiva `#define` non viene effettuato nessun controllo sui tipi.

Esempio 2.14.

```
#define COMPENSO_ORARIO 20500.50f
```

Poiché `COMPENSO_ORARIO` non è una costante del programma, essa non ha un tipo associato. Pertanto il compilatore non verifica alcuna corrispondenza tra tipi quando il valore viene sostituito all'etichetta `COMPENSO_ORARIO` (se, ad esempio, si assegnasse il valore `20500.50f` ad una costante intera, il compilatore ci informerebbe della possibile perdita di informazione).

2.3. Espressioni numeriche

Analogamente al linguaggio Python, variabili, costanti e letterali di tipo numerico (`int`, `float`, `double`, `char`) possono essere combinati usando *operatori aritmetici* (ma non esclusivamente) per creare *espressioni numeriche*, ovvero formule che rappresentano valori numerici.

Gli operatori aritmetici più comuni in C sono:

- $+$, $-$ (unari), applicabili a tutti i tipi numerici;
- $+$, $-$, $*$, / (binari), applicabili a tutti i tipi numerici;
- % (resto della divisione, binario), applicabile solo agli interi.

Quando un valore di tipo `char` occorre in un'espressione aritmetica, esso viene semplicemente interpretato come intero.

Esempio 2.15.

```
char a = 'A';
char b = 'B';
printf("%d\n",a); // Stampa 65
printf("%d\n",b); // Stampa 66
printf("%d\n",a+b); // Stampa 131
```

Nell'ultima istruzione, il valore delle variabili `a` e `b`, rispettivamente 65 e 66, viene interpretato come intero. L'espressione `a+b` restituisce pertanto il valore 131.

Le regole di composizione delle espressioni in cui compaiono gli operatori aritmetici sono essenzialmente quelle dell'aritmetica. Ad esempio, `3 + a` è un'espressione (se `a` è una variabile di tipo numerico) ma `3 + a -*` non lo è.

Anche le regole di precedenza e le associatività rispettano quelle dell'aritmetica. Gli operatori a precedenza più alta sono $+$ e $-$ unari, $*$, / e %, mentre quelli a precedenza più bassa sono $+$ e $-$ binari. Gli operatori unari sono associativi a destra, quelli binari a sinistra. Le parentesi tonde `(,)` possono essere utilizzate per modificare la precedenza tra gli operatori.

Un'espressione viene valutata nella maniera usuale a partire dagli elementi e dagli operatori di cui le espressioni sono composte. Variabili e costanti (di tipo numerico), che, insieme ai letterali, rappresentano i casi più semplici di espressione, hanno valore pari al valore ad esse assegnato.

Il valore di un'espressione può essere intero o reale, a seconda degli elementi che la compongono. Elementi di tipo intero o reale possono essere combinati in una stessa espressione. In tal caso, il valore ottenuto è di tipo reale.

Esempio 2.16.

```
123 // vale 123 (intero)
int x = 12;
x // vale 12 (intero)
```

```
float y = 1.2;
x+y // vale 13.2 (reale)
10/3+10%3 // vale 4 (intero)
2*-3+4 // vale  -2 (intero)
int a = 5, b =7;
2*a+b // vale  17 (intero)
2*(a+b) // vale  24 (intero)
```

Si noti l'impatto delle parentesi e della precedenza degli operatori nelle ultime due espressioni.

Come in Python, Il valore di un'espressione può essere assegnato ad una variabile (o costante). L'assegnazione avviene valutando prima l'espressione a destra dell'operatore = e successivamente assegnando il valore ottenuto alla variabile a sinistra.

Esempio 2.17.

```
int x = 3;
int y = x + 4;
```

Se il tipo dell'espressione è diverso da quello della variabile a cui viene assegnata, il valore dell'espressione viene *convertito automaticamente* prima che l'assegnazione abbia effetto. Questo meccanismo, detto *cast*, verrà analizzato in dettaglio più avanti.

