4

DATI ED OPERAZIONI

Se hai letto i capitoli precedenti, ed hai superato la fase dello svezzamento, è tempo di sollevare un altro velo: capire cosa corrisponda effettivamente ai diversi tipi di dati e che operazioni tu ci possa eseguire. È quello che mi propongo di fare in questo capitolo. Il contenuto comincia a diventare un po' più tecnico rispetto ai capitoli precedenti.

4.1 I tipi di dati

Possiamo distinguere 4 tipi fondamentali:
- *interi*, rappresentati in base 2 e facendo uso dell'aritmetica modulare;
- *in virgola mobile*, rappresentazione come coppia mantissa–esponente più un bit di segno;
- *caratteri*, rappresentati mediante il codice ASCII;
- *campi generici di bit* il cui significato viene assegnato dal programmatore.

A queste quatto categorie dobbiamo aggiungere anche
- *puntatori* o *indirizzi*, che controllano la posizione o il movimento di dati in memoria.

Sui manuali si distingue solitamente anche tra *variabili* e *costanti*.

Dopo un inizio così brillante immagino che la tua testa, caro lettore non ancora abituato alle acrobazie del computerese, sia come la campagna padana in una sera d'inverno: piena di nebbia. Vediamo di orientarci. In questo capitolo non discuterò di puntatori o indirizzi: ne parleremo diffusamente più avanti. Concentrerò invece l'attenzione sui tipi di dati e sulle operazioni che possiamo compiere.

La memoria, come ti ho già spiegato, è fatta di celle contenenti dei bit, identificate da un indirizzo. La domanda buona è: cosa posso rappresentare con i bit della memoria? Risposta: tutto quello che potresti rappresentare in modo simbolico: basta associare ad ogni simbolo un numero.

E allora, perché si parla di dati in qualche modo predefiniti? Semplice: perché è utile svolgere delle operazioni sui dati, e alcune operazioni sono standard. Quando si dice che la CPU riconosce certi tipi di dati si intende semplicemente che può svolgere delle operazioni elementari su questi tipi di dati. In termini precisi: tra i codici di istruzione che la CPU è in grado di interpretare ed eseguire ce ne sono alcuni che operano su certi tipi predefiniti di dati. Ad esempio, se ho due numeri interi la CPU è in grado di addizionarli, sottrarre il primo dal secondo, moltiplicarli o dividere il primo per il secondo. Se ho due scritte che rappresentano due numeri interi (tra poco ti spiegherò come) la CPU è in grado di spostarle in memoria – un carattere per volta – ma di farci delle operazioni aritmetiche proprio non se ne parla: ci vuole un programma che converta i numeri in binario. Analogamente, la CPU è in grado di eseguire operazioni sui numeri reali (affermazione da prendersi con le dovute cautele!... continua a leggere). Ma di fare i conti con le frazioni non se ne parla proprio, a meno di scrivere un programma.

Non è mia intenzione farti un elenco delle istruzioni della CPU o spiegarti il linguaggio Assembler. Mi basta comunicarti quanto serve per metterti in grado di intravedere cosa si nasconda dietro la facciata delle dichiarazioni di tipo e delle istruzioni, e soprattutto di evitare i trabocchetti nascosti tra le pieghe del linguaggio. Per le informazioni tecniche dettagliate ci sono i manuali.

4.1.1 I dati numerici di tipo intero

Hai già fatto la conoscenza con i tipi `int` e `double`. Ora è il momento di elencare in modo ragionevolmente completo i vari tipi utilizzabili.

Iniziamo con i dati numerici di tipo intero. Le differenti dichiarazioni corrispondono a diverse lunghezze del campo di bit destinato a memorizzare il dato. Questo si riflette sull'intervallo di numeri rappresentabili. I numeri qui sotto ti sembreranno probabilmente misteriosi, ma non c'è nessuna cabala. Se vuoi capire il perché dei limiti leggi il paragrafo 4.3.

I tipi che seguono riguardano i dati interi con segno.

- `char` : occupa 1 byte (8 bit), e può contenere interi tra -128 e 127 .
- `short int` , abbreviabile in `short` : occupa 2 bytes (16 bit), e può contenere interi tra $-32\,768$ e $32\,767$.
- `long int` , abbreviabile in `long` : occupa 4 bytes (32 bit), e può contenere interi tra $-2\,147\,483\,648$ e $2\,147\,483\,647$.
- `long long int` : occupa 8 bytes (64 bit), e può contenere interi tra $-9\,223\,372\,036\,854\,775\,808$ e $9\,223\,372\,036\,854\,775\,807$.
- `int` è considerato in qualche modo il tipo standard di intero, ma quale sia lo standard dipende dal compilatore e dalla macchina. Il compilatore `gcc` (gnu C compiler) per macchine del tipo PC–IBM identifica il tipo `int` col tipo `long int` , ma è bene non fare affidamento su questa identificazione: altri compilatori identificano `int` con `short int` .

Accanto ai tipi interi con segno si possono utilizzare anche tipi interi senza segno, ossia numeri non negativi. Per completezza, ecco l'elenco.

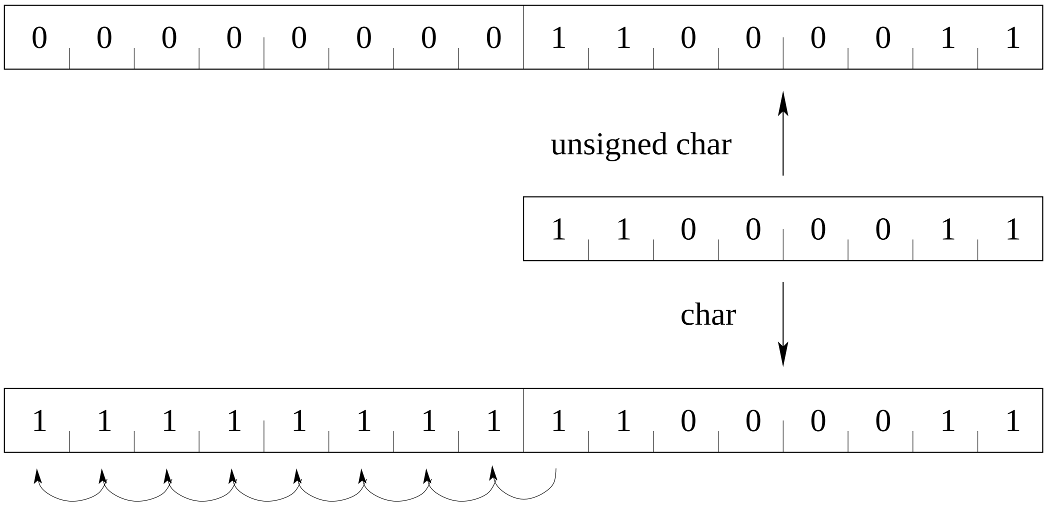

Figura 4.1. Il trasferimento di un dato intero da una cella di un byte ad una di due bytes: i bit più significativi vengono riempiti col criterio di estensione del segno.

- `unsigned char`: occupa 1 byte (8 bit), e può contenere interi tra 0 e 255 .
- `unsigned short int`, abbreviabile in `unsigned short`: occupa 2 bytes (16 bit), e può contenere interi tra 0 e 65 535 .
- `unsigned long int`, abbreviabile in `unsigned long`: occupa 4 bytes (32 bit), e può contenere interi tra 0 e 4 294 967 295 .
- `unsigned long long int`: occupa 8 bytes (64 bit), e può contenere interi tra 0 e 18 446 744 073 709 551 615 .
- `unsigned int`: valgono le stesse considerazioni svolte per il tipo `int`, mutatis mutandis.

Parlare semplicemente di dati interi è ingenuo e fuorviante: dati di lunghezza diversa devono essere considerati a tutti gli effetti come diversi, pena il rischio di incorrere in errori subdoli e difficilmente identificabili. Lo stesso vale per dati con o senza segno. I rischi si presentano quando si fanno operazioni aritmetiche o si trasferiscono dati tra variabili o espressioni di diverso tipo. Qualche esempio dovrebbe chiarire il problema.

Il primo caso è l'assegnazione di una variabile ad una seconda di lunghezza o di tipo superiore, ad esempio `char` o `unsigned char`, di lunghezza 1 byte, ad uno dei tipi `short int` o `long int` o long long int. Gli 8 bit del dato contenuto nel byte non sono evidentemente sufficienti a riempire tutto il campo di destinazione. Il risultato è illustrato in figura 4.1: il criterio è quello dell'estensione del segno.

Guardiamo dietro le quinte. Supponi di aver scritto l'istruzione `j=k`, dove `j` è di tipo `short int` e `k` è di tipo `char`. Per trasferire il dato la CPU deve fare una conversione, aggiungendo dei bit in testa al dato `char`. Che criterio usa? Cerco di spiegartelo.

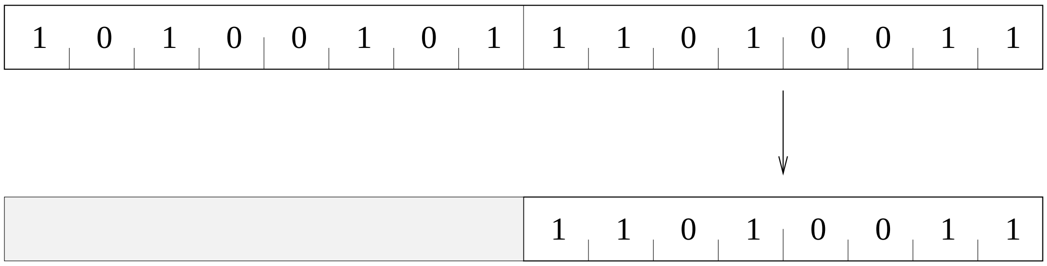

Figura 4.2. Il trasferimento di un dato intero da una cella ad una più corta: la parte in eccesso viene semplicemente troncata.

Il bit più significativo della cella viene detto *bit di segno*; per il momento questo fatto ti sembrerà alquanto misterioso, ma prendilo per buono: lo capirai quando avrai letto il paragrafo 4.3. Il significato di questo bit cambia a seconda che il dato `k`, nell'esempio che stiamo facendo, sia stato dichiarato `unsigned char` o semplicemente `char` — cioè se abbia segno o no. A dire il vero, la CPU non riconosce un intero come con o senza segno semplicemente leggendolo dalla memoria: i bit sono tutti uguali. Ciò che differenzia i tipi di dati sono le istruzioni predisposte dal compilatore: l'istruzione della CPU che converte un dato senza segno è diversa da quella che converte un dato con segno. In effetti, se il dato è di tipo `unsigned` la conversione viene effettuata azzerando i bit aggiunti; se invece il dato è con segno viene effettuata un'*estensione del segno.*[1] Precisamente, il valore del bit più significativo viene copiato nei bit da aggiungere. Lo scopo di questa regola è quello di garantire che il numero rappresentato si mantenga durante il trasferimento.

Più delicato è il trasferimento opposto: da una cella ad una più corta. Quello che accade è illustrato in figura 4.2. Non essendovi spazio sufficiente per trasferire tutto il dato, si ricorre al metodo brutale della decapitazione. Questo può rivelarsi un meccanismo molto utile, se vuoi davvero buttar via la testa — del dato, naturalmente. Ma può avere conseguenze del tutto imprevedibili se questa operazione ti fa perdere delle informazioni. Questi sono i fatti; non c'è molto da aggiungere.

Ma che succederebbe se trasferissi un `unsigned int` in un `int`, o viceversa, o se...? Le domande di questo tipo potrebbero essere molte. La risposta generale te l'ho già data, ed il massimo che posso fare è aggiungere che se le celle hanno la stessa lunghezza il contenuto viene copiato tale e quale (il che era da aspettarsi). Se ti resta qualche dubbio, hai una sola cosa da fare: prova!

Le operazioni aritmetiche che si possono effettuare sugli interi sono quelle consuete di addizione (`+`), sottrazione (`-`), moltiplicazione (`*`) e divisione (`/`). Attenzione però: la divisione comporta un troncamento del risul-

[1] Ripeto: è una locuzione un po' misteriosa se non sai come vengono rappresentati i dati negativi. Accontentati per il momento di sapere cosa succede ai bit. Ti si chiarirà tutto quando ti avrò spiegato la rappresentazione degli interi.

tato alla parte intera: `1/3` dà come risultato zero! Ed il resto? Il linguaggio C permette anche l'operazione di calcolo del resto, indicata col simbolo `%`. Ad esempio, `8%5` dà come risultato 3.

I meccanismi di conversione che funzionano per le assegnazioni si applicano anche alle espressioni aritmetiche. Comincia a considerare un'operazione tra due numeri. La regola è un pochino contorta: le operazioni vengono normalmente eseguite nel tipo `int`, qualunque esso sia sulla macchina e col compilatore che stai usando; se un'operazione coinvolge un dato di lunghezza superiore a `int`, il tipo di lunghezza inferiore viene convertito in quello più alto. Dunque, fin che si lavora con numeri che non superano i limiti del tipo `int` non dovrebbero verificarsi inconvenienti: comunque i dati vengono convertiti in `int` prima di eseguire l'operazione. I guai saltano fuori nel momento in cui si superano i limiti, e qui c'è il tranello. Supponiamo che tu abbia scritto un programmino di questo genere: [2]

```
#include <stdio.h>
int main()
  long long int a;
  int b,c;
  b = c = 2000000000; a=b*c;
  printf("%d x %d = %Ld\n",b,c,a);
  exit(0);
```

Cosa ti aspetti come risultato? Un rapido calcolo mentale, e mi risponderai: un 4 seguito da 18 zeri! Ebben, proviamo. Sbuffando un po', perché ti sembra di faticare per nulla visto che conosci il risultato, batti il sorgente, compili, esegui e... resti a bocca aperta davanti alla risposta

```
2000000000 x 2000000000 = -1651507200
```

Ma allora è proprio stupido! No, insisto: è un calcolatore,... con tutto quel che segue. Che è successo? Semplice. Il risultato eccede i limiti della cella `int`; infatti tu, saggiamente, gli hai detto di memorizzare il risultato in un dato `long long int`. Ma i due dati da moltiplicare sono di tipo `int`, quindi la CPU ha eseguito l'operazione usando il tipo `int` ed ha buttato a mare tutti i bit che sono fuorusciti dalla cella `int`. Poi ha convertito quello che gli restava in `long long int` e l'ha trasferito in `a`. Il numero che ti salta fuori non è casuale: leggi il paragrafo 4.3, e scoprirai che avresti potuto prevederlo esattamente.