2.4. Espressioni con side-effect ed istruzioni

Il termine *espressione* in C indica due diverse nozioni:

- le espressioni che hanno come effetto solamente il calcolo di un valore, come quelle viste in precedenza;
- le espressioni che, oltre a calcolare un valore, effettuano operazioni sulla memoria, ad esempio, come si vedrà a breve, un'assegnazione. Per queste ultime useremo il termine *espressioni con side-effect*: il termine side-effect si usa per indicare una modifica nella memoria del programma. trasformate in istruzioni se terminate da ";".

2.4.1. Espressioni con assegnazione

L'operatore di assegnazione può essere usato anche per costruire espressioni.

Esempio 2.18.
`x = 7` è un'espressione (con side-effect) che può essere utilizzata come un'espressione qualunque:

```
int x = 0;
printf("%d",x=7);
printf("%d",x);
```

Si noti che terminando l'espressione con il carattere `;`, si ottiene l'istruzione di assegnazione `x = 7;`.

Il valore di un'espressione del tipo *`variabile`* `=` *`espressione`* è pari al valore dell'espressione a destra dell'operatore di assegnazione. Pertanto, l'espressione dell'esempio precedente ha valore 7, che è infatti il valore stampato quando la seconda riga viene eseguita.

Quando un'espressione con side-effect viene valutata, vengono anche eseguite le operazioni in memoria corrispondenti. Nella seconda riga dell'esempio precedente, viene assegnato il valore 7 alla variabile `x`, che è anche il valore stampato quando l'ultima riga viene eseguita.

Nelle espressioni del tipo *`variabile`* `=` *`espressione`*, l'espressione a destra può a sua volta essere un'espressione con side-effect, in particolare contenente l'operatore =. Anche in tal caso, assegnazione e valutazione vengono effettuate considerando prima l'espressione a destra.

Esempio 2.19.

```
int x = 0, y = 0;
printf("%d", y = 3 + (x = 7)); // stampa 10
printf("%d", x); // stampa 7
```

Per valutare l'espressione `y = 3 + (x = 7)` della seconda riga viene prima valutata l'espressione a destra `3 + (x = 7)`, per la quale occorre prima valutare `x=7`. Quest'ultima effettua side-effect assegnando il valore 7 ad `x`, quindi restituisce tale valore. Conseguentemente, l'espressione `3 + (x = 7)` ha valore 10, che è quello che viene assegnato, per effetto dell'operatore = ad `y`.

Occorre prestare particolare attenzione all'uso degli operatori di uguaglianza (==) e assegnazione (=). Uno degli errori più comuni in C è infatti quello di usare = al posto di == all'interno delle condizioni, ad esempio nell'istruzione `if`.

Esempio 2.20.

```
int x = 0;
if (x == 0)
        printf("x vale 0"); // stampa: x vale 0
if (x=0)
        printf("x vale 0"); // non stampa nulla
```

Nella seconda riga, l'espressione `x == 0` restituisce un valore non nullo, in quanto `x` ha effettivamente valore 0, quindi la condizione è soddisfatta ed il ramo *then* viene eseguito.

Nella terza riga, invece, l'espressione (con side-effect) `x=0` assegna alla variabile `x` il valore 0, che è anche il valore restituito in quanto espressione. Conseguentemente, la condizione dell'`if` non è soddisfatta, ed il ramo *then* non viene eseguito.

Sebbene il linguaggio C consenta un uso indifferenziato dei due tipi di espressioni, noi useremo le espressioni con side-effect per formare delle istruzioni, evitandone sempre l'uso all'interno di espressioni matematiche e, in particolare, condizioni.

Esempio 2.21.
L'istruzione

```
x = 5 * (y = 7);
```

dovrebbe essere scritta come

```
y = 7;
x = 5 * y;
```

2.4.2. Operatori di assegnazione composta

Il seguente frammento di programma:

```
somma = somma + addendo;
salario = salario * aumento;
```

si può abbreviare in:

```
somma += addendo;
salario *= aumento;
```

In generale l'assegnazione:

`x = x` *`operatore espressione`*

si può abbreviare in:

`x` *`operatore= espressione`*

Per ciascun operatore `+`, `-`, `*`, `/`, `%` esiste il corrispondente `+=`, `-=`, `*=`, `/=`, `%=`.

2.4.3. Operatori di incremento e decremento

Per incrementare o decrementare una variabile intera di 1, alcune tra le più comuni operazioni presenti in un programma, il C mette a disposizione gli operatori `++` e `--`, che possono essere usati come operatori prefissi (ad esempio nell'istruzione `++x;`) o postfissi(ad esempio `--x;`).

Esempio 2.22.

```
int x = 1;
x++; // incrementa x di 1
printf("%d",x); // stampa 2
--x; // decrementa x di 1
printf("%d",x); // stampa 1
```

Dal punto di vista degli effetti sul valore della variabile a cui sono applicati, non c'è nessuna differenza tra l'uso prefisso o postfisso dello stesso operatore: l'esecuzione delle istruzioni `x++;` o `++x;` comporta lo stesso effetto sulla variabile `x`, ovvero ne viene incrementato il valore di 1.

Tuttavia, tali operatori possono essere usati all'interno di espressioni. In tal caso, le due forme, pur comportando lo stesso effetto sulla variabile, danno luogo a valori diversi delle espressioni. In particolare:

- il valore dell'espressione `++x` è pari al valore iniziale della variabile `x` incrementato di 1;
- il valore dell'espressione `x++` è pari al valore iniziale della variabile `x`;

Un comportamento analogo si ha con l'operatore `--`.

Esempio 2.23. Il seguente frammento di codice mostra il comportamento degli operatori `++` e `--` usati nelle forme prefissa e postfissa all'interno di espressioni.

```
int x = 0;
printf("%d",x); // stampa 0
printf("%d",x++); // stampa 0
printf("%d",x); // stampa 1
printf("%d",x--); // stampa 1
printf("%d",x); // stampa 0
printf("%d",++x); // stampa 1
printf("%d",--x); // stampa 0
printf("%d",x); // stampa 0
```

Quando ++ è usato in forma prefissa, viene chiamato operatore di *pre-incremento* (in quanto incrementa la variabile prima di restituirne il valore). Se usato in forma postfissa è detto di *post-incremento*. Analogamente, l'operatore -- usato in forma prefissa è detto di *pre-decremento* e di *post-decremento* se usato in forma postfissa.

Si noti che le espressioni contenenti ++ e -- sono espressioni con side-effect che è preferibile non utilizzare, per evitare di introdurre errori difficili da rilevare e rendere più chiari i programmi.

2.5. Lettura e scrittura di espressioni numeriche

Lettura (da standard input) e scrittura (su standard output) di espressioni numeriche si effettuano, rispettivamente, mediante le funzioni `scanf` e `printf`, definite nel file di sistema `stdio.h`.

Le funzioni `scanf()` e `printf()` accettano come primo parametro una stringa di caratteri e come parametri successivi le espressioni di cui si vuole stampare il valore. Non c'è limite al numero di espressioni che si possono passare alle funzioni.

2.5.1. La funzione `scanf()`

La funzione `scanf` può essere invocata usando la sintassi seguente:

```
scanf(stringa,par1,par2,...)
```

Il primo parametro, detto *specifica di formato*, è una stringa di caratteri che rappresenta il tipo di dato letto (intero, reale, ecc.)[3]. I parametri successivi sono espressioni della forma `&nome-variabile` che identificano

[3] Questa stringa può in realtà essere più complessa, in quanto `scanf` non si limita a leggere semplici valori. Tuttavia questi aspetti avanzati non saranno considerati in questo corso.

le variabili in cui i dati letti devono essere memorizzati (il significato dell'operatore `&` sarà discusso più avanti).

Esempio 2.24. Il seguente frammento di codice memorizza nella variabile `i` il valore immesso da standard input (ovvero tastiera, se non diversamente specificato).

```
int i;
printf("Inserisci un valore intero\n");
scanf("%d",&i);
```

La stringa `%d` indica alla funzione `scanf` che il valore letto deve essere interpretato come un intero. Il parametro `&i` indica che tale valore deve essere memorizzato nella variabile `i`.