Non c'è modo di spiegargli che i conti li deve fare giusti? Certamente: basta costringerlo ad effettuare la conversione *prima* di moltiplicare. Modifica

---

[2] L'istruzione `b=c=2000000000` non è un errore: il trasferimento avviene da destra a sinistra. Significa: metti in `b` il risultato dell'operazione che c'è a destra del segno `=`; l'operazione incriminata è l'assegnazione di un valore a `c`, quindi il risultato è il valore di `c`. Un modo rapido per assegnare lo stesso valore a più variabili. Il campo `%Ld` nel formato di conversione dell'istruzione `printf(...)` sta ad indicare che il dato è di tipo `long long int`, e non `int`.

l'istruzione `a=b*c` in

```
a = ((long long int) b)*((long long int) c);
```

ricompila, riesegui, e vedrai che la risposta diventerà

```
2000000000 x 2000000000 = 4000000000000000000
```

Conta pure gli zeri: sono proprio 18, e non perché io li abbia battuti con attenzione: ho solo trasferito qui la riga scritta dal programma sul terminale, senza modificarla in nessun modo. La scrittura `((long long int) b)` significa: prima di fare qualunque altra cosa converti il dato `b` in `long long int`; tanto è bastato per costringere il compilatore ad eseguire il calcolo usando le istruzioni per il tipo di lunghezza superiore.

Se devo calcolare espressioni più complesse basta ricordare che la CPU non le calcola in blocco: è il compilatore che predispone le operazioni in modo che vengano trasformate in una successione di operazioni aritmetiche tra due operandi. Le regole di conversione si applicano a ciascuna operazione. L'ordine è quello consueto dell'algebra: le moltiplicazioni e divisioni hanno priorità superiore ad addizioni e sottrazioni; le parentesi modificano l'ordine; due operazioni consecutive della stessa priorità vengono eseguite nell'ordine in cui sono scritte.

Conviene aggiungere qualche parola anche sulle costanti. Scrivere un numero intero in un programma è del tutto lecito, ovviamente; del resto l'abbiamo già fatto. Naturalmente, una costante occupa una cella di memoria, esattamente come una variabile. Ma allora che differenza c'é? Proprio il fatto che è costante, cioè non dovrebbe essere mai modificata. Il compilatore C di solito assegna una cella di memoria – e come potrebbe non farlo? – ma non mi permette di utilizzarne l'indirizzo. Così, scrivere `127 = j` non è lecito, perché il compilatore si rifiuta di trasferire il valore di `j` nella cella che contiene 127 (anche se la CPU non avrebbe nessuna remora a farlo, se lo programmassi in Assembler); e neppure è lecito scrivere `&(127)` intendendo che voglio sapere a che indirizzo si trova quel dato: il compilatore si rifiuta di dirmelo, proprio per evitare che io ne modifichi il valore.[3]

C'è una domanda più sottile: quando scrivo una costante intera, che tipo (ovvero, che lunghezza per la cella di memoria) usa il compilatore? La risposta ufficiale è: una cella di tipo `int`, se basta; altrimenti passa al tipo

---

[3] La discussione sembra un po' bizantina, ma il problema è reale. Io potrei chiamare la funzione `scanf` scrivendo `scanf("%d",&(127));` e la funzione trasferirebbe il dato che ha letto nella cella di memoria che contiene 127. Se l'istruzione successiva fosse `j=127` il compilatore potrebbe tradurla come: prendi il contenuto della cella dove ho messo 127 e trasferiscilo nella cella associata a `j`. Riesci ad immaginare che accadrebbe? Il guaio è che alcuni linguaggi permettono una cosa del genere (anche se il comportamento in esecuzione è tutt'altro che uniforme tra diverse macchine e compilatori); il guaio ancor più grosso è che qualcuno lo fa, inavvertitamente, e poi corre dalla maga per chiederle di scovare l'errore con la sfera di cristallo.

di lunghezza superiore. [4] Volendo, sono libero di forzare l'uso di una cella più lunga scrivendo il numero nella forma, ad esempio, `2147483647L`. La `L` finale sta ad indicare che voglio il tipo di lunghezza superiore ad `int`.

A questo punto dovresti essere in grado di giocare con gli interi come ti pare: ti manca solo di sapere come vengono memorizzati, ma per questo ti rimando al paragrafo 4.3.

4.1.2 I dati in virgola mobile

I dati reali utilizzabili come standard nel linguaggio C sono fondamentalmente di due tipi, che si differenziano anch'essi per la lunghezza della cella di memoria utilizzata. Naturalmente la differente lunghezza della cella si riflette sia sulla precisione del dato, sia sull'intervallo di numeri rappresentabili. [5]

Il superamento del limite superiore della rappresentazione viene detto *overflow*; che possa accadere è evidente a chiunque. Meno immediato è rendersi conto che esiste anche un numero reale minimo rappresentabile; il tentativo di calcolare un numero più piccolo viene detto *underflow*. Gli intervalli positivo e negativo di rappresentazione sono simmetrici, quindi basta specificare quello positivo. Gli intervalli specificati qui sotto sono approssimati, ma ragionevolmente vicini alla realtà.

- `float`: occupa 4 bytes (32 bit), e può rappresentare numeri compresi tra 1.41×10^{-45} e 3.40×10^{38} con una precisione di 6–7 cifre significative.
- `double`: occupa 8 bytes (64 bit), e può rappresentare numeri compresi tra 4.94×10^{-324} e 1.79×10^{308} con una precisione di 15–16 cifre significative.

Alcuni compilatori ammettono anche la dichiarazione `long double`, che dovrebbe corrispondere alla cosiddetta *quadrupla precisione*, superiore a quella del tipo `double`. Tuttavia il comportamento non è uniforme. Idealmente dovrebbe corrispondere ad una lunghezza della cella di 16 bytes (128 bit), e consentire di rappresentare numeri compresi tra 3.65×10^{-4951} e 1.18×10^{4932} con una precisione massima di 32–33 cifre decimali. Tuttavia questa precisione è effettivamente raggiungibile solo se il floating point processor della macchina lo consente, o se esiste un software di simulazione che esegue il calcolo. La sola raccomandazione è: prima di buttarti a calcolare in quadrupla precisione, controlla se sulla tua macchina e col tuo compilatore esiste realmente. Le versioni più recenti del compilatore Gnu–CC funzionanti sui PC tipo IBM accettano la dichiarazione `long double`, ma in realtà assegnano 12 bytes (96 bit) alla cella di memoria, e ne utilizzano effettivamente

---

[4] Attenzione però: non bisogna mai sottovalutare lo spirito di iniziativa di chi l'ha scritto, il compilatore.

[5] In queste note faccio esplicito riferimento a calcolatori con un memoria organizzata in bytes di 8 bit. Ovviamente, una diversa organizzazione della memoria comporta anche differenze sui reali rappresentabili. La sola buona regola è verificare ogni volta che si accede ad una nuova macchina.

solo 10. L'intervallo dei numeri rappresentabili resta quello indicato, ma la precisione si riduce a 19 cifre decimali – sensibilmente ridotta rispetto a quella garantita da una vera quadrupla precisione. In effetti il compilatore non fa altro che sfruttare il fatto che il floating point processor esegue il calcolo in registri di 80 bit; per l'appunto 10 bytes.

Come per gli interi, occorre prestare attenzione all'uso di dati di tipi diversi. Non è proibito, naturalmente, ma occorre prestare attenzione alle conversioni: se si trasforma un dato `double` in `float` si perde in precisione nella rappresentazione, perché le cifre che eccedono le 6–7 rappresentabili in `float` vengono semplicemente troncate. Inoltre occorre tener conto che l'intervallo rappresentabile in `float` è sensibilmente ridotto rispetto a quello del `double`. Si può quindi incorrere in un overflow, oppure un underflow. Considerazioni simili valgono anche per il tipo `long double`.

Più delicata è la questione della conversione tra dati interi e dati in virgola mobile. Il comportamento è quello che ti puoi immaginare: la conversione da intero a virgola mobile può provocare la perdita di qualche cifra significativa in caso di conversione da intero su 32 bit a `float` o di di intero su 64 bit a `double` (il massimo intero rappresentabile eccede il numero di cifre significative). Fatti salvi questi casi, la conversione può considerarsi praticamente esatta. Nel caso di conversione da virgola mobile ad intero si ha certamente il troncamento della parte decimale del numero, oltre al rischio di incorrere in un overflow.

Tra dati in virgola mobile si possono eseguire le quattro operazioni aritmetiche consuete; non è ammesso invece l'uso dell'operazione resto della divisione (`%`), che è consentito solo tra interi. Se questa ti sembra una grave lacuna, non preoccuparti: esiste la funzione di libreria `fmod` che esegue lo stesso calcolo. Precisamente, l'istruzione `z = fmod(x,y)` pone in `z` il resto della divisione di `x` per `y`. Le regole di conversione sono le stesse enunciate per gli interi, con la sola aggiunta dell'informazione che `float`, `double` e `long double`, nell'ordine, sono considerate di classe superiore rispetto a qualunque tipo `int`. In altre parole, se un'operazione aritmetica coinvolge un intero ed un dato in virgola mobile il dato intero viene promosso a virgola mobile, e poi si esegue l'operazione.

Una questione che merita particolare attenzione è l'uso degli interi nelle espressioni aritmetiche che coinvolgono la divisione. C'è il tranello, naturalmente. L'esempio tradizionale è: `x=1/3` dà come risultato 0, e non $0.3333333\ldots$. L'espressione generale pericolosa è del tipo `x=j/k`, con `j` e `k` interi: l'operazione viene eseguita su interi, e quindi con troncamento. Per superare il problema basta forzare la conversione scrivendo, ad esempio, `x=((double) j)/((double) k)`.

Infine, una nota sulla scrittura delle costanti. La prima osservazione apparentemente curiosa, ma estremamente utile, è che per forzare la memorizzazione di un numero intero come dato in virgola mobile basta aggiungere il punto decimale. In altre parole, le scritture `3` e `3.` non sono equivalenti:

nel primo caso il compilatore memorizza il dato come `int`, nel secondo come `double` (il tipo `float` in C è ammesso, ma non usato come standard). Tornando all'esempio fatto poco sopra, basta scrivere `x=1./3.` per trovare il risultato corretto (nei limiti della precisione di macchina, ovviamente). La seconda informazione riguarda l'uso della notazione esponenziale: un esempio vale più che una lunga disquisizione: la scrittura `6.022169e+23` significa 6.022169×10^{23}. Il segno `+` dell'esponente può essere omesso. Tutte le altre considerazioni svolte parlando delle costanti intere si applicano ancora.

4.1.3 Caratteri

Rappresentare i caratteri su un computer può sembrare, a prima vista, una faccenda complicata. Ma è più semplice di quanto non appaia a prima vista... salvo complicazioni. Il fatto è che i caratteri altro non sono che un insieme di simboli. Basta associare a ciascun simbolo un numero, e il gioco è fatto. Quello che è difficile è mettere d'accordo tutti i costruttori sul codice di rappresentazione, soprattutto quando i simboli diventano tanti.

Il codice attualmente usato come standard è detto *codice ASCII.*[6] Lo trovi nella tabella 4.1. I caratteri rappresentabili sono tutte le lettere maiuscole e minuscole, le cifre ed un certo numero di caratteri speciali (di interpunzione, parentesi, &c).

Se scorri la tabella noterai che all'inizio ci sono 32 caratteri dai nomi strani, e lo stesso vale per l'ultimo carattere; poi comincia a comparire un certo ordine. È comodo fare riferimento al codice ottale (i numeri si ricordano meglio, almeno credo). Il codice 40_8 è il carattere spazio (eh sí, anche se diresti che sulla carta non c'è nulla tra due parole ci devi ben mettere qualcosa che le separi; ad esempio, quando scrivi devi battere la barra spaziatrice); a partire da 60_8 trovi le cifre; a partire da 101_8 trovi l'alfabeto maiuscolo; a partire da 141_8 trovi l'alfabeto minuscolo. Tutto quanto è lasciato libero da spazio, cifre, caratteri alfabetici e caratteri dall'apparenza strana è riempito da caratteri speciali — segni di interpunzione &c. Richiamo la tua attenzione sul fatto che la differenza tra il codice di una lettera minuscola e quello della corrispondente maiuscola è 40_8: un fatto comodo.

Cosa sono i caratteri dai nomi strani? Alcuni sono stati introdotti per il controllo della trasmissione di dati, tipicamente, almeno all'inizio, su linee telefoniche. Altri invece sono i caratteri che, per così dire, non si vedono, ma lasciano un segno, un po' come lo spazio. Eccoli:

- `BEL` , codice ottale 007 : è quello che se inviato al terminale fa suonare il cicalino;
- `BS` , codice ottale 010 : *back–space*, sposta indietro il cursore di un carattere; a volte viene usato per cancellare il carattere precedente;

---

[6] Il nome ASCII è un acronimo per *American Standard Code for Information Interchange*. Lo scopo è evidente dal nome stesso.

Tavola 4.1. Il codice ASCII per la rappresentazione dei caratteri. Sono riportati i codici ottale, decimale ed esadecimale.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL	053	43	2B	+	126	86	56	V
001	1	01	SOH	054	44	2C	,	127	87	57	W
002	2	02	STX	055	45	2D	-	130	88	58	X
003	3	03	ETX	056	46	2E	.	131	89	59	Y
004	4	04	EOT	057	47	2F	/	132	90	5A	Z
005	5	05	ENQ	060	48	30	0	133	91	5B	[
006	6	06	ACK	061	49	31	1	134	92	5C	\
007	7	07	BEL	062	50	32	2	135	93	5D	]
010	8	08	BS	063	51	33	3	136	94	5E	^
011	9	09	HT	064	52	34	4	137	95	5F	_
012	10	0A	LF	065	53	35	5	140	96	60	`
013	11	0B	VT	066	54	36	6	141	97	61	a
014	12	0C	FF	067	55	37	7	142	98	62	b
015	13	0D	CR	070	56	38	8	143	99	63	c
016	14	0E	SO	071	57	39	9	144	100	64	d
017	15	0F	SI	072	58	3A	:	145	101	65	e
020	16	10	DLE	073	59	3B	;	146	102	66	f
021	17	11	DC1	074	60	3C	<	147	103	67	g
022	18	12	DC2	075	61	3D	=	150	104	68	h
023	19	13	DC3	076	62	3E	>	151	105	69	i
024	20	14	DC4	077	63	3F	?	152	106	6A	j
025	21	15	NAK	100	64	40	@	153	107	6B	k
026	22	16	SYN	101	65	41	A	154	108	6C	l
027	23	17	ETB	102	66	42	B	155	109	6D	m
030	24	18	CAN	103	67	43	C	156	110	6E	n
031	25	19	EM	104	68	44	D	157	111	6F	o
032	26	1A	SUB	105	69	45	E	160	112	70	p
033	27	1B	ESC	106	70	46	F	161	113	71	q
034	28	1C	FS	107	71	47	G	162	114	72	r
035	29	1D	GS	110	72	48	H	163	115	73	s
036	30	1E	RS	111	73	49	I	164	116	74	t
037	31	1F	US	112	74	4A	J	165	117	75	u
040	32	20	Space	113	75	4B	K	166	118	76	v
041	33	21	!	114	76	4C	L	167	119	77	w
042	34	22	"	115	77	4D	M	170	120	78	x
043	35	23	#	116	78	4E	N	171	121	79	y
044	36	24	$	117	79	4F	O	172	122	7A	z
045	37	25	%	120	80	50	P	173	123	7B	{
046	38	26	&	121	81	51	Q	174	124	7C	|
047	39	27	’	122	82	52	R	175	125	7D	}
050	40	28	(	123	83	53	S	176	126	7E	~
051	41	29	)	124	84	54	T	177	127	7F	DEL
052	42	2A	*	125	85	55	U				

- `HT` , codice ottale 011 : *horizontal tab*, il carattere di tabulazione orizzontale, utile per compilare tabelle;
- `LF` , codice ottale 012 : *line feed*, salta alla riga successiva, ma, almeno in linea di principio, senza portare a capo il cursore;
- `VT` , codice ottale 013 : *vertical tab*, il carattere di tabulazione verticale (a dire il vero non molto usato, ma c'è);
- `FF` , codice ottale 014 : *form feed*, il salto pagina: se lo invii alla stampante causa l'uscita del foglio ed il caricamento del successivo;
- `CR` , codice ottale 015 : *carriage return*, il ritorno a capo del carrello (pensa alla macchina da scrivere) o del cursore, ma senza passare alla riga successiva.

Osserverai probabilmente due cose: la prima è che il codice ASCII fa uso di 128 numeri, mentre in un byte ce ne possono stare 256; la seconda è che mancano tutti i caratteri accentati. Il fatto di lasciare libero un bit ha una sua giustificazione: la trasmissione di informazioni su linee telefoniche, spesso disturbate, richiede dei controlli di correttezza. Uno di questi è il controllo di parità: ci si accorda che entro ciascun byte il numero di bit che hanno valore 1 debba essere sempre pari *(even parity)*, oppure sempre dispari *(odd parity)*. Di conseguenza l'ottavo bit viene alzato o abbassato in modo che la regola sia rispettata. Se il ricevente scopre che un byte viola la regola della parità butta il messaggio nel cestino, e chiede che venga ritrasmesso. Naturalmente, il cambiamento di due bit entro lo stesso byte vanifica l'effetto del controllo di parità.

Il secondo problema – quello dei caratteri accentati – è un po' più complesso. Il fatto è che se si tien conto di tutti gli accenti che compaiono in tutte le lingue – restringendoci a quelle che usano caratteri latini – il numero totale di caratteri aumenta considerevolmente: altro che un byte! E poi, vogliamo dimenticare il russo e il greco, per non parlare di arabo, cinese, giapponese e quant'altro? Un bel ginepraio! Se ne esce – quando si può – proprio sfruttando i codici tra 200_8 e 377_8 (sempre in ottale), e costruendo degli alfabeti che coprano le esigenze almeno di un gruppo di lingue. Ce ne sono diversi, e soprattutto non è facile mettere tutti d'accordo. Ma questo è un argomento che ci porterebbe troppo lontano: per il momento accontentati del codice ASCII.

Da quanto ti ho detto concluderai immediatamente che per rappresentare un carattere basta un byte di memoria: il tipo `char` è proprio quello giusto. Tutto corretto, salvo una piccola avvertenza: alcune funzioni di libreria che permettono, ad esempio, di trasferire da o su terminale un singolo carattere usano per il trasferimento un `int`. Normalmente non te ne accorgi, ma in qualche caso l'informazione è preziosa; ad esempio quando la funzione stessa ti restituisce dei codici numerici che riportano lo stato dell'operazione: questi ultimi codici devono usare necessariamente un campo più ampio, altrimenti si confondono coi caratteri.

Il fatto che i caratteri vengano rappresentati come dato `char` li rende del tutto equivalenti ad interi; piccoli – in un char non ci sta molto – ma interi. Ad esempio, se nel dato `c` ho memorizzato la lettera `A` (maiuscola) mi basta scrivere `c = c + 040` per trasformarla in minuscolo.

Per introdurre i caratteri come costanti, oltre al codice ottale, decimale o esadecimale (ad libitum) si può scrivere il carattere stesso tra apici. Così, ad esempio, se scrivo `c='a'` intendo che al dato `c` deve essere assegnato il codice ASCII della lettera a. Le scritture `c='a'`, `c=97`, `c=0141` e `c=0x61` sono di fatto equivalenti.

E come rappresento una scritta? Semplice: con un vettore di dati `char`: ogni elemento del vettore contiene un carattere. Con una avvertenza: per convenzione, il linguaggio C riconosce la fine di una scritta, o, come si dice abitualmente, di una *stringa di caratteri*, perché c'è uno zero (nel senso di zero binario, o carattere `NUL` del codice ASCII).

Per la scrittura di stringhe di caratteri si racchiude la stringa tra virgolette, esattamente come abbiamo fatto per il formato di conversione da passare come argomento alle funzioni `printf` o `scanf`. In altre parole, la scrittura `"Ciao, vecchio pirata!"` viene memorizzata come vettore di 22 dati di tipo `char` (non dimenticare nel conteggio il `NUL` che chiude la stringa).

Ed è anche il momento di chiarire il mistero del `\n` per andare a capo. Da quanto ho detto a proposito del codice ASCII avrai visto che per iniziare una nuova riga di scrittura occorre inserire un coppia di caratteri: `CR` per mandare i carrello a capo, e `LF` per passare alla riga successiva. La scrittura `\n` inserisce nella stringa un carattere `LF`; al `CR` aggiuntivo ci pensa qualcun altro (a che punto venga aggiunto dipende dal sistema).

E se volessi inserire in una stringa un codice che non ha un corrispondente stampabile? Semplice: basta usare la scrittura *xxx*, dove *xxx* sta per il codice ottale a tre cifre del carattere che voglio. Ad esempio, `"qui metto un FF\014"` è una stringa che contiene un carattere *form feed* dopo `FF`. Analogamente, la scrittura `"\000"` sta ad indicare una stringa composta dal solo carattere `NUL`, ovvero una stringa vuota.

4.1.4 Campi generici di bit

Un dato di tipo intero – qualunque esso sia – si può pensare anche semplicemente come una stringa di bit, a ciascuno dei quali il programmatore può assegnare il significato che vuole. Un uso automatico di questa possibilità riguarda le variabili che hanno un valore logico, o booleano, ossia possono assumere solo il valore vero o falso. Un secondo uso è: fanne quello che vuoi; io ti metto a disposizione delle istruzioni per manipolare i singoli bit.

Cominciamo con i valori logici. Non c'è in C una dichiarazione speciale: si usa una qualunque variabile di tipo intero. In linea di principio basterebbe un bit: 1 corrisponde a vero, 0 a falso. Ma modificare un singolo bit può essere un'operazione complessa – nel senso che richiede più di una istruzione elementare – quindi spesso conviene largheggiare e usare un qualunque tipo intero. Ma attenzione: macchine e linguaggi non sempre adottano le stesse convenzioni. Ce ne sono alcune che possono apparire più naturali di altre:

- ∘ si guarda il bit meno significativo, quindi un numero dispari sta per vero, uno pari sta per falso;
- ∘ se il numero è zero si intende falso, altrimenti si intende vero; a mia conoscenza, questa è una scelta comune, e dovrebbe essere lo standard per il linguaggio C – salvo idiosincrasie di chi ha scritto il compilatore.
- ∘ si guarda il bit più significativo, che equivale a intendere falso se il numero è positivo o nullo, e vero altrimenti;
- ∘ si stabilisce in modo rigido che 0 corrisponde a falso, e 1 a vero.

Chi ha più fantasia può immaginare molti altri modi. Ma si possono evitare i trabocchetti stando ben attenti a non confondere mai variabili che contengono dati interi con variabili logiche: le operazioni sono ben diverse.

Dunque veniamo alle operazioni. Anzitutto il confronto tra numeri interi o reali. Gli operatori utilizzabili sono `<` per strettamente minore, `<=` per minore o eguale, `==` per eguale, `>=` per maggiore o eguale, `>` per strettamente maggiore e `!=` per diverso. Presta ben attenzione al test di eguaglianza: non è un refuso. L'espressione `a=b` è un'assegnazione: prendi il valore di `b` e assegnalo ad `a`. L'espressione `a==b` equivale a domandarsi: `a` è eguale a `b`? La risposta è un valore logico: vero o falso. Confondere l'operatore `=` di assegnazione con l'operatore `==` di confronto di eguaglianza è errore molto frequente, e molto pericoloso.

Esercizio 4.1: Scrivi un breve programma che metta in evidenza gli effetti perversi della confusione tra `=` e `==`.

Naturalmente, i confronti funzionano anche se li uso su un dato `char` che contiene un carattere. Che il contenuto sia un carattere lo so io, ma la CPU no — non dimenticare che si tratta solo di un calcolatore! Quindi, se voglio mettere in ordine alfabetico una serie di parole posso ben sfruttare il fatto che i codici ASCII sono ordinati: il carattere `'a'` precede `'b'`, &c. Unico difetto: i caratteri maiuscoli precedono quelli minuscoli.

Esercizio 4.2: Scrivi un breve programma che converta una scritta da minuscolo a maiuscolo, o viceversa.

Si possono costruire anche espressioni logiche più complesse usando gli operatori *or*, *and* e *not*. Supponi che `w` e `z` siano valori logici (o espressioni logiche). Allora:

- ∘ *and*: `w && z` è vero se e solo se sia `w` che `z` sono veri; altrimenti è falso;
- ∘ *or*: `w || z` è falso se e solo se sia `w` che `z` sono falsi; altrimenti è vero;
- ∘ *not*: `!w` è vero se `w` è falso, e viceversa.

Ad esempio, se `c` è un carattere ho:

- ∘ `(c >= 'a') && (c <= 'z')` è vero se `c` è un carattere minuscolo;
- ∘ `((c >= 'A')&& (c <= 'Z')) || ((c >= 'a')&& (c <= 'z'))` è vero se `c` è un carattere alfabetico;
- ∘ `!(((c >= 'A')&& (c <= 'Z')) || ((c >= 'a')&& (c <= 'z')))` è vero se `c` non è un carattere alfabetico.

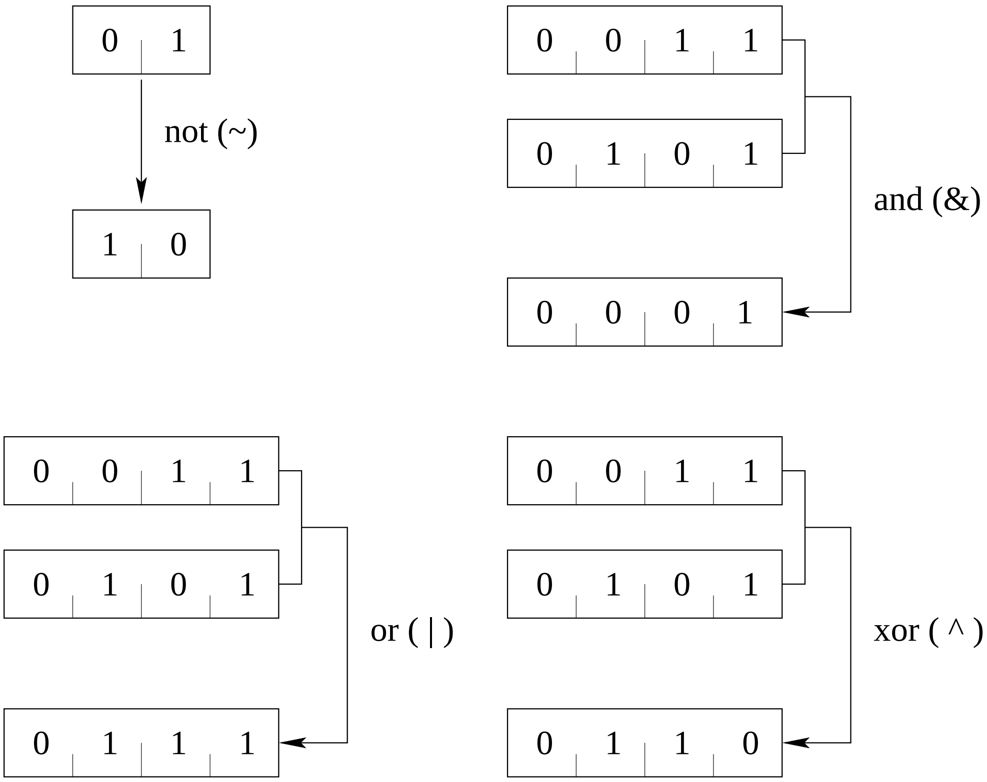

Figura 4.3. Le operazioni booleane bit–a–bit. l'operazione viene eseguita tra i due bit in colonna (ovvero: quelli che occupano la stessa posizione all'interno della cella). Sono rappresentati tutti i casi possibili.