La specifica di formato inizia con il carattere `%` ed è seguita da una combinazione di caratteri che dipendono dal tipo di dato da memorizzare. La seguente tabella mostra le specifiche per i tipi più comuni.

Tipo	Specifica di formato
`int`	`d`
`unsigned int`	`u`
`short int`	`hd`
`long int`	`ld`
`float`	`f`
`double`	`lf`
`char`	`c`

È possibile leggere e memorizzare più valori con una sola invocazione di `scanf`.

Esempio 2.25. I seguenti frammenti, in cui si usa la funzione `scanf` per leggere un valore intero ed uno reale, sono equivalenti per quanto riguarda i valori assegnati alle variabili.

```
int i;
float f;
scanf("%d%f", &i, &f);
```

```
int i;
float f;
scanf("%d", &i);
scanf("%f", &f);
```

In questo caso, l'associazione tra parametri e specifiche di formato avviene su base posizionale, ovvero al secondo parametro è associata la prima specifica, al terzo parametro la seconda e così via.

2.5.2. La funzione `printf()`

La funzione `printf` può essere invocata con una sintassi simile a quella della `scanf`:

`printf(`*`stringa`*`,`*`par1`*`,`*`par2`*`,...)`

Ci sono tuttavia due differenze principali rispetto alla `scanf`:

- La stringa di caratteri può contenere anche i caratteri che si desidera stampare.
- I parametri sono le espressioni di cui si vuole stampare il valore (non si deve anteporre il carattere `&`).

Esempio 2.26. Il seguente frammento mostra un esempio in cui i valori appena letti vengono stampati:

```
int i;
float f;
scanf("%d%f",&i,&f);
printf("i=%d\nf=%f\n",i,f);
```

La funzione `printf` stampa la stringa di caratteri passata come primo parametro, sostituendo alla prima specifica di formato (`%d`) il valore del secondo parametro (ovvero dell'espressione `i`), alla seconda specifica di formato (`%f`), il valore del terzo parametro (`f`) e così via.

Esempio 2.27. L'esecuzione dell'esempio precedente con i seguenti parametri di input:

```
5
12.23
```

Produce il seguente output:

```
i=5
f=12.230000
```

Se la specifica di formato ed il valore letto/stampato non sono conformi, il comportamento delle funzioni risulta non corretto (vedi esempio successivo).

Esempio 2.28. Si consideri il seguente frammento di codice:

```
int i;
scanf("%d",&i);
printf("%f",i);
```

Con il seguente input:
23
L'esecuzione del codice produce l'output:
1027676083... (Si noti, comunque, che il compilatore ci informa, tramite un messaggio di *warning*, dell'uso scorretto del formato).

2.6. Lettura e scrittura di `char`

La lettura e la scrittura di valori di tipo `char` possono avvenire in maniera analoga a quanto visto per i tipi numerici, tramite le funzioni `scanf` e `printf`, usando la specifica di formato `c`, come mostrato nel seguente esempio.

Esempio 2.29.

```
char c;
printf("Inserisci un carattere\n");
scanf("%c",&c);
printf("Il carattere inserito e' %c\n", c);
```

2.6.1. Le funzioni `getchar()` e `putchar()`

Per la lettura e la scrittura di un singolo carattere, il C fornisce inoltre le funzioni `getchar()` e `putchar()`, rispettivamente. La funzione `putchar()` stampa su schermo (o su file, se lo standard output è stato redirezionato con l'operatore >) il carattere passato come argomento. La funzione `getchar()` legge un carattere da tastiera (o da file, se lo standard input è stato redirezionato con l'operatore <) e ne restituisce il rispettivo codice come `int`, che può successivamente essere assegnato ad un `char`. Si noti che sebbene il valore restituito sia di tipo `int`, poiché `getchar` restituisce il codice del carattere, esso può essere assegnato ad una variabile di tipo `char` senza che si verifichi perdita di informazione.