Le parentesi specificano esattamente l'ordine in cui devono essere eseguite le operazioni.

Veniamo alle operazioni su campi di bit, dette anche operazioni *booleane*. È comune anche usare il termine *bitwise*, che tradurrò con *bit–a–bit*. Operano sull'insieme di valori $\{0, 1\}$. L'operazione *not*, o negazione, opera su ogni singolo bit; viene detta operazione *unaria*. Le altre operazioni vengono dette *binarie*, perché operano su ogni coppia di bit e producono un risultato. Quando vengono applicate a dati interi l'operazione viene eseguita indipendentemente sui bit che occupano la stessa posizione entro la cella di memoria. La figura 4.3 riassume tutti i casi possibili:

- *not* negazione o complemento a 1, denotato con l'operatore `~`: il bit 0 viene cambiato in 1, e viceversa;
- *and*, denotato con l'operatore `&`: il risultato è 1 se e solo se ambedue i bit sono 1, altrimenti è 0;
- *or*, denotato con l'operatore `|`: il risultato è 0 se e solo se ambedue i bit sono 0, altrimenti è 1;
- *xor*, denotato con l'operatore `^`, o *exclusive or*: il risultato è 1 se i bit sono diversi, altrimenti è 0.

Avrai notato la grande somiglianza con le operazioni logiche: in pratica si tratta di sostituire 1 o 0 a vero o falso. Ma non farti trarre in inganno: ciò che è diverso non è tanto il modo di operare, ma il dato su cui si opera. Se hai la sensazione che le tue idee siano confuse, permettimi di precisare meglio. Una cella che contiene un valore logico ha una certa configurazione di bit che viene interpretata come vero oppure falso, ed un'operazione logica determina il risultato senza agire singolarmente sui bit. Un'operazione bit–a–bit invece non ha nulla a che vedere sull'interpretazione del contenuto della cella come valore logico: i bit sono considerati singolarmente. Se un valore logico venisse rappresentato con un singolo bit, sarebbe la stessa cosa; ma così non è: la programmazione, o almeno il codice macchina, diventerebbe decisamente pesante.

Un esempio forse servirà a chiarire la situazione. Scriviamo un programmino molto semplice: definisce due variabili di tipo `char` assegnando loro i valori 2 e 4, rispettivamente; poi applica sia gli operatori bit-a-bit (`&`, `|` e `~`) che gli operatori logici (`&&`, `||` e `!`), e stampa i risultati come valori ottali. Può darsi che tu abbia bisogno delle informazioni del paragrafo 4.3 per comprendere appieno il risultato, ma i commenti che trovi più avanti potrebbero anche bastarti. Questo è il sorgente completo: [7]

```
#include <stdio.h>
int main()
{
  unsigned char j,k,l;
  j=02; k=04;
  l=j&k;  printf("%03o &  %03o = %03o\n",j,k,l);
  l=j&&k; printf("%03o && %03o = %03o\n",j,k,l);
  l=j|k;  printf("%03o |  %03o = %03o\n",j,k,l);
  l=j||k; printf("%03o || %03o = %03o\n",j,k,l);
  l=~k;   printf("~%03o = %03o\n",k,l);
  l=!k;   printf("!%03o = %03o\n",k,l);
  exit(0);
}
```

Prima di proseguire, rimboccati le maniche: batti il programma, compila ed esegui. Ma sta bene attento: non confondere `&&` con `&`, o `||` con `|`: sono cose ben diverse. I caratteri ripetuti non sono refusi, almeno qui. Eccoti il risultato in ottale (sulla mia macchina, col mio compilatore):

```
002 &  004 = 000
002 && 004 = 001
002 |  004 = 006
002 || 004 = 001
```

[7] Il formato di conversione `%03o` significa che voglio la stampa a 3 cifre, in ottale, e con degli zeri di riempimento al posto degli spazi.

```
~004 = 373
!004 = 000
```

Come vedi, le operazioni bit–a–bit e quelle logiche danno risultati ben diversi. Imprevedibili? Niente affatto. Tieni presenti queste informazioni, che dovrebbero bastare:

- il compilatore Gnu–CC memorizza il valore vero con 1 ed il valore falso con 0, ma interpreta come vero qualunque valore diverso da zero;
- i risultati, come ti ho detto, sono scritti in ottale, ma se vuoi la traduzione in binario eccola: $001_8 = 00000001_2$, $002_8 = 00000010_2$, $004_8 = 00000100_2$, $006_8 = 00000110_2$ e $373_8 = 11111011_2$.

Convinto? Forse stai pensando che non ci sono limiti alla stupidità di questa macchina. E che ti aspettavi?... è un calcolatore.

Oltre alle operazioni bit–a–bit esiste un'altra operazione di estrema utilità: lo *shift*, spostamento. L'operazione è rappresentata in figura 4.4, trabocchetti compresi. I bit vengono idealmente fatti scorrere verso destra o verso sinistra, come se avessero ricevuto uno spintone.[8] Naturalmente, in uno shift di un posto c'è un bit che viene buttato fuori dalla cella: viene semplicemente buttato via.[9] In compenso, una cella dal lato opposto resta vuota: cosa ci vien messo? Dipende. Se lo shift è verso sinistra, allora la cella che rimane vuota è quella che corrisponde al bit meno significativo; il vuoto viene riempito con uno zero. Se lo shift è verso destra la faccenda è più complicata: il comportamento dipende dal tipo di variabile in gioco. Se si tratta di un intero senza segno (`unsigned`), il campo vuoto viene riempito con uno zero: si parla talvolta di *shift logico*. Se si tratta di un intero con segno, allora il bit più significativo viene lasciato nello stato in cui si trova: si parla talvolta di *shift aritmetico*. Ti sembra un comportamento contorto? Prova a ripensarci dopo aver letto il paragrafo 4.3: ti renderai conto che questo comportamento è consistente col fatto che lo shift verso sinistra è di fatto una moltiplicazione per 2, e lo shift verso destra una divisione per 2.

Fin qui ho tacitamente sottinteso che lo shift sposti i bit di un posto per volta. Naturalmente l'operazione si può ripetere. Il tutto si può fare con una singola istruzione C: `j = j << k` sposta tutti i bit della cella `j` verso sinistra di `k` posizioni; `j = j >> k` li sposta verso destra.

---

[8] Le parole destra e sinistra sembrano un po' buffe: dove sta la destra e dove la sinistra nella memoria? Non ci sono, ma le posizioni dei bit sono numerate, esattamente come quelle dei bytes, e quindi ha senso associare alla numerazione interna dei bit una convenzione di scrittura. La più conveniente consiste nel rappresentare più a destra il bit che nella rappresentazione binaria è il meno significativo — la convenzione abituale per la scrittura di numeri.

[9] Questo non è del tutto vero: il bit buttato fuori viene spostato nel registro di stato della CPU. Un programmatore Assembler potrebbe controllarlo ed utilizzarlo, ma il linguaggio C non ha istruzioni che permettano di accedere direttamente al registro di stato.

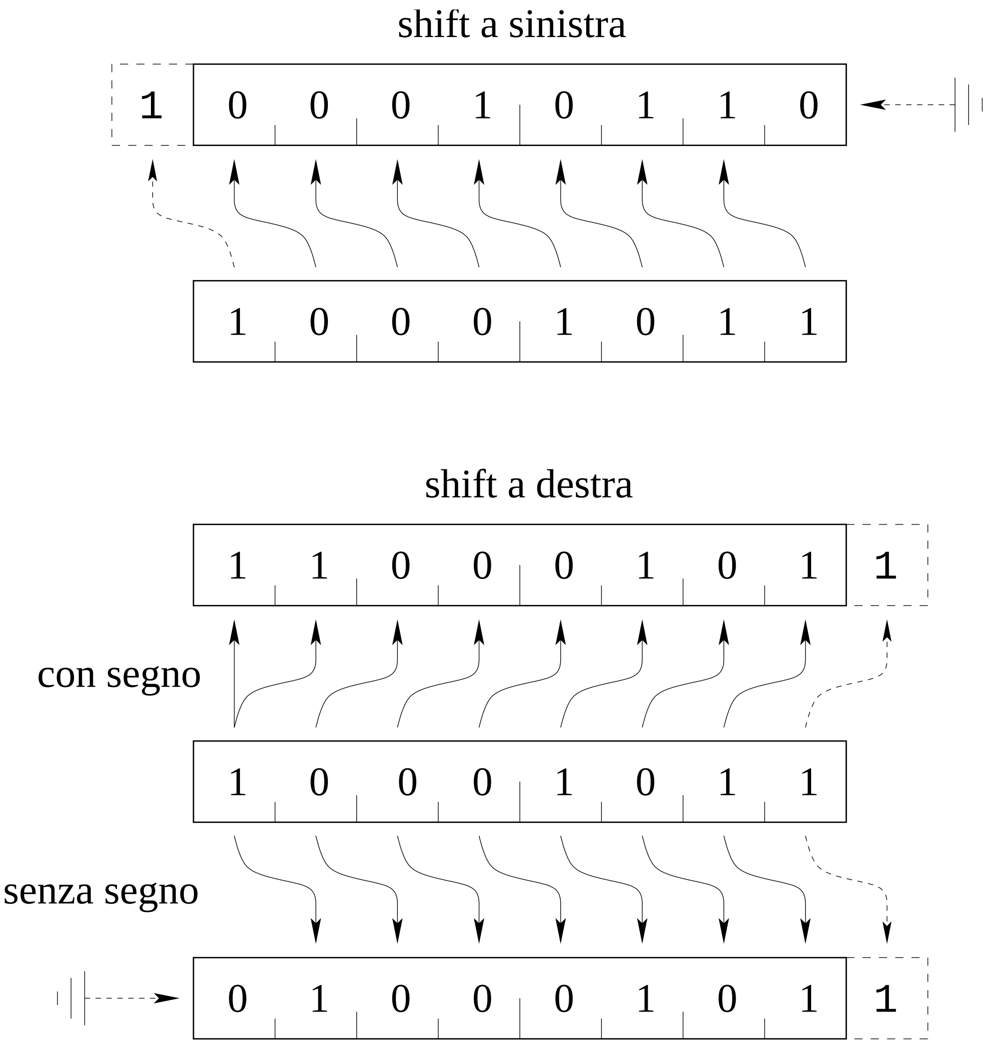

Figura 4.4. Le operazioni di shift. Nello shift a sinistra il bit più significativo viene perduto, ed il bit meno significativo viene sostituito da uno zero. Nello shift a destra occorre fare attenzione al segno: per dati `unsigned` il bit più significativo viene azzerato; per dati con segno il bit più significativo viene mantenuto invariato. Il bit meno significativo viene perduto.

4.2 Operazioni ed espressioni

Di operazioni ed espressioni abbiamo già parlato in più di un'occasione, ma può valer la pena di riassumere, e di completare, il quadro delle espressioni disponibili. L'esposizione qui è volutamente sintetica: mi dilungherò solo ad illustrare informazioni realmente nuove e non banali.

Distinguiamo tra:

- operatori *unari*, che agiscono su un solo dato;
- operatori *binari*, che agiscono su una coppia di dati e producono un

risultato.
- operatori di *assegnazione*: il trasferimento di un dato in una cella di memoria associata ad una variabile;

4.2.1 Operatori unari

Eccoti l'elenco:

`!`	Operatore logico di negazione: scambia il valore vero col valore falso. Opera su un valore logico: vedi il paragrafo 4.1.4.
`~`	Operatore bit–a–bit di negazione: esegue il complemento a 1 di ciascun bit. Opera su dati interi: vedi il paragrafo 4.1.4.
`-`	Posto davanti ad un dato numerico ne cambia il segno (il consueto segno meno dell'algebra).
(*tipo*)	Operatore di conversione, spesso indicato come *cast operator*: converte il tipo di dato. Ad esempio, `(double) x` forza la conversione del dato `x`, intero o floating, al tipo `double`.
`sizeof`	Si usa nella forma `sizeof(`*tipo*`)`, e restituisce la lunghezza in bytes del tipo specificato. Ad esempio, `sizeof(int)` restituisce la lunghezza in bytes del tipo `int`. Non opera su dati, ma sul tipo dei dati.
`++`,`--`	Operatori di incremento e decremento: `++` incrementa di 1 il dato intero a cui è applicato; `--` lo decrementa. Operano su variabili intere e, con qualche attenzione, anche su indirizzi; di quest'ultimo caso parleremo più avanti. Attenzione: prima di usarli leggi il paragrafo fino in fondo.
`&`	Operatore *address of*, o *indirizzo di*: applicato al nome di una variabile o di una funzione restituisce l'indirizzo di memoria a cui si trova il dato associato al nome. Ad esempio, `&k` restituisce l'indirizzo di memoria a cui si trova il dato `k`.
`*`	Applicato ad un indirizzo, identifica il dato che si trova a quell'indirizzo. Puoi pensarlo come l'operatore inverso di `&`: scrivere `*(&j)` è come scrivere `j`.

Gli operatori di incremento e decremento, pur non essendo indispensabili, sono spesso utili per compattare il codice sorgente. Occorre però aver ben presente la differenza tra anteporre l'operatore ad un nome di variabile o posporlo.

Le espressioni `++j` e `j++` isolate (nel senso che costituiscono da sole un'istruzione) sono del tutto equivalenti tra loro, ed equivalenti a `j=j+1`. Analogo discorso vale per `--j` e `j--`. Ad esempio, in un ciclo `for` è perfettamente equivalente usare una delle forme

```
for(j=0; j<100; j++)
for(j=0; j<100; ++j)
```

La differenza emerge quando si inserisce l'autoincremento in un'espressione più complessa. In tal caso occorre tener ben presente che:

- `j++` significa: usa il valore di `j` per valutare l'espressione, poi incrementa il valore di `j`;
- `++j` significa: incrementa il valore di `j`, ed usa il nuovo valore di `j` per valutare l'espressione.

L'esempio classico per illustrare la differenza è l'azzeramento di un vettore. L'istruzione

```
for(j=0; j<100; vet[j++]=0);
```

è molto sintetica, ma esegue tutto il lavoro. L'istruzione da eseguire prima di passare al ciclo successivo è: azzera `vet[j]`, poi incrementa `j`. Se scrivessi

```
for(j=0; j<100; vet[++j]=0);
```

non otterrei lo stesso risultato. L'istruzione da eseguire prima di passare al ciclo successivo sarebbe: incrementa `j`, poi azzera `vet[j]`. Alla fine del ciclo non avrei azzerato `vet[0]`, ed in compenso avrei azzerato `vet[100]` — probabilmente un'invasione di memoria, e certamente contrario alla mia intenzione di azzerare il vettore. Per avere il risultato corretto dovrei scrivere invece una delle due espressioni

```
for(j=-1; j<99; vet[++j]=0);
for(j=100; j>0; vet[--j]=0);
```

Nel secondo caso l'azzeramento viene eseguito partendo dal fondo del vettore, ma il risultato è corretto.

4.2.2 Operatori binari

Anche in questo caso scriviamo l'elenco.

`+,-,*,/`	Operatori aritmetici di addizione, sottrazione, moltiplicazione e divisione. Operano su dati interi o reali.
`%`	Resto della divisione tra interi.
`<,<=,>=,>`	Operatori di relazione: minore, minore o eguale, maggiore o eguale, maggiore. Si applicano a coppie di interi, reali o indirizzi, e restituiscono un valore logico.
`==,!=`	Operatori di eguaglianza: eguale, diverso. Si applicano a coppie di interi, reali o indirizzi, e restituiscono un valore logico.
`&&,||`	Connettivi logici: *and*, *or*. Operano su coppie di valori logici, e restituiscono un valore logico.
`&,|,^`	Operatori bit–a–bit (*bitwise*): *and*, *or*, *xor* (*exclusive or*). Operano su coppie di interi di egual lunghezza.
`<<,>>`	Operatori di shift: vedi il paragrafo 4.1.4.

4.2.3 Operatori di assegnazione

L'operatore `=` è il prototipo. Come ho già detto più volte, significa: prendi il risultato dell'espressione che c'è a destra, e mettilo nella cella di memoria associata alla variabile che sta a sinistra.

In linea di principio, se sai come usare l'operatore `=` è sufficiente, ma il linguaggio C mette a disposizione anche operatori di assegnazione che richiedono un'operazione aggiuntiva. Ce ne sono tanti, ma tutti obbediscono

Tavola 4.2. Elenco degli operatori in ordine decrescente di precedenza. Gli operatori unari si applicano all'espressione che li segue; gli operatori binari che stanno sulla stessa linea di priorità vengono valutati nell'ordine in cui sono scritti.

! ~ ++ -- - (*tipo*) & * (sizeof)
* / %
+ -
<< >>
< <= >= >
== !=
&
^
|
&&
||
= += -= *= /= %= &= ^= |= <<= >>=

alla stessa logica. L'istruzione `j+=k` significa: somma `k` a `j`. È equivalente a `j=j+k`. La stessa regola vale per gli operatori `-=`, `*=`, `/=`, `&=`, `|=`, `^=`, `<<=`, `>>=`.

La forma generale è: $\langle \textit{risultato} \rangle$ $\langle \textit{op} \rangle$=$\langle \textit{espressione} \rangle$. Il significato è: calcola l'espressione a destra; applica l'operazione che precede il segno `=` alla coppia costituita dalla variabile che compare come risultato e da quello che hai appena calcolato; quello che ottieni memorizzalo nel risultato.

Il linguaggio C ammette anche un'interpretazione ampia del significato della parola espressione (logica, aritmetica o quant'altro): anche un'assegnazione è considerata un'espressione, il cui risultato è proprio il valore che è stato assegnato. Più che le parole vale un esempio: l'istruzione `j = 2+ (k=m+1)` significa: calcola `m+1` e memorizzalo in `k` — questo è il risultato dell'espressione `k=m+1`; addiziona `2` al valore di `k` che hai appena calcolato; quello che hai ottenuto mettilo in `j`. Alla fine della sequenza di operazioni sia `k` che `j` sono stati modificati.

4.2.4 Ordine di precedenza

Nel caso di espressioni aritmetiche siamo abituati da tempo a stabilire in quale ordine debbano essere eseguite le singole operazioni: moltiplicazioni e divisioni vanno eseguite nell'ordine in cui sono scritte, e prima di addizioni e sottrazioni. Se vogliamo modificare l'ordine di esecuzione ci basta usare delle parentesi.

Ma che accade se mescoliamo altri operatori a quelli aritmetici? Occorre tener conto delle regole di precedenza. La tavola 4.2 raccoglie la lista degli operatori, in ordine crescente di precedenza dal basso verso l'alto. Nel dubbio, una coppia di parentesi in più non guasta.

4.3 La rappresentazione dei dati interi

Parlare dei numeri interi sembra un po' buffo: li conosci dalle elementari. Eppure c'è probabilmente qualcosa che non conosci su come la CPU vede i dati interi, e come esegue i calcoli. È quello che vorrei svelarti in questo paragrafo. Ma per entrare a sufficienza nei dettagli mi servono un paio di digressioni di carattere matematico: la rappresentazione dei numeri in base 2, e l'aritmetica modulare.

4.3.1 Prima digressione: la rappresentazione dei numeri su una base

Come ben sai, per rappresentare i numeri naturali su una base devi:
- ◦ scegliere un numero $b > 1$, che chiamiamo *base di numerazione*; (domanda: perché 1 non va bene? Prova a rispondere.)
- ◦ scegliere b simboli per rappresentare i numeri $0, \ldots, b-1$; li chiameremo *cifre*;
- ◦ scomporre ciascun numero naturale in una somma di addendi della forma $c_0 b^0 + c_1 b^1 + \ldots + c_n b^n$, dove n dipende dalla grandezza del numero, con la prescrizione che debba valere $0 \leq c_j < b$ per $j = 0, \ldots, n$; puoi aggiungere anche la prescrizione che sia $c_n \neq 0$;
- ◦ stabilire la convenzione del valore posizionale della cifra: le cifre $c_0, c_1, \ldots, c_n$ vengono scritte consecutivamente, da destra verso sinistra.

Come conseguenza di queste convenzioni, il numero $c_0 b^0 + c_1 b^1 + \ldots + c_n b^n$ viene rappresentato come $c_n \ldots c_1 c_0$. La scelta della base b di numerazione è del tutto arbitraria: noi abitualmente usiamo 10 — che casualmente è il numero delle dita delle nostre mani. Ma qui sto parlando di basi diverse; quindi adotterò l'ulteriore convenzione di aggiungere a ciascun numero un pedice che indichi la base utilizzata. Così, ad esempio, 13_{10} è solo una complicazione della scrittura che usiamo tutti i giorni, perché la base 10 è quella abituale, mentre 15_8 indicherà che sto usando la base 8, e mi sto servendo delle cifre $0, 1, 2, 3, 4, 5, 6, 7$. Naturalmente, basta un attimo per verificare che $13_{10} = 15_8$.

Come si passa da una base all'altra? Convertire un numero da una base b qualsiasi alla base 10 è facile: basta usare le regole che ti ho enunciato sopra. Ad esempio, se avessi il numero 157_8 non dovrei fare altro che calcolare $7 \times 8^0 + 5 \times 8^1 + 1 \times 8^2 = 7_{10} + 40_{10} + 64_{10} = 111_{10}$. Fare l'operazione contraria sembra più difficile, ma se ci pensi un momento scopri una regola molto semplice: *se voglio scrivere un numero in base b mi basta calcolare i resti delle divisioni successive di quel numero per b, e scriverli da destra a sinistra*. Ad esempio, se voglio scrivere 231_{10} in base 8 devo procedere così (svolgo i calcoli in rappresentazione decimale, con le notazioni della maestra delle elementari): $231 : 8 = 28$ con resto 7; $28 : 8 = 3$ con resto 4; $3 : 8 = 0$ con resto 3. Adesso scrivo i resti da destra a sinistra, e trovo $231_{10} = 347_8$. L'operazione si può tradurre in uno schema di calcolo: ti basta procedere compilando una tabellina del tipo

231	7
28	4
3	3
0	

La regola è semplice: scrivi il numero sulla colonna di sinistra; dividi per 8 e scrivi il risultato sotto il numero, ed il resto della divisione sulla colonna di destra allineato col numero; poi passa alla seconda riga e ripeti il procedimento fin che trovi il risultato zero. Infine leggi la colonna dei resti dal basso verso l'alto, scrivendo le cifre da sinistra a destra. Il metodo funziona con qualsiasi base, lo vedi subito. Ma... ti sei accorto che ti ho programmato?

Adesso prova a riflettere. I metodi che ti ho descritto sembrano privilegiare la base 10: per passare da una rappresentazione in base b alla base 10 faccio delle moltiplicazioni; per passare dalla base 10 alla base b faccio delle divisioni. Dunque il procedimento non è simmetrico. Ancor peggio: se dovessi convertire un numero da una base b ad una base b' dovrei passare dalla base b alla base 10 facendo delle moltiplicazioni, e poi dalla base 10 alla base b' facendo delle divisioni.

Che la base 10 sia privilegiata rispetto alle altre? No, ovviamente: è la nostra testa che è distorta! Noi siamo tanto abituati a calcolare in base 10 che non sappiamo fare i conti se non in quella base. Colpa della programmazione a cui ci ha sottoposto la maestra alle elementari!

Ma come faremmo a calcolare in una base diversa da 10? Semplice: esattamente come facciamo in base 10, ma usando delle tabelline di addizione e moltiplicazione diverse. Mi spiego. Se ben ci pensi, osserverai che tutte le regole per addizione, sottrazione, moltiplicazione e divisione che hai faticosamente appreso alle scuole elementari si reggono su un criterio semplice: devi riuscire a lavorare su una cifra per volta. In particolare, se sai eseguire addizione e moltiplicazione tra due numeri che si rappresentano con una cifra sola sei in grado di svolgere tutte le altre operazioni.

Perché gli algoritmi che ti ha insegnato la maestra funzionano? Per un motivo molto semplice: si fondano, oltre che sulle regole per la rappresentazione in una base che ti ho enunciato qui sopra, sulle proprietà commutativa ed associativa della somma e della moltiplicazione tra interi, sulla proprietà distributiva della moltiplicazione rispetto alla somma, e sulle proprietà che $a + 0 = a$, $a \cdot 0 = 0$ e $a \cdot 1 = a$, per qualunque numero naturale a. Ora, tutto questo è indipendente dalla base di numerazione. Quindi le regole che applichi con la base 10 devono applicarsi in qualunque base. Basta costruirsi la tavola pitagorica. Ad esempio, la tavola pitagorica per la base 8 è rappresentata in figura 4.5.

Lascio a te l'esercizio di costruire la tavola per l'addizione. Mi limito a farti osservare che se sapessi fare le addizioni e le moltiplicazioni in una base b qualsiasi con la stessa facilità con cui le fai in base 10 potresti convertire i numeri da una base b' alla una base b usando solo addizione e moltiplicazione, comunque tu scelga b e b'. Una bella comodità: ricordi quanto la divisione

1	2	3	4	5	6	7	10
2	4	6	10	12	14	16	20
3	6	11	14	17	22	25	30
4	10	14	20	24	30	34	40
5	12	17	24	31	36	43	50
6	14	22	30	36	44	52	60
7	16	25	34	43	52	61	70
10	20	30	40	50	60	70	100

Figura 4.5. La tavola pitagorica per la moltiplicazione in base 8

fosse più difficile di una moltiplicazione?

E veniamo al calcolatore. Il modo stesso in cui viene costruito privilegia la base 2: non ci sono dubbi. D'altra parte, la base 2 ha qualcosa di speciale: è la base minima che si possa utilizzare, e questo conterà ben qualcosa. E infatti te ne rendi subito conto se provi a costruire le tabelline. Per l'addizione basta sapere che $0_2 + 0_2 = 0_2$, $0_2 + 1_2 = 1_2 + 0_2 = 1_2$ e $1_2 + 1_2 = 10_2$. Per la moltiplicazione la tavola pitagorica si riduce praticamente a nulla: le proprietà dello zero e dell'unità rispetto alla moltiplicazione dicono tutto.

La cosa più divertente poi riguarda la divisione. Ricordi quanto fosse complicato stabilire quante volte 37 è contenuto in 245? Il problema è che le risposte possibili sono a priori 10, e tu devi scegliere quella giusta. Con la base 2 la risposta può essere solo 0 oppure 1, cioè il divisore nel dividendo ci sta o non ci sta. Facile, no? Ti immagini una scuola elementare senza le tabelline?

Naturalmente, come in ogni cosa, c'è il rovescio della medaglia. Arrivato al traguardo della maggior età, 18_{10} anni, dovresti dire che hai compiuto $10\,010_2$ anni. E se poi, conquistata la patente di guida, provassi a realizzare il sogno di acquistare un'automobile dal costo di $10\,000_{10}$ Euro, il concessionario ti chiederebbe di sborsarne $10\,011\,100\,010\,000_2$.