Esempio 2.30.

```
printf("inserisci un carattere\n");
char c = getchar(); // legge il carattere inserito
putchar(c); // stampa il carattere letto
```

Si osservi che nell'assegnazione della seconda riga si sta eseguendo una conversione del valore restituito da `getchar`, dal tipo `int` al tipo `char` della variabile `c`.

2.6.2. Sequenze di escape

Oltre ai caratteri stampabili comuni (`'a'`,`'b'`,`'c'`, etc.), il codice ASCII definisce anche un insieme di caratteri speciali, rappresentati da sequenze. Esse identificano sia caratteri utili nella formattazione dell'output, come ritorno a capo o tab, o caratteri che non potrebbero essere stampati se usati nella forma comune, ad esempio l'apice singolo `'`. Nella seguente tabella sono riportate le sequenze di escape più comuni in C.

Nome	**Sequenza**	Significato
Alert	`\a`	emette il suono di alert
Backspace	`\b`	cancella l'ultimo carattere
New line	`\n`	torna a capo
Carriage return	`\r`	torna all'inizio della riga corrente
Tab orizzontale	`\t`	inserisce degli spazi (in genere 8)
Backslash	`\\`	stampa il carattere \
Apice singolo	`\'`	stampa il carattere '
Apice doppio	`\"`	stampa il carattere "

2.7. Esercizio: teorema di Pitagora

Scrivere un programma che riceva in input da tastiera il valore di due cateti di un triangolo rettangolo e restituisca il valore dell'ipotenusa.

Esempio d'uso

```
Inserisci il valore del primo cateto
3
Inserisci il valore del secondo cateto
4
Il valore dell'ipotenusa e' 5.0
```

Soluzione

```
#include<math.h> // Per funzioni sqrt() e pow()
```

```
#include<stdio.h> // Per scanf() e printf()

int main(){
    float c1, c2, ip;
    printf("Inserisci il valore del primo cateto\n");
    scanf("%f",&c1);
    printf("Inserisci il valore del secondo cateto\n");
    scanf("%f",&c2);
    ip = sqrt(pow(c1,2)+pow(c2,2));
    printf("Il valore dell'ipotenusa e'%f\n",ip);
}
```

2.8. Conversione di tipo

Il C permette di specificare espressioni aritmetiche e logiche che coinvolgono tipi primitivi diversi (incluso il tipo `char`).

Per valutare tali espressioni è necessario *convertire* i valori in maniera tale da eseguire le operazioni tra valori dello stesso tipo.

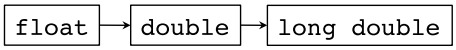
Il C esegue queste conversioni senza l'intervento del programmatore nei seguenti casi:

1. I tipi coinvolti in un'espressione logico-aritmetica sono diversi (ad esempio `int` e `float`);
2. Il tipo dell'espressione nel lato destro di un'assegnazione non coincide con quello del lato sinistro (ad esempio `int x = 3.2;`)
3. Il tipo del parametro attuale passato ad una funzione non coincide con il tipo del parametro formale corrispondente (v. seguito);
4. Il tipo dell'espressione di una `return` non coincide con il tipo di ritorno della funzione (v. seguito).

Normalmente, la conversione avviene in maniera tale da non perdere informazione, ovvero verso i tipi il cui dominio copre l'intervallo più grande.

Ad esempio nell'espressione 1 + 3.0, il valore 1 viene convertito da `int` a `float` prima di valutare l'espressione.

Per il caso 1, si consideri un'espressione del tipo *`v1 op v2`*, dove *`op`* denota un operatore aritmetico. In questo caso, se uno dei due operandi è di tipo floating point, tutti gli operandi vengono convertiti nel tipo tipo floating point a precisione maggiore.



Esempio 2.31.