Nel mondo informatico, proprio per evitare di scrivere numeri lunghi come treni, si preferisce adottare due basi che ben si adattano alla struttura binaria della macchina, e precisamente le basi 8 e 16. Della base 8 abbiamo già parlato. Per la base 16 le cose non cambiano molto: si tratta solo di inventare altre 6 cifre. Con un'ardita intrusione in campo alfabetico qualcuno ha deciso che le cifre sono $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$. Non è poi tanto

Tavola 4.3. Tabella di conversione tra le basi 2, 8 e 16. Per comodità è riportata anche la colonna della base 10.

base 2	base 8	base 16	base 10
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	a	10
1011	13	b	11
1100	14	c	12
1101	15	d	13
1110	16	e	14
1111	17	f	15

difficile: basta ricordare che $a_{16} = 10_{10}$, $b_{16} = 11_{10}$ &c, fino a $f_{16} = 15_{10}$. Il resto vien quasi da sé. L'uso frequente ha indotto anche a coniare i termini *ottale* (*octal*) per la rappresentazione in base 8 ed *esadecimale* (*hexadecimal*) per la base 16.

La comodità delle basi 8 e 16 ha una motivazione molto semplice: sono potenze di 2, e quindi la conversione con la base 2 è una faccenda banale. La tabella di conversione la trovi nella tavola 4.3; per comodità ci ho aggiunto anche la colonna dei valori decimali. L'uso della tabella è illustrato in figura 4.6: per tradurre da binario ad ottale basta raccogliere i bit in gruppi di tre, procedendo da destra a sinistra, e sostituire ciascun gruppo di tre bit con la corrispondente cifra ottale; per tradurre da binario ad esadecimale si procede in modo analogo, con gruppi di quattro bit. La conversione da ottale o esadecimale a binario si effettua col procedimento opposto: ad ogni cifra si sostituisce il gruppetto di tre o quattro bit. Ad esempio, i 18_{10} anni della maggiore età diventano 22_8 oppure 12_{16}, e i $10\,000_{10}$ Euro diventano 23420_8 oppure 2710_{16}.

4.3.2 Seconda digressione: l'aritmetica modulare

Primo passo: si fissa un numero $N > 1$ e si dividono i numeri interi in classi di equivalenza $\pmod N$. Precisamente, si dice che due numeri k, j sono

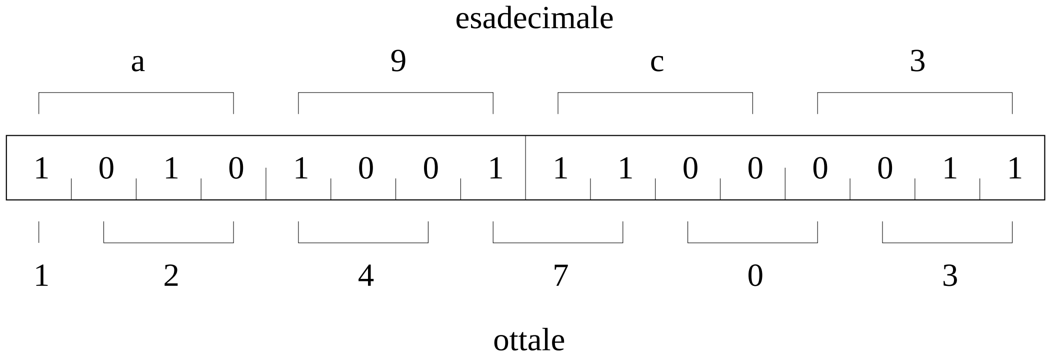

Figura 4.6. La conversione da binario ad ottale o esadecimale per un numero di 16 bit. Il numero binario $1\,010\,100\,111\,000\,011_2$ diventa 124703_8 in ottale e $a9c3_{16}$ in esadecimale.

equivalenti se $k-j$ è multiplo di N; in tal caso scriveremo $k \equiv j$.

Per chi non fosse ancora familiare con le classi di equivalenza occorre qualche riga in più. Dire che una relazione, come quella che abbiamo definito qui sopra, è *di equivalenza* significa che gode delle proprietà riflessiva, simmetrica e transitiva. Nel nostro caso queste tre proprietà valgono certamente. Infatti, indicati con k, j, l tre numeri interi qualsiasi abbiamo:

- ∘ $k \equiv k$, perché $k-k=0$ è multiplo di N;
- ∘ se $k \equiv j$ allora è anche $j \equiv k$, perché se $k-j$ è multiplo di N anche $j-k$ lo è;
- ∘ se $k \equiv j$ e $j \equiv l$ allora è anche $k \equiv l$, perché $k-l=(k-j)+(j-l)$, e la somma di due multipli di N è ancora multiplo di N.

Ora, è un fatto generale che una relazione di equivalenza su un insieme ci permette di ripartire l'insieme in sottoinsiemi disgiunti, ponendo in ciascun sottoinsieme tutti gli elementi tra loro equivalenti. Se la relazione è l'equivalenza $\pmod N$ tra interi che stiamo considerando, allora i sottoinsiemi disgiunti sono esattamente N. Ad esempio, se $N=3$ troviamo le tre classi

$$
\begin{aligned}
&\{\ldots,-9,-6,-3,0,\,3,\,6,\,9,\ldots\}\\
&\{\ldots,-8,-5,-2,1,\,4,\,7,\,10,\ldots\}\\
&\{\ldots,-7,-5,-1,2,\,5,\,8,\,11,\ldots\}
\end{aligned}
$$

Ciò che ci interessa è che le classi di equivalenza $\pmod N$ sono compatibili con le operazioni aritmetiche di somma algebrica e di moltiplicazione (la divisione è una bestia a parte). Precisamente:

- ∘ se $k \equiv k'$ e $j \equiv j'$ allora $k+j \equiv k'+j'$;
- ∘ se $k \equiv k'$ e $j \equiv j'$ allora $kj \equiv k'j'$.

Questa è un'affermazione che non ti sarà difficile dimostrare in pochi minuti. La conseguenza è che possiamo tranquillamente parlare di operazioni di addizione e moltiplicazione tra classi, e non semplicemente tra interi.

Ora, quando si hanno delle classi di equivalenza è utile individuare dei rappresentanti delle classi. Ad esempio, possiamo ben eleggere come rappresentanti i numeri $0,\ldots,N-1$. Ma non è detto che sia la scelta migliore. Per fare un altro esempio, possiamo scegliere, $-2,-1,\ldots,0,\ldots,N-3$ (natural-

1 0 1 0 0 1 $\boxed{0\ 1\ 0\ 1\ 1\ 0\ 0\ 1}$

Figura 4.7. La rappresentazione binaria su 8 bit equivale a troncare qualunque numero binario mantenendo solo le 8 cifre meno significative. Tutto il resto non può essere rappresentato. Di conseguenza, tutti numeri non negativi tra loro equivalenti $\pmod{2^8}$ hanno la stessa rappresentazione.

mente supponendo $N > 3$). In tal caso abbiamo eletto come rappresentanti anche dei numeri negativi.

Ebbene, la rappresentazione degli interi sul calcolatore si fonda proprio sulla combinazione di rappresentazione di numeri in base 2, più l'aritmetica modulare, più la scelta dei rappresentanti delle classi di equivalenza. Della base 2 abbiamo discusso, e non c'è molto da aggiungere; vediamo il resto.

L'aritmetica modulare si introduce in conseguenza delle limitazioni delle celle di memoria destinate a contenere gli interi. Se usiamo una cella di n bit è giocoforza scegliere $N = 2^n$. Infatti, se abbiamo n caselle ciascuna delle quali può contenere la cifra 0 oppure 1 possiamo riempire queste caselle in 2^n modi diversi.[10] Se pretendiamo che ciascuna configurazione corrisponda ad una classe distinta dalle altre abbiamo $N = 2^n$ classi.

Come sono fatte queste classi di equivalenza? Semplice: guarda la figura 4.7. Supponi di aver un qualunque numero naturale scritto in rappresentazione binaria, ma di avere a disposizione solo n bit (nella figura $n = 8$). Seguendo le regole del valore posizionale della cifra, inizi a scrivere le cifre da destra verso sinistra, partendo da quella delle unità. Se il numero richiede meno di n cifre binarie, aggiungi degli zeri a sinistra fino ad esaurimento degli n bit.[11] Ma se il numero richiede più di n cifre binarie non hai abbastanza spazio per rappresentarlo tutto: lo devi decapitare. Ebbene, la decapitazione corrisponde proprio a sostituire il tuo numero con un altro

[10] Nella prima casella possiamo mettere 0 oppure 1, quindi abbiamo due possibilità; una volta fissato il contenuto della prima casella, per la seconda abbiamo due possibilità, e quindi un totale di 2×2 possibilità per le prime due caselle; avendo fissato il contenuto delle prime due caselle, per la terza abbiamo due possibilità, e quindi un totale di $2 \times 2 \times 2$ possibilità per le prime tre caselle. Procedendo a questo modo arriviamo alla conclusione che per n caselle le possibilità sono 2^n, che è quanto affermato. Se questa dimostrazione ti sembra poco elegante, puoi riformularla in forma induttiva. Per $n = 1$ l'affermazione è palesemente vera, perché per una casella le possibilità sono 2. Supponiamo che sia vera fino ad $n - 1$; allora per ciascuna scelta del contenuto delle prime $n - 1$ caselle posso scegliere in 2 modi il contenuto della n–esima. Quindi per n caselle ho $2^{n-1} \times 2 = 2^n$ possibilità.

[11] Non puoi dire semplicemente: ignoro il resto, come faccio abitualmente. I bit in memoria ci sono, devono pur avere un valore.

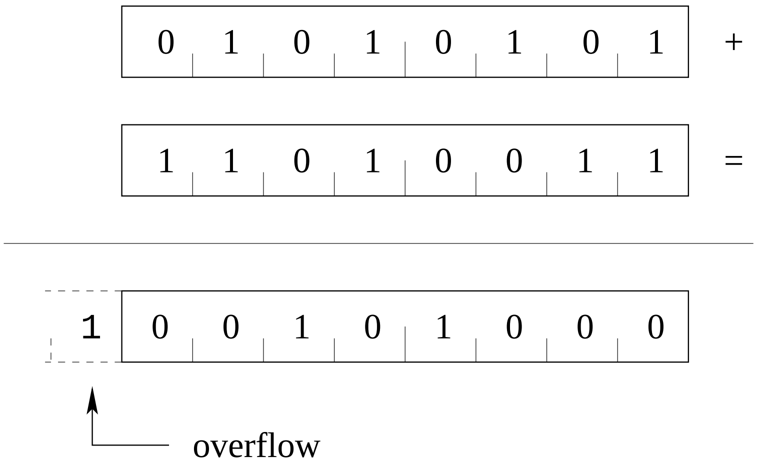

Figura 4.8. L'addizione tra due numeri può provocare un *overflow*, ossia la fuoruscita di un bit dalla cella che rappresenta il risultato. In tal caso il risultato viene troncato.

che appartenga alla stessa classe di equivalenza $\pmod{2^n}$. Il bello è che la scelta è unica: non puoi prendere gli n bit meno significativi in due maniere diverse!

E i numeri negativi? Un bit può assumere solo i valori 0 o 1; quindi la sola cosa che puoi fare è sostituire il numero negativo con uno non negativo ad esso equivalente, e rappresentare quest'ultimo. [12]

Veniamo all'addizione. Ti ho già spiegato che il metodo di addizionare i numeri scrivendoli in colonna che hai faticosamente appreso alle scuole elementari funziona benissimo. Trovi un esempio interessante in figura 4.8. Esegui l'addizione partendo dalla colonna di destra, e ricordando che in binario $1+1=10$, e quindi scrivi 0 e tieni 1 come riporto, &c. Quando arrivi alla fine ti accorgi che ti è avanzato un riporto 1, ma nella tua cella non c'è spazio per memorizzarlo. Non puoi farci nulla: devi decapitare il risultato. Si dice che l'operazione ha provocato un *overflow*, o fuoruscita di un bit. Giusto o sbagliato? Dipende dai punti di vista. Se quello che ti importa è

[12] A dire il vero, esiste un'altra possibilità: degli n bit a disposizione ne uso $n-1$ per rappresentare il numero, e uno per il segno, con la convenzione che 0 corrisponde al segno $+$ e 1 corrisponde al segno $-$. In tal caso sarei costretto a rappresentare i numeri $\pmod{2^{n-1}}$, ma dovrei anche trattare numeri negativi e positivi in modo diverso, come se appartenessero ad insiemi differenti. Inoltre, lo zero non avrebbe una rappresentazione univoca, perché potrei scriverlo indifferentemente con $+0$ o -0. Detto in modo più preciso, ciascuno dei due sottoinsiemi dei numeri positivi e negativi avrebbe un suo zero. In effetti questo tipo di rappresentazione è stato usato in passato da alcuni costruttori, magari con altre varianti. Tuttavia non presenta nessun reale vantaggio rispetto al metodo che sto esponendo, e che, a mia conoscenza, è ormai universalmente adottato.

l'aritmetica sui numeri naturali il risultato è palesemente sbagliato. Ma se pensi all'addizione come operazione tra classi di equivalenza modulo 2^n il risultato è perfettamente corretto. L'aritmetica del calcolatore lavora sulle classi di equivalenza modulo 2^n, non sui numeri naturali o interi che dir si voglia: questo è un fatto.

Dove va a finire il bit di overflow? Ai fini del calcolo è perso: non c'è speranza. Ma la CPU non lo butta via. Come ti ho detto, nella CPU c'è un registro di stato che mantiene, come dice il nome, lo stato della macchina. Uno dei bit di questo registro viene alzato se l'ultima operazione eseguita dalla CPU ha provocato un overflow; questo bit viene chiamato *OF*, una abbreviazione per *Overflow Flag*. [13] Il linguaggio C non mette a disposizione istruzioni che possano controllare direttamente il valore di questo bit. Ma un programmatore che usasse il linguaggio Assembler sarebbe perfettamente in grado di farlo.

Esercizio 4.3: Sapresti scrivere un programma che stampa la tavola pitagorica per la base 16? Quello che hai imparato fin qui dovrebbe metterti in condizione di fare tutto il calcolo necessario. Può darsi che la stampa ti crei qualche piccola difficoltà: devi imparare bene come scrivere il formato di conversione. Perché non provare a sfidare il manuale?