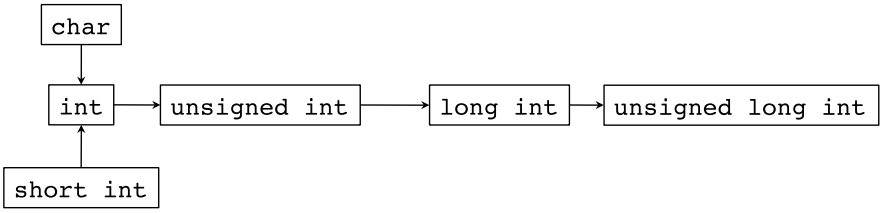
```
double d = 3.23199;
float f = 56.3419;
int i = 100;

printf("%lf\n", (d + i));
printf("%f\n", (f + d));
printf("%lf\n", (f + d + i));
```

L'espressione `(d+i)` nella quarta riga viene valutata convertendo prima il risultato dell'espressione intera `i` in `double` e successivamente sommando il valore ottenuto (100.000000) a quello dell'espressione `double d`, ottenendo come risultato il valore 103.231990.

Nell'espressione `(f+d)` è invece `f` ad essere convertita in `double` prima di eseguire la somma.

Nell'espressione complessa `(f + d + i)`, viene convertito il valore di `i` in `double`, quindi viene valutata l'espressione `(d + i)` (di tipo `double`), successivamente viene convertito il valore di `f` in `double` e infine viene calcolato il valore dell'espressione completa `(f+(d+i))`.

Se i tipi coinvolti sono solo interi, eventualmente con rappresentazioni diverse, ad esempio `char` e `unsigned`, gli `short` ed i `char`, se presenti, vengono convertiti in `int` e successivamente viene applicata la conversione dell'operando con rappresentazione più piccola nel tipo di quello con rappresentazione più grande, secondo lo schema seguente:



Esempio 2.32.

```
long int i = 100;
```

```
short int j = 8;
printf("%ld\n", (j-i));
```

L'espressione `(i-j)` viene valutata convertendo il valore (`short`) di `i` in `int`, poi da `int` in `long int`, e successivamente eseguendo la differenza.

Nel caso in cui *op* sia un operatore logico (ad esempio `&&`), gli operandi vengono convertiti in `int`. Si noti che questa operazione può comportare perdita di precisione.

La seguente tabella descrive il tipo risultante di una espressione della forma `a + b` per ciascuna coppia di tipi possibili (signed) per `a` e `b`:

a + b	char	short	int	long	float	double
char	int	int	int	long	float	double
short	int	short	int	long	float	double
int	int	int	int	long	float	double
long	long	long	long	long	float	double
float	float	float	float	float	float	double
double	double	double	double	double	double	double

Per il caso 2, il valore dell'espressione nel lato destro dell'assegnazione viene convertito nel tipo della variabile a sinistra, e successivamente viene effettuata l'assegnazione.

Si noti che quando non si ha perdita di informazione, la conversione non può generare errori derivanti dal valore dell'espressione a destra dell'operatore di assegnazione.

Esempio 2.33.

```
int i = 10;
long int j = i;
float f = 100.23f;
double d = f;
```

Se si effettua una conversione da un tipo in virgola mobile ad uno intero, si ha il troncamento della parte razionale.

Esempio 2.34.

```
double d = 102.345;
long int i = d; // i vale 102
```

Negli altri casi (ad esempio da `int` a `short`), affinché la conversione non dia luogo a perdita d'informazione, occorre assicurarsi che il valore da convertire sia incluso nell'insieme dei valori rappresentabili nel tipo di destinazione.

Esempio 2.35.

```
int i = 32767;
short j = i; //OK: 32767 rappresentabile come short
printf("%d\n",j); // 32767
j = i+1; // NO: 32768 fuori intervallo short
printf("%d\n",j); // -32768 !!!
```

Per i casi 3 e 4, valgono essenzialmente le stesse regole del caso 2, ma l'espressione convertita viene assegnata al parametro della funzione o restituita dalla funzione.

2.9. Casting

Il C permette di convertire dati da un tipo ad un altro in maniera esplicita. Ad esempio, è possible convertire il `float` 3.2 in `int`, ottenendo 3. Questa operazione è detta *cast* ed ha la seguente sintassi.

Cast

Sintassi:

(*tipo*) *espressione*;

- *tipo* è il nome di un tipo di dato;
- *espressione* è l'espressione di cui si vuole convertire il tipo in *tipo*

Semantica:
Converte il tipo di *espressione* in *tipo*.

L'operatore di cast ha precedenza più alta rispetto agli operatori aritmetici.

Esempio 2.36. Nel seguente esempio, a seconda dell'uso dell'operatore di cast, / può identificare la divisione tra reali o tra interi.

```
float x = 7.0f;
float y = 2.8f;
int a = x / y; // Divisione reale, risultato = 2.5
printf("%d\n",a); // Stampa 2
```

```
a = (int) x / (int) y; // Divisione intera, risultato = 3
printf("%d\n",a); // Stampa 3
```

Nella terza riga, l'espressione `x /y` restituisce il valore reale $2.66\cdots$, che viene poi troncato a 2 per completare l'assegnazione ad `a`. Nella quinta riga, invece, il casting viene effettuato *prima* che sia applicata la divisione, convertendo i valori delle espressioni `x` ed `y` rispettivamente in 7 e 2. Poichè gli operandi sono interi, la divisione viene interpretata come intera, e viene restituito pertanto il valore 3.

Si noti che, in questo esempio, la conversione comporta *perdita di informazione*, dovuta al troncamento.

ATTENZIONE: le variabili `x` e `y` rimangono di tipo `float`! È solo il valore restituito dalle espressioni ad essere convertito.

In generale è possible eseguire il casting da e verso qualunque tipo. Le regole del casting sono le stesse discusse per le conversioni in presenza di assegnazione.

Esempio 2.37.

```
double d; float f; long l; int i; short s;

/* Le seguenti assegnazioni sono corrette
 (e non comportano perdita di informazione)*/

d=(double)f; l=(long)i; i=(int)s;

/* Le seguenti assegnazioni sono corrette
 (ma possono comportare perdita di informazione) */

f=(float)d; l=(long)f; i=(int)l; s=(short)i; f=(float)l;
```

Nonostante vi siano molte occasioni in cui la conversione di tipo avviene in maniera implicita e non vi sia necessità di eseguire il casting, è sempre opportuno farlo per migliorare la leggibilità del codice.

Esempio 2.38.

```
float f = 23.45f;
int a = f; // NO!
int a = (int) f; // SI
```

2.10. L'operatore `sizeof`

L'operatore `sizeof`, applicato ad un tipo, restituisce la quantità di memoria, in byte, necessaria a memorizzare un valore del tipo specificato come parametro.

Esempio 2.39.

```
int i = sizeof(char); // i vale 1 (Byte)
i = sizeof(int); // i vale 4 (in architettura a 32 bit)
```

Il valore restituito da `sizeof` è di tipo intero senza segno.
`sizeof` può anche essere applicato ad espressioni (incluse costanti e variabili). In tal caso restituisce la dimensione associata al tipo del valore dell'espressione.

Esempio 2.40.

```
int i = 10;
int j = sizeof(i); // j vale 4 (in architettura a 32 bit)
j = sizeof(i+2); // j vale 4 (in architettura a 32 bit)
j = sizeof(i+2.0); // j vale 8 (i+2.0 e' un double!)
```

2.11. Definizione di nuovi tipi

In C è possibile definire nuovi tipi mediante l'istruzione `typedef`.

Esempio 2.41.

```
typedef int TipoA;
typedef int TipoB;
typedef int bool;

TipoA a = ...;
TipoB b = ...;
```

I tipi definiti sono sempre compatibili con il tipo base, quindi nell'esempio precedente l'istruzione `int x=a+b;` è valida e ritorna un risultato di tipo `int`.

Il vantaggio principale nell'uso di `typedef` consiste in una migliore manutenibiltà del programma.

Esempio 2.42.

```
typedef int valuta;
valuta stipendio;
/* codice in cui stipendio viene letto, manipolato, etc. */
```

Se la valuta cambia da lira ad euro (quindi da intero a reale), è sufficiente rimpiazzare la prima riga con la seguente definizione di tipo:

```
typedef float valuta;
```