Esercizio 4.4: Questo non è un esercizio di programmazione, ma ti può servire per verificare se hai capito l'aritmetica modulare. Tra le tante cose che hai appreso alle scuole elementari c'è probabilmente anche la *prova del nove*. Non che funzioni sempre: se fallisce, il calcolo è sicuramente sbagliato; se risulta corretta, il calcolo potrebbe essere sbagliato, ma in modo perverso. Se questa faccenda ti risulta del tutto misteriosa dimentica pure l'esercizio. Altrimenti prova a rispondere alla domanda: perché questa prova funziona? Esiste una prova analoga in una base diversa da 10?

Esercizio 4.5: Un'altra domanda che non ha molto a che vedere con la programmazione. Tra i criteri di divisibilità che hai appreso c'è anche quello dell'undici: prendi un numero, somma tra loro le cifre di posto pari, poi somma tra loro le cifre di posto dispari; se la differenza tra queste due somme è multiplo di 11 allora il numero è divisibile per 11. Perché? Puoi enunciare un criterio simile se usi una base di numerazione diversa da 10?

E i numeri negativi? Anche questi hanno sempre un equivalente $(\mathrm{mod} 2^n)$ positivo e rappresentabile su n bit. Per quanto possa sembrare sorprendente, è del tutto vero: una volta stabilito che addizione e moltiplicazione sono

[13] Il termine *flag*, letteralmente una bandiera, viene ampiamente usato per indicare se un determinato evento si sia verificato o no. Nella programmazione identifica una variabile che può assumere il valore 0 o 1, ed il cui valore viene definito in conseguenza dell'evento preso in considerazione. Molto spesso si usa proprio un bit, in una posizione predefinita di un byte o di una word, o quant'altro. Un po' come una bandierina che viene alzata per mandare un segnale.

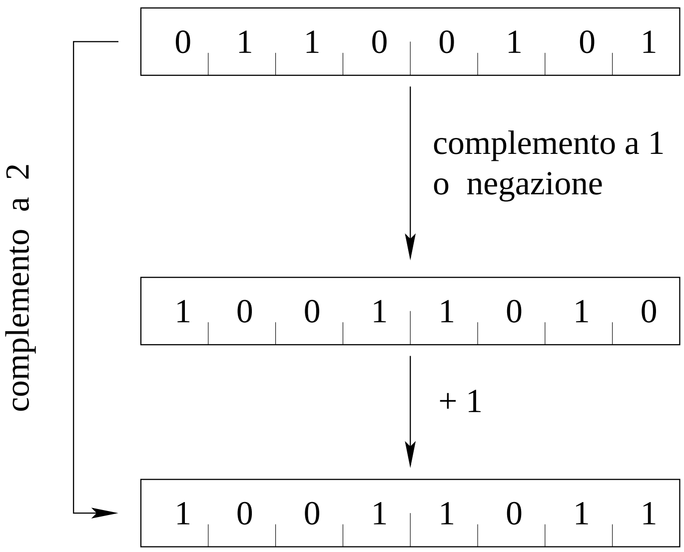

Figura 4.9. Il complemento a 2 di un dato intero: si cambiano tutti i bit 0 in 1 e viceversa, e si aggiunge 1 al risultato.

operazioni sulle classi $(\mathrm{mod} 2^n)$ non c'è null'altro da aggiungere. Ma facciamo un esempio che chiarisca meglio come funziona tutta la faccenda.

Se lavoro con celle di 8 bit, posso rappresentare i numeri non negativi compresi tra 0 e $11\,111\,111_2 = 255_{10}$. Più in generale, con n bit posso rappresentare tutti i numeri compresi tra 0 e $2^n - 1$; ma qui fisserò $n = 8$, perché vorrei rendere comprensibile l'esempio. Dunque, lavoro con classi di equivalenza $(\mathrm{mod}\ 256)$. Come rappresento il numero -1? Risposta: costruisco la classe di equivalenza di -1, e trovo

$$\ldots, -513, -257, -1, 255, 511 \ldots .$$

Tra questi numeri il solo compreso tra 0 e 255_{10} (e quindi rappresentabile su 8 bit senza ricorrere a decapitazioni) è 255_{10}. Dunque, il rappresentante di -1 è $11\,111\,111_2$. Procedendo allo stesso modo potresti trovare che -2 si rappresenta con $11\,111\,110_2$, e -3 con $11\,111\,101_2$, &c.

Supponiamo di avere un numero positivo k. Esiste un modo semplice per passare dal positivo al negativo, ossia per trovare il rappresentante di $-k$ tra le classi di equivalenza $(\mathrm{mod}\ 2^n)$? Domanda complicata, posta in questo modo! Quale numero negativo? Ogni classe ne contiene infiniti! Ma ti rendi subito conto che ogni classe ne contiene uno solo compreso tra -255_{10} e 0. Per trovarlo si esegue un'operazione molto semplice, detta *complemento a 2* del numero. È illustrata in figura 4.9. Si compone di due passi: prima si esegue il *complemento a 1*, o *negazione*, che consiste nello scambiare i valori 0 e 1 in ciascun bit; poi si somma 1. Se nel sommare 1 si incappa in un overflow poco male. Se provi con qualche numero troverai un paio di situazioni buffe: il complemento a 2 di 00 000 000 è ancora 00 000 000, e il complemento a 2 di 10 000 000 è ancora 10 000 000. Naturalmente, questo non dipende dal fatto che stiamo considerando proprio il caso di 8 bit. Per un n generico, il primo caso è conseguenza di $\sum_{j=0}^{n-1} 2^j = 2^n - 1$, ed è proprio il caso in cui sommando 1 dopo la negazione si incappa in un overflow; il secondo caso invece si riconduce al fatto che $2^{n-1} \equiv -2^{n-1} \pmod{2^n}$.

L'operazione di sottrazione non è un problema: $a - b$ si calcola come $a + (-b)$. Non è una battuta, o una banalità: significa che prima complementi b a 2, e trovi un numero della classe di equivalenza di $-b$; poi fai una semplice addizione.

La divisione invece è come un capello nella minestra. Il fatto è che non rispetta le classi di equivalenza. Quindi è da trattare con qualche precauzione. Ma non c'è da meravigliarsi, tutto sommato: non è ben definita neppure sui numeri interi. Dobbiamo rassegnarci: ci vuole almeno l'estensione ai razionali. In buona sostanza, la divisione non crea particolari problemi – a parte l'inevitabile troncamento – fin che non si superano i limiti della rappresentazione, e quindi non si chiama realmente in causa l'aritmetica modulare.

4.3.3 La rappresentazione degli interi

Se hai ben assimilato le due digressioni di carattere matematico, hai in mano tutti gli strumenti per capire la rappresentazione degli interi: la CPU memorizza i numeri in base 2, ed esegue le operazioni di addizione e moltiplicazione secondo le regole dell'aritmetica modulo 2^n, dove n è il numero di bit della cella che contiene l'intero.

Se ci accontentiamo di operare solo su numeri non negativi, allora l'aritmetica è quella consueta sui numeri naturali fin quando non si verifica un overflow. Quindi basta non eccedere il limite del massimo numero rappresentabile nella cella di memoria destinata all'intero. Se la cella contiene n bit, il massimo rappresentabile è $2^n - 1$. Questo caso è quello che si verifica per i dati di tipo `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int` e `unsigned long long int`.

Se invece vogliamo tener conto anche dei numeri negativi dobbiamo solo scegliere come rappresentanti delle classi di equivalenza $\pmod{2^n}$ anche dei numeri negativi, che andranno a rimpiazzare i corrispondenti positivi: basta trovare un accordo sulla suddivisione, che resta peraltro arbitraria. Qualunque sia questa scelta, resterà comunque vero che l'aritmetica modulare coincide con quella sui numeri interi fin che non si superano i limiti della rappresentazione.

La convenzione abituale, e tutto sommato la più comoda, è questa: tutti i numeri che hanno il bit più significativo alzato vengono interpretati come negativi; quelli che hanno il bit più significativo abbassato vengono interpretati come positivi. Per questo motivo il bit più significativo viene detto *bit di segno*. Così, facendo sempre riferimento al caso di celle di 8 bit, il numero $10\,010\,110_2$ è negativo: per conoscerne il valore in decimale dobbiamo farne il complemento a 2, trovando $01\,101\,010_2$, e poi convertire in decimale anteponendo il segno $-$; otteniamo così -106_{10}. Invece il numero $01\,100\,111_2$ è positivo, e si converte direttamente in 103_{10}. Il caso apparentemente ambiguo è $10\,000\,000_2$, il cui complemento a 2, come ti ho già fatto osservare, è ancora $10\,000\,000_2$. Nessun tentennamento: si segue ciecamente la regola enunciata, e si converte quest'ultimo numero in decimale anteponendogli il segno $-$; il risultato è -128.

In conseguenza delle regole enunciate, l'intervallo di numeri interi rappresentabili con una cella di n bit va da -2^{n-1} a $2^{n-1}-1$. Questa convenzione si applica ai tipi `char`, `short int`, `int`, `long int` e `long long int`. Gli strani limiti della raprresentazione che ti ho puntigliosamente elencato nel paragrafo 4.1.1 vengono proprio da qui.

Un'ultima informazione, prima di chiudere questo paragrafo. L'uso di costanti ottali o esadecimali è così frequente da meritare una notazione *ad hoc*. Un numero intero senza segno preceduto dalla cifra `0` (zero) viene intepretato come ottale; un numero preceduto da `0x` viene intepretato come esadecimale. Ad esempio, la scrittura `0177777` corrisponde al decimale 65535, e lo stesso numero si può scrivere in esadecimale come `0xffff`. Va da sé che in una costante ottale le cifre 8 e 9 non possono comparire. La convenzione non è valida per tutti i linguaggi, ma per il linguaggio C funziona.

4.4 La rappresentazione in virgola mobile

Anzitutto, è d'obbligo chiarire una questione: il calcolatore non rappresenta tutti i numeri reali. Quello che viene messo in memoria assomiglia molto di più ai "numeri con la virgola" di cui ti ha parlato a lungo la maestra delle elementari. C'è solo qualche piccola complicazione in più – la notazione esponenziale – ma la sostanza è quella. I numeri reali dell'analisi sono solo un'astrazione.

Un po' brutale come inizio, ma serve a sottolineare subito che qualunque calcolo fatto con la rappresentazione di macchina dei numeri reali è soggetto ad un troncamento, e che questo introduce inevitabilmente un errore. Quali siano le conseguenze di questo errore dipende dall'algoritmo di calcolo che si usa. Non è mia intenzione qui entrare in questa materia, ma è bene non dimenticare che il problema c'è, e spesso si vede.

Fatta l'introduzione d'obbligo, veniamo alla rappresentazione. Si tratta anzitutto, anche qui, di cambiare base, passando da 10 a 2. Ma non funziona solo per gli interi? Neanche per sogno. Ricorda cosa ti ha insegnato la maestra. In termini brevi si riassume così: si considerano le unità, con dieci unità si fa una decina, con dieci decine si fa un centinaio, &c; per le parti frazionarie si divide in decimi, quello che resta in centesimi, poi in millesimi, &c, e si scrivono le cifre da sinistra a destra, dopo quella dell'unità; per convenzione, si mette una virgola che separa le unità dai decimi (altrimenti non si saprebbe da dove cominciare). Nell'ambiente scientifico è ormai universalmente adottata la convenzione anglosassone di usare il punto invece della virgola.

Ebbene, tutto questo si fa in qualunque base. In base 2 invece di parlare di decimi, centesimi &c si parla di mezzi, quarti, ottavi, sedicesimi &c — come in musica. In generale, se ho adottato una base b qualsiasi scomporrò un numero nella forma

$$\ldots + c_3 \times b^3 + c_2 \times b^2 + c_1 \times b^1 + c_0 \times b^0 + c_{-1} \times b^{-1} + c_{-2} \times b^{-2} + c_{-3} \times b^{-3} + \ldots$$

sempre con la convenzione che valga $0 \leq c_j < b$ per tutti i coefficienti c_j, e poi scriverò

$$\ldots c_3 c_2 c_1 c_0 \,.\, c_{-1} c_{-2} c_{-3} \ldots \ .$$

E come faccio a trovare le cfre c_{-1}, c_{-2} …? Semplice: basta adattare l'algoritmo per la divisione, sempre quello che ha usato la maestra per programmarti quando andavi alle scuole elementari: ad ogni passo calcoli una nuova cifra, solo si tratta di una cifra in base b invece che 10. Funziona benissimo! Ad esempio, prova a scrivere 1/3 in binario. Troverai (ricordandoti che $3_{10} = 11_2$):

$$\frac{1}{11} = 0.0101010101010101\ldots$$

Periodico, naturalmente, come tutti i numeri razionali. Se ci provi con un irrazionale la faccenda può essere un pochino più complicata, ma dato che ti devi comunque accontentare di un numero finito di cifre puoi ben prendere un'approssimazione razionale sufficientemente buona e convertire quella. Ad esempio, potrai scoprire che in binario si scrive

$$\begin{aligned}
\sqrt{2} &= 1.0110101000001001111001100110011111110011101111001100100\ldots \\
\pi &= 11.001001000011111101101010100010001000010110100011000010\ldots \\
e &= 10.101101111110000101010001011000101000101011101101001010\ldots
\end{aligned}$$

(ho scritto il 2 sotto radice in decimale, ma probabilmente se avessi scritto $\sqrt{10}$ avrei generato maggior confusione).

Stabilito che scrivere un numero reale in binario non è più difficile che scriverlo in decimale, veniamo alla rappresentazione di macchina. Il problema è il solito: i bit possono avere solo i valori 0 e 1: dove la mettiamo la virgola? Il biblico re Salomone avrebbe probabilmente preso una decisione del tutto arbitraria ma molto semplice: uso metà dei bit per la parte intera, e l'altra metà per la parte decimale. In altre parole, assumo che la virgola di separazione tra parte intera e parte frazionaria si trovi in una posizione fissa. Si chiama rappresentazione in *virgola fissa*. Ma ha il difetto di essere alquanto inefficiente: ad esempio, se lavorassi sistematicamente con numeri inferiori a 1 butterei regolarmente a mare tutti i bit della parte intera. Bisognerebbe trovare il modo di associare ad ogni numero una posizione diversa della virgola. In altre parole, ci vorrebbe una rappresentazione in *virgola mobile*. È possibile? Certamente, e intendo proprio spiegarti come.

Il trucco sta nell'utilizzare la notazione scientifica. Ad esempio, potresti scrivere il numero π, approssimato con un certo numero di cifre, in diversi modi:

$$
\begin{array}{rl}
\ldots\ldots & \\
314.159265358979323846264338327950\,29 & \times 10^{-2} \\
31.4159265358979323846264338327950\,29 & \times 10^{-1} \\
3.14159265358979323846264338327950\,29 & \times 10^{0} \\
0.314159265358979323846264338327950\,29 & \times 10^{1} \\
0.0314159265358979323846264338327950\,29 & \times 10^{2} \\
\ldots\ldots &
\end{array}
$$

Tutte le rappresentazioni qui sopra sono equivalenti, ma noi solitamente consideriamo quella sulla terza riga come più bella delle altre semplicemente perché se dimentichiamo il suffisso $\times 10\ldots$ troviamo che è l'unica che ha la parte intera costituita da una sola cifra non nulla — come dire che possiamo scrivere tutte le cifre evitando gli zeri inutili, pur di assumere che la virgola stia sempre dopo la prima cifra. Diremo che un numero scritto in questa forma è *normalizzato*. Ora, stabiliamo una convenzione: per rappresentare un numero lo poniamo in forma normalizzata, decidiamo di porre un limite, ad esempio di due cifre, all'esponente di 10, e scriviamo tre informazioni:

- il *segno*: scriviamo 0 al posto di $+$ e 1 al posto di $-$;
- l'*esponente* di 10 al quale sommiamo la base 49;
- la *mantissa*, ovvero la singola cifra intera più la parte frazionaria, senza alcun bisogno di aggiungere la virgola di separazione perché sappiamo comunque che la cifra intera è una sola.

Con queste convenzioni rappresenteremo il numero π – sempre in modo approssimato – come

$$(0, 49, 314159265358979323846264338327950\,29)$$

Aggiungere 49 all'esponente è utile perché ci permette di rappresentare anche esponenti negativi: con due cifre a disposizione il minimo possibile sarà -49 (che scriveremo come 0), ed il massimo sarà 50 (che scriveremo come 99). Il numero di cifre della mantissa naturalmente dipenderà da quante caselle abbiamo a disposizione per metterci delle cifre. Se, ad esempio, supponiamo di avere in tutto 10 caselle ne potremo usare una per il segno, due per l'esponente e le rimanenti 7 per la mantissa. Avremo così le rappresentazioni,

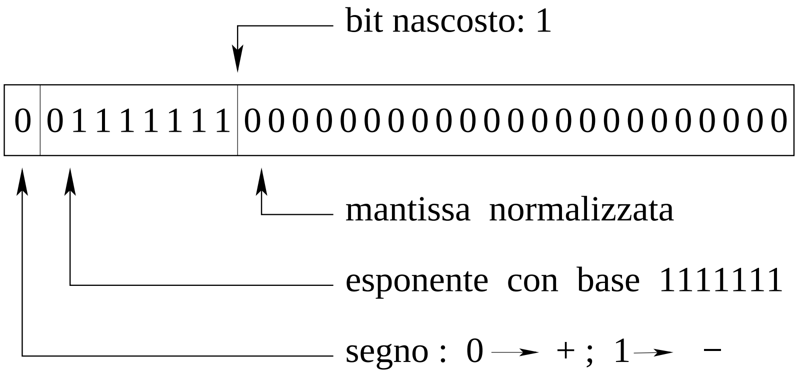

Figura 4.10. La rappresentazione in virgola mobile. Il caso considerato qui è quello del tipo `float`: 32 bit suddivisi tra segno (1 bit), esponente (8 bit) e mantissa (23 bit + 1 bit nascosto). Qui è rappresentato il valore 1.

con arrotondamento:

$$
\begin{array}{ll}
(0,49,3141593) & \pi\ , \\
(0,49,2718282) & e\ , \\
(0,49,1414214) & \sqrt{2}\ , \\
(0,72,6022169) & \text{numero di Avogadro}\ , \\
(0,25,1672614) & \text{massa del protone in grammi}\ , \\
(1,30,1602191) & \text{carica dell'elettrone in Coulomb}\ ,
\end{array}
$$

e potremmo continuare fin che abbiamo fiato.

Ebbene, questo è esattamente quanto si fa sul calcolatore, naturalmente lavorando in binario anziché in decimale, e aggiungendo un piccolo trucco (altrimenti sarebbe troppo semplice). Lo schema è rappresentato in figura 4.10, dove si fa riferimento al tipo `float`. Il bit più significativo della cella viene usato per il segno: 0 sta per + e 1 per −. Segue un campo di m bit per l'esponente, scegliendo come base il valore $011\ldots1$; in tal modo i valori possibili vengono equamente divisi tra positivi e negativi. [14] Il resto è mantissa normalizzata, ma qui arriva il trucco. Consideriamo come esempi i numeri 1, 2 e 1/2, che in binario si scrivono 1.00000000×10^0, 1.00000000×10^1, $1.00000000 \times 10^{-1}$ — qui 10 è inteso come binario. La forma normalizzata è proprio quella che ho scritto: non serve fare altro. Ma se la parte intera deve essere formata da una sola cifra binaria non nulla deve essere per forza 1; e allora… la dimentichiamo: è sottintesa. Questo è il *bit nascosto*, o *hidden bit*. In pratica abbiamo guadagnato un bit di mantissa, il che non guasta.

Supponiamo, come in figura 4.10 e come avviene in effetti per il tipo

[14] Ho impropriamente tradotto il termine *biased exponent* dei manuali tecnici con *esponente con base*. Chiamarlo, con traduzione più vicina a quella letterale, esponente truccato mi sembrava troppo teatrale.

`float`, di avere a disposizione 32 bit, e di dedicarne 8 all'esponente; ne restano 1 per il segno e 23 per la mantissa. Seguendo le regole enunciate sopra, dovremmo rappresentare i numeri 1, 2 e 1/2 come

$$\begin{array}{ll} (0, 01111111, 00000000000000000000000) & 1 \ , \\ (0, 10000000, 00000000000000000000000) & 2 \ , \\ (0, 01111110, 00000000000000000000000) & \dfrac{1}{2} \ . \end{array}$$

La rappresentazione di 1 è esattamente ciò che compare nella figura. Altri esempi? Eccoli:

$$\begin{array}{ll} (0, 10000000, 10010010000111111011100) & \pi \ , \\ (0, 10000000, 01011011111100001010101) & e \ , \\ (0, 01111111, 01101010000010011110111) & \sqrt{2} \ , \\ (0, 11001101, 11111110000110001111101) & \text{numero di Avogadro} \ , \\ (0, 00110000, 00000101101001100010 01) & \text{massa del protone (g)} \ , \\ (1, 01000000, 01111010010011101000000) & \text{carica dell'elettrone (C)} \ . \end{array}$$

Naturalmente in memoria non ci sono bit a forma di virgola o di parentesi, ma non è grave: la suddivisione della cella in campi è rigorosamente fissata dal tipo.

Per i tipi che utilizziamo col linguaggio C abbiamo:

- `float`: 32 bit suddivisi in 1 di segno, 8 di esponente e 24 effettivi di mantissa, tenuto conto del bit nascosto;
- `double`: 64 bit suddivisi in 1 di segno, 11 di esponente e 53 effettivi di mantissa, tenuto conto del bit nascosto;
- `long double`: 1 bit di segno, 15 di esponente ed il resto, più il bit nascosto, per la mantissa. Nel caso dei PC con processore Intel o simili il dato occupa 96 bit, ma floating point processor opera su una lunghezza totale di 80 bit, quindi la mantissa contiene 65 bit effettivi.

Tutto chiaro? Spero di sí, ma non credere che sia finita! Ci sono alcune configurazioni che assumono un significato particolare, e precisamente quando i bit del campo esponente sono tutti a zero, o tutti a 1. Questi casi, assieme ai limiti della rappresentazione, vengono illustrati in tabella 4.4.

Il caso di esponente nullo viene usato per rendere meno critico, nei limiti del possibile, il rischio di avere un *underflow* in un'operazione. Improvvisamente ci si dimentica sia della normalizzazione che del bit nascosto, e si considera la mantissa come un numero che ha la prima cifra della mantissa come parte intera, ed il resto come parte frazionaria. Questa scelta permette di estendere verso lo zero i limiti dell'intervallo dei numeri rappresentabili, come è illustrato dalle prime quattro righe della tabella 4.4. Il numero della terza riga è il minimo: al di sotto si incorre nell'*underflow*.

Il caso di esponente con tutti i bit a 1 viene usato per due scopi distinti: se la mantissa è zero, il numero viene considerato come $+\infty$ o $-\infty$, secondo il segno. Se

Tavola 4.4. Alcuni esempi che illustrano i casi in qualche modo particolari della rappresentazione: i limiti minimi e massimi, i valori $-\infty$, $+\infty$ e di un dato numerico illegale, indicato con NaN (Not a Number).

0	00000000	00000000000000000000100	$\rightarrow$	5.605194×10^{-45}
0	00000000	00000000000000000000010	$\rightarrow$	2.802597×10^{-45}
0	00000000	00000000000000000000001	$\rightarrow$	1.401298×10^{-45}
0	00000000	00000000000000000000000	$\rightarrow$	0.
0	11111110	11111111111111111111111	$\rightarrow$	$3.402823 \times 10^{+38}$
0	11111111	00000000000000000000000	$\rightarrow$	$+\infty$
1	11111110	11111111111111111111111	$\rightarrow$	$-3.402823 \times 10^{+38}$
1	00000000	00000000000000000000001	$\rightarrow$	$-1.401298 \times 10^{-45}$
1	11111111	00000000000000000000000	$\rightarrow$	$-\infty$
0	11111111	10000000000000000000000	$\rightarrow$	NaN
0	11111111	11111111111111111111111	$\rightarrow$	NaN
1	11111111	10000000000000000000000	$\rightarrow$	NaN
1	11111111	11111111111111111111111	$\rightarrow$	NaN

invece la mantissa è diversa da zero la rappresentazione viene considerata illegale, ed il dato è classificato come *Not a Number*, abbreviato in NaN.

A questo punto è facile indovinare qual sia il massimo numero positivo rappresentabile: il valore è dato dalla quinta riga della tabella. I limiti negativi sono simmetrici a quelli positivi: la sola differenza è nel bit di segno, che per i numeri negativi vale 1.

Tutto quanto ho detto l'ho illustrato con l'esempio del tipo `float` con 32 bit, ma si applica parola per parola a tutti i tipi: cambiano solo le lunghezze dei campi esponente e mantissa, ma il resto rimane valido.

Calcolare esattamente i limiti della rappresentazione è ormai solo questione solo di un po' di pazienza. Eseguiamo il calcolo per il tipo `float` che abbiamo usato fin qui come esempio.

Il numero minimo positivo rappresentabile è quello della terza riga della tabella 4.4. Se all'esponente 0 sottraiamo la base $1111111_2 = 127_{10}$ troviamo l'esponente -127_{10}, e dunque dobbiamo moltiplicare la mantissa per 2^{-127}. A sua volta la mantissa contiene 22 zeri seguiti da 1; uno di questi è la parte intera, quindi la mantissa rappresenta, con scrittura decimale, il numero 2^{-22}. Risultato: il numero della terza riga è 2^{-149}.

Per il limite massimo basta guardare la quinta riga. L'esponente è $11111110_2 = 254_{10}$; sottraendo la base 127 si ottiene l'esponente 127. Il calcolo per la mantissa è di poco più complicato, ma possiamo ben osservare che è praticamente 2. Se non ne sei convinto, nota che, tenuto conto del bit nascosto, la mantissa è $1.111111\ldots$, e sostituendo la rappresentazione finita con una somma infinita si ottiene la serie $1 + \frac{1}{2} + \frac{1}{4} + \ldots = 2$. Concludiamo che il numero massimo è 2^{128}.

Il calcolo dei limiti per i tipi `double` e long double si fa allo stesso modo, ma può essere più interessante ricavare una formula generale. Supponiamo di avere a disposizione p bit per l'esponente e m per la mantissa. La base per l'esponente è il numero binario $01\ldots1$ con $p-1$ bit con valore 1, ossia $2^{p-1}-1$. Il numero minimo ha 0 nel campo esponente, corrispondente ad un esponente effettivo $-2^{p-1}+1$, e mantissa formata da $m-1$ zeri seguiti da 1, ossia $2^{-(m-1)}$. Dunque il numero minimo rappresentabile è $2^{-2^{p-1}+1}\cdot 2^{-m+1} = 2^{-2^{p-1}-m+2}$. Il numero massimo ha nel campo esponente $1\ldots10$ con $p-1$ bit che hanno il valore 1, ossia 2^p-2, e quindi un esponente effettivo $2^p-2-2^{p-1}+1 = 2^{p-1}-1$, e nel campo mantissa ha m bit con valore 1 più il bit nascosto, che con l'argomento già usato si vede essere praticamente 2. Dunque il numero massimo è praticamente $2^{2^{p-1}-1}\cdot 2 = 2^{2^{p-1}}$.

Applicando le formule che abbiamo appena trovato possiamo valutare i limiti per i vari tipi:

- `float`: $p=8$ e $m=23$, limite di underflow 2^{-149}, limite di overflow 2^{128};
- `double`: $p=11$ e $m=52$, limite di underflow 2^{-1074}, limite di overflow 2^{1024};
- `long double` con uso effettivo di soli 80 bit, come su CPU Intel o simili: $p=15$ e $m=64$, limite di underflow 2^{-16446}, limite di overflow 2^{16384